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 ........................................................................... 9
4 Command Line ........................................................... 17
5 Operators ................................................................. 33
6 Expressions ............................................................... 65
7 Simplification ............................................................. 101
8 Plotting ..................................................................... 113
9 Input and Output ........................................................ 137
10 Floating Point .......................................................... 169
11 Contexts ................................................................... 173
12 Polynomials .............................................................. 179
13 Constants .................................................................. 205
14 Logarithms ............................................................... 209
15 Trigonometric ........................................................... 213
16 Special Functions ...................................................... 221
17 Elliptic Functions ...................................................... 243
18 Limits ....................................................................... 249
19 Differentiation ........................................................... 251
20 Integration ............................................................... 263
21 Equations ................................................................. 285
22 Differential Equations ............................................... 303
23 Numerical ................................................................. 307
24 Arrays ....................................................................... 317
25 Matrices and Linear Algebra ......................................... 327
26 Affine ....................................................................... 351
27 itensor ...................................................................... 355
28 ctensor ..................................................................... 389
29 atensor ..................................................................... 415
30 Series ........................................................................ 419
31 Number Theory ........................................................ 433
32 Symmetries .............................................................. 443
33 Groups .................................................................... 461
34 Runtime Environment ............................................... 463
35 Miscellaneous Options .............................................. 467
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>Rules and Patterns</td>
<td>475</td>
</tr>
<tr>
<td>37</td>
<td>Lists</td>
<td>493</td>
</tr>
<tr>
<td>38</td>
<td>Sets</td>
<td>501</td>
</tr>
<tr>
<td>39</td>
<td>Structures</td>
<td>529</td>
</tr>
<tr>
<td>40</td>
<td>Function Definition</td>
<td>533</td>
</tr>
<tr>
<td>41</td>
<td>Program Flow</td>
<td>561</td>
</tr>
<tr>
<td>42</td>
<td>Debugging</td>
<td>573</td>
</tr>
<tr>
<td>43</td>
<td>asympa</td>
<td>581</td>
</tr>
<tr>
<td>44</td>
<td>augmented_lagrangian</td>
<td>583</td>
</tr>
<tr>
<td>45</td>
<td>bode</td>
<td>585</td>
</tr>
<tr>
<td>46</td>
<td>cobyla</td>
<td>587</td>
</tr>
<tr>
<td>47</td>
<td>contrib_ode</td>
<td>591</td>
</tr>
<tr>
<td>48</td>
<td>descriptive</td>
<td>599</td>
</tr>
<tr>
<td>49</td>
<td>diag</td>
<td>621</td>
</tr>
<tr>
<td>50</td>
<td>distrib</td>
<td>629</td>
</tr>
<tr>
<td>51</td>
<td>draw</td>
<td>665</td>
</tr>
<tr>
<td>52</td>
<td>drawdf</td>
<td>727</td>
</tr>
<tr>
<td>53</td>
<td>dynamics</td>
<td>731</td>
</tr>
<tr>
<td>54</td>
<td>ezunits</td>
<td>741</td>
</tr>
<tr>
<td>55</td>
<td>f90</td>
<td>757</td>
</tr>
<tr>
<td>56</td>
<td>finance</td>
<td>759</td>
</tr>
<tr>
<td>57</td>
<td>fractals</td>
<td>765</td>
</tr>
<tr>
<td>58</td>
<td>ggf</td>
<td>769</td>
</tr>
<tr>
<td>59</td>
<td>graphs</td>
<td>771</td>
</tr>
<tr>
<td>60</td>
<td>grobner</td>
<td>801</td>
</tr>
<tr>
<td>61</td>
<td>impdiff</td>
<td>809</td>
</tr>
<tr>
<td>62</td>
<td>implicit_plot</td>
<td>811</td>
</tr>
<tr>
<td>63</td>
<td>interpol</td>
<td>813</td>
</tr>
<tr>
<td>64</td>
<td>lapack</td>
<td>819</td>
</tr>
<tr>
<td>65</td>
<td>lbfgs</td>
<td>827</td>
</tr>
<tr>
<td>66</td>
<td>lindstedt</td>
<td>833</td>
</tr>
<tr>
<td>67</td>
<td>linearalgebra</td>
<td>835</td>
</tr>
<tr>
<td>68</td>
<td>lsquares</td>
<td>849</td>
</tr>
<tr>
<td>69</td>
<td>minpack</td>
<td>859</td>
</tr>
<tr>
<td>70</td>
<td>makeOrders</td>
<td>861</td>
</tr>
<tr>
<td>71</td>
<td>mnewton</td>
<td>863</td>
</tr>
<tr>
<td>72</td>
<td>numericalalio</td>
<td>865</td>
</tr>
<tr>
<td>73</td>
<td>opsubst</td>
<td>871</td>
</tr>
<tr>
<td>74</td>
<td>orthopoly</td>
<td>873</td>
</tr>
<tr>
<td>----</td>
<td>-----------</td>
<td>-----</td>
</tr>
<tr>
<td>75</td>
<td>plotdf</td>
<td>885</td>
</tr>
<tr>
<td>76</td>
<td>romberg</td>
<td>893</td>
</tr>
<tr>
<td>77</td>
<td>simplex</td>
<td>897</td>
</tr>
<tr>
<td>78</td>
<td>simplification</td>
<td>899</td>
</tr>
<tr>
<td>79</td>
<td>solve_rec</td>
<td>909</td>
</tr>
<tr>
<td>80</td>
<td>stats</td>
<td>913</td>
</tr>
<tr>
<td>81</td>
<td>stirling</td>
<td>931</td>
</tr>
<tr>
<td>82</td>
<td>stringproc</td>
<td>933</td>
</tr>
<tr>
<td>83</td>
<td>unit</td>
<td>945</td>
</tr>
<tr>
<td>84</td>
<td>zeilberger</td>
<td>955</td>
</tr>
<tr>
<td>85</td>
<td>Indices</td>
<td>959</td>
</tr>
<tr>
<td>A</td>
<td>Function and Variable Index</td>
<td>961</td>
</tr>
</tbody>
</table>
# Table of Contents

1 Introduction to Maxima .......................... 1

2 Bug Detection and Reporting ................. 7
   2.1 Functions and Variables for Bug Detection and Reporting .... 7

3 Help ............................................. 9
   3.1 Lisp and Maxima .................................. 9
   3.2 Garbage Collection .............................. 10
   3.3 Documentation .................................. 10
   3.4 Functions and Variables for Help .................. 11

4 Command Line .................................. 17
   4.1 Introduction to Command Line .................... 17
   4.2 Functions and Variables for Command Line .......... 21

5 Operators ...................................... 33
   5.1 nary ............................................ 33
   5.2 nofix ........................................... 33
   5.3 postfix .......................................... 33
   5.4 prefix .......................................... 33
   5.5 Arithmetic operators ............................ 33
   5.6 Relational operators ............................ 36
   5.7 General operators ................................ 38

6 Expressions .................................... 65
   6.1 Introduction to Expressions ..................... 65
   6.2 Complex ......................................... 65
   6.3 Nouns and Verbs .................................. 66
   6.4 Identifiers ....................................... 67
   6.5 Strings .......................................... 68
   6.6 Inequality ....................................... 68
   6.7 Syntax .......................................... 68
   6.8 Functions and Variables for Expressions .......... 71

7 Simplification ................................. 101
   7.1 Functions and Variables for Simplification .......... 101
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>itensor</td>
<td>355</td>
</tr>
<tr>
<td>27.1</td>
<td>Introduction to itensor</td>
<td>355</td>
</tr>
<tr>
<td>27.1.1</td>
<td>New tensor notation</td>
<td>356</td>
</tr>
<tr>
<td>27.1.2</td>
<td>Indicial tensor manipulation</td>
<td>356</td>
</tr>
<tr>
<td>27.2</td>
<td>Functions and Variables for itensor</td>
<td>359</td>
</tr>
<tr>
<td>27.2.1</td>
<td>Managing indexed objects</td>
<td>359</td>
</tr>
<tr>
<td>27.2.2</td>
<td>Tensor symmetries</td>
<td>368</td>
</tr>
<tr>
<td>27.2.3</td>
<td>Indicial tensor calculus</td>
<td>369</td>
</tr>
<tr>
<td>27.2.4</td>
<td>Tensors in curved spaces</td>
<td>374</td>
</tr>
<tr>
<td>27.2.5</td>
<td>Moving frames</td>
<td>377</td>
</tr>
<tr>
<td>27.2.6</td>
<td>Torsion and nonmetricity</td>
<td>380</td>
</tr>
<tr>
<td>27.2.7</td>
<td>Exterior algebra</td>
<td>382</td>
</tr>
<tr>
<td>27.2.8</td>
<td>Exporting TeX expressions</td>
<td>386</td>
</tr>
<tr>
<td>27.2.9</td>
<td>Interfacing with ctensor</td>
<td>386</td>
</tr>
<tr>
<td>27.2.10</td>
<td>Reserved words</td>
<td>387</td>
</tr>
<tr>
<td>28</td>
<td>ctensor</td>
<td>389</td>
</tr>
<tr>
<td>28.1</td>
<td>Introduction to ctensor</td>
<td>389</td>
</tr>
<tr>
<td>28.2</td>
<td>Functions and Variables for ctensor</td>
<td>391</td>
</tr>
<tr>
<td>28.2.1</td>
<td>Initialization and setup</td>
<td>391</td>
</tr>
<tr>
<td>28.2.2</td>
<td>The tensors of curved space</td>
<td>394</td>
</tr>
<tr>
<td>28.2.3</td>
<td>Taylor series expansion</td>
<td>396</td>
</tr>
<tr>
<td>28.2.4</td>
<td>Frame fields</td>
<td>399</td>
</tr>
<tr>
<td>28.2.5</td>
<td>Algebraic classification</td>
<td>399</td>
</tr>
<tr>
<td>28.2.6</td>
<td>Torsion and nonmetricity</td>
<td>402</td>
</tr>
<tr>
<td>28.2.7</td>
<td>Miscellaneous features</td>
<td>403</td>
</tr>
<tr>
<td>28.2.8</td>
<td>Utility functions</td>
<td>405</td>
</tr>
<tr>
<td>28.2.9</td>
<td>Variables used by ctensor</td>
<td>410</td>
</tr>
<tr>
<td>28.2.10</td>
<td>Reserved names</td>
<td>413</td>
</tr>
<tr>
<td>28.2.11</td>
<td>Changes</td>
<td>414</td>
</tr>
<tr>
<td>29</td>
<td>atensor</td>
<td>415</td>
</tr>
<tr>
<td>29.1</td>
<td>Introduction to atensor</td>
<td>415</td>
</tr>
<tr>
<td>29.2</td>
<td>Functions and Variables for atensor</td>
<td>416</td>
</tr>
<tr>
<td>30</td>
<td>Series</td>
<td>419</td>
</tr>
<tr>
<td>30.1</td>
<td>Introduction to Series</td>
<td>419</td>
</tr>
<tr>
<td>30.2</td>
<td>Functions and Variables for Series</td>
<td>419</td>
</tr>
<tr>
<td>30.3</td>
<td>Poisson series</td>
<td>431</td>
</tr>
<tr>
<td>31</td>
<td>Number Theory</td>
<td>433</td>
</tr>
<tr>
<td>31.1</td>
<td>Functions and Variables for Number Theory</td>
<td>433</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>32</td>
<td>Symmetries</td>
<td>443</td>
</tr>
<tr>
<td>32.1</td>
<td>Introduction to Symmetries</td>
<td>443</td>
</tr>
<tr>
<td>32.2</td>
<td>Functions and Variables for Symmetries</td>
<td>443</td>
</tr>
<tr>
<td>32.2.1</td>
<td>Changing bases</td>
<td>443</td>
</tr>
<tr>
<td>32.2.2</td>
<td>Changing representations</td>
<td>447</td>
</tr>
<tr>
<td>32.2.3</td>
<td>Groups and orbits</td>
<td>448</td>
</tr>
<tr>
<td>32.2.4</td>
<td>Partitions</td>
<td>451</td>
</tr>
<tr>
<td>32.2.5</td>
<td>Polynomials and their roots</td>
<td>452</td>
</tr>
<tr>
<td>32.2.6</td>
<td>Resolvents</td>
<td>453</td>
</tr>
<tr>
<td>32.2.7</td>
<td>Miscellaneous</td>
<td>459</td>
</tr>
<tr>
<td>33</td>
<td>Groups</td>
<td>461</td>
</tr>
<tr>
<td>33.1</td>
<td>Functions and Variables for Groups</td>
<td>461</td>
</tr>
<tr>
<td>34</td>
<td>Runtime Environment</td>
<td>463</td>
</tr>
<tr>
<td>34.1</td>
<td>Introduction for Runtime Environment</td>
<td>463</td>
</tr>
<tr>
<td>34.2</td>
<td>Interrupts</td>
<td>463</td>
</tr>
<tr>
<td>34.3</td>
<td>Functions and Variables for Runtime Environment</td>
<td>463</td>
</tr>
<tr>
<td>35</td>
<td>Miscellaneous Options</td>
<td>467</td>
</tr>
<tr>
<td>35.1</td>
<td>Introduction to Miscellaneous Options</td>
<td>467</td>
</tr>
<tr>
<td>35.2</td>
<td>Share</td>
<td>467</td>
</tr>
<tr>
<td>35.3</td>
<td>Functions and Variables for Miscellaneous Options</td>
<td>467</td>
</tr>
<tr>
<td>36</td>
<td>Rules and Patterns</td>
<td>475</td>
</tr>
<tr>
<td>36.1</td>
<td>Introduction to Rules and Patterns</td>
<td>475</td>
</tr>
<tr>
<td>36.2</td>
<td>Functions and Variables for Rules and Patterns</td>
<td>475</td>
</tr>
<tr>
<td>37</td>
<td>Lists</td>
<td>493</td>
</tr>
<tr>
<td>37.1</td>
<td>Introduction to Lists</td>
<td>493</td>
</tr>
<tr>
<td>37.2</td>
<td>Functions and Variables for Lists</td>
<td>493</td>
</tr>
<tr>
<td>38</td>
<td>Sets</td>
<td>501</td>
</tr>
<tr>
<td>38.1</td>
<td>Introduction to Sets</td>
<td>501</td>
</tr>
<tr>
<td>38.1.1</td>
<td>Usage</td>
<td>501</td>
</tr>
<tr>
<td>38.1.2</td>
<td>Set Member Iteration</td>
<td>503</td>
</tr>
<tr>
<td>38.1.3</td>
<td>Bugs</td>
<td>504</td>
</tr>
<tr>
<td>38.1.4</td>
<td>Authors</td>
<td>505</td>
</tr>
<tr>
<td>38.2</td>
<td>Functions and Variables for Sets</td>
<td>505</td>
</tr>
<tr>
<td>39</td>
<td>Structures</td>
<td>529</td>
</tr>
<tr>
<td>39.1</td>
<td>Introduction to Structures</td>
<td>529</td>
</tr>
<tr>
<td>39.2</td>
<td>Functions and Variables for Structures</td>
<td>529</td>
</tr>
</tbody>
</table>
57 fractals ........................................ 765
  57.1 Introduction to fractals .................... 765
  57.2 Definitions for IFS fractals ............... 765
  57.3 Definitions for complex fractals .......... 766
  57.4 Definitions for Koch snowflakes .......... 767
  57.5 Definitions for Peano maps ............... 767

58 ggf ........................................... 769
  58.1 Functions and Variables for ggf .......... 769

59 graphs ....................................... 771
  59.1 Introduction to graphs ................... 771
  59.2 Functions and Variables for graphs ...... 771
     59.2.1 Building graphs ...................... 771
     59.2.2 Graph properties .................... 777
     59.2.3 Modifying graphs ..................... 792
     59.2.4 Reading and writing to files .......... 794
     59.2.5 Visualization ......................... 795

60 grobner ...................................... 801
  60.1 Introduction to grobner .................. 801
     60.1.1 Notes on the grobner package ....... 801
     60.1.2 Implementations of admissible monomial orders in
grobner ........................................ 801
  60.2 Functions and Variables for grobner .... 802
     60.2.1 Global switches for grobner ...... 802
     60.2.2 Simple operators in grobner ...... 803
     60.2.3 Other functions in grobner ....... 804
     60.2.4 Standard postprocessing of Groebner Bases .... 805

61 impdiff ...................................... 809
  61.1 Functions .................................. 809

62 implicit_plot .................................. 811
  62.1 Functions and Variables for implicit_plot .. 811

63 interpol ..................................... 813
  63.1 Introduction to interpol .................. 813
  63.2 Functions and Variables for interpol ...... 813

64 lapack ....................................... 819
  64.1 Introduction to lapack .................... 819
  64.2 Functions and Variables for lapack ....... 819
65 lbfgs ........................................ 827
   65.1 Introduction to lbfgs .......................... 827
   65.2 Functions and Variables for lbfgs .......... 827

66 lindstedt .................................... 833
   66.1 Functions and Variables for lindstedt .... 833

67 linearalgebra ................................. 835
   67.1 Introduction to linearalgebra ............... 835
   67.2 Functions and Variables for linearalgebra 836

68 lsquares ..................................... 849
   68.1 Introduction to lsquares ..................... 849
   68.2 Functions and Variables for lsquares ...... 849

69 minpack ...................................... 859
   69.1 Introduction to minpack ...................... 859
   69.2 Functions and Variables for minpack ...... 859

70 makeOrders ................................... 861
   70.1 Functions and Variables for makeOrders ... 861

71 mnewton ..................................... 863
   71.1 Introduction to mnewton ..................... 863
   71.2 Functions and Variables for mnewton ...... 863

72 numericalio ................................... 865
   72.1 Introduction to numericalio ................. 865
       72.1.1 Plain-text input and output .......... 865
       72.1.2 Separator flag values for input ...... 865
       72.1.3 Separator flag values for output ..... 865
       72.1.4 Binary floating-point input and output 866
   72.2 Functions and Variables for plain-text input and output ........................................ 866
   72.3 Functions and Variables for binary input and output ... 868

73 opsubst ...................................... 871
   73.1 Functions and Variables for opsubst ........ 871
<table>
<thead>
<tr>
<th>74</th>
<th>orthopoly</th>
<th>.........................</th>
<th>873</th>
</tr>
</thead>
<tbody>
<tr>
<td>74.1</td>
<td>Introduction to orthogonal polynomials</td>
<td>.........................</td>
<td>873</td>
</tr>
<tr>
<td>74.1.1</td>
<td>Getting Started with orthopoly</td>
<td>.........................</td>
<td>873</td>
</tr>
<tr>
<td>74.1.2</td>
<td>Limitations</td>
<td>................................</td>
<td>875</td>
</tr>
<tr>
<td>74.1.3</td>
<td>Floating point Evaluation</td>
<td>................................</td>
<td>877</td>
</tr>
<tr>
<td>74.1.4</td>
<td>Graphics and orthopoly</td>
<td>.........................</td>
<td>878</td>
</tr>
<tr>
<td>74.1.5</td>
<td>Miscellaneous Functions</td>
<td>.........................</td>
<td>879</td>
</tr>
<tr>
<td>74.1.6</td>
<td>Algorithms</td>
<td>................................</td>
<td>880</td>
</tr>
<tr>
<td>74.2</td>
<td>Functions and Variables for orthogonal polynomials</td>
<td>.........................</td>
<td>880</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>75</th>
<th>plotdf</th>
<th>.........................</th>
<th>885</th>
</tr>
</thead>
<tbody>
<tr>
<td>75.1</td>
<td>Introduction to plotdf</td>
<td>.........................</td>
<td>885</td>
</tr>
<tr>
<td>75.2</td>
<td>Functions and Variables for plotdf</td>
<td>.........................</td>
<td>885</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>76</th>
<th>romberg</th>
<th>.........................</th>
<th>893</th>
</tr>
</thead>
<tbody>
<tr>
<td>76.1</td>
<td>Functions and Variables for romberg</td>
<td>.........................</td>
<td>893</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>77</th>
<th>simplex</th>
<th>.........................</th>
<th>897</th>
</tr>
</thead>
<tbody>
<tr>
<td>77.1</td>
<td>Introduction to simplex</td>
<td>.........................</td>
<td>897</td>
</tr>
<tr>
<td>77.2</td>
<td>Functions and Variables for simplex</td>
<td>.........................</td>
<td>897</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>78</th>
<th>simplification</th>
<th>.........................</th>
<th>899</th>
</tr>
</thead>
<tbody>
<tr>
<td>78.1</td>
<td>Introduction to simplification</td>
<td>.........................</td>
<td>899</td>
</tr>
<tr>
<td>78.2</td>
<td>Package absimp</td>
<td>.........................</td>
<td>899</td>
</tr>
<tr>
<td>78.3</td>
<td>Package facexp</td>
<td>.........................</td>
<td>899</td>
</tr>
<tr>
<td>78.4</td>
<td>Package functs</td>
<td>.........................</td>
<td>901</td>
</tr>
<tr>
<td>78.5</td>
<td>Package ineq</td>
<td>.........................</td>
<td>904</td>
</tr>
<tr>
<td>78.6</td>
<td>Package rducon</td>
<td>.........................</td>
<td>905</td>
</tr>
<tr>
<td>78.7</td>
<td>Package scifac</td>
<td>.........................</td>
<td>906</td>
</tr>
<tr>
<td>78.8</td>
<td>Package sqdnst</td>
<td>.........................</td>
<td>906</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>79</th>
<th>solve_rec</th>
<th>.........................</th>
<th>909</th>
</tr>
</thead>
<tbody>
<tr>
<td>79.1</td>
<td>Introduction to solve_rec</td>
<td>.........................</td>
<td>909</td>
</tr>
<tr>
<td>79.2</td>
<td>Functions and Variables for solve_rec</td>
<td>.........................</td>
<td>909</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>80</th>
<th>stats</th>
<th>.........................</th>
<th>913</th>
</tr>
</thead>
<tbody>
<tr>
<td>80.1</td>
<td>Introduction to stats</td>
<td>.........................</td>
<td>913</td>
</tr>
<tr>
<td>80.2</td>
<td>Functions and Variables for inference_result</td>
<td>.........................</td>
<td>913</td>
</tr>
<tr>
<td>80.3</td>
<td>Functions and Variables for stats</td>
<td>.........................</td>
<td>915</td>
</tr>
<tr>
<td>80.4</td>
<td>Functions and Variables for special distributions</td>
<td>.........................</td>
<td>929</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>81</th>
<th>stirling</th>
<th>.........................</th>
<th>931</th>
</tr>
</thead>
<tbody>
<tr>
<td>81.1</td>
<td>Functions and Variables for stirling</td>
<td>.........................</td>
<td>931</td>
</tr>
</tbody>
</table>
82 stringproc ........................................ 933
  82.1 Introduction to string processing ............... 933
  82.2 Functions and Variables for input and output ..... 934
  82.3 Functions and Variables for characters .......... 937
  82.4 Functions and Variables for strings .......... 938

83 unit ............................................ 945
  83.1 Introduction to Units ................................ 945
  83.2 Functions and Variables for Units ............... 946

84 zeilberger ....................................... 955
  84.1 Introduction to zeilberger ....................... 955
    84.1.0.1 The indefinite summation problem .... 955
    84.1.0.2 The definite summation problem .... 955
    84.1.1 Verbosity levels ........................... 955
  84.2 Functions and Variables for zeilberger ...... 956
  84.3 General global variables ........................ 957
  84.4 Variables related to the modular test .... 958

85 Indices ......................................... 959

Appendix A Function and Variable Index .... 961
Chapter 1: Introduction to Maxima

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:

\begin{verbatim}
(%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
\end{verbatim}

\begin{verbatim}
-- 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.
\end{verbatim}

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:

\begin{verbatim}
(%i1) u: expand ((x + y)^6);
6 5 2 4 3 3 4 2 5 6
(%o1) y + 6 x y + 15 x y + 20 x y + 15 x y + 6 x y + x
(%i2) diff (u, x);
5 4 2 3 3 2 4 5
(%o2) 6 y + 30 x y + 60 x y + 60 x y + 30 x y + 6 x
(%i3) factor (\%o2);
5
(%o3) 6 (y + x)
\end{verbatim}
Maxima knows about complex numbers and numerical constants:

\begin{verbatim}
(%i1) cos(%pi);
(%o1) - 1
(%i2) exp(%i*%pi);
(%o2) - 1
\end{verbatim}

Maxima can do differential and integral calculus:

\begin{verbatim}
(%i1) u: expand ((x + y)^6);
6 5 2 4 3 3 4 2 5 6
y + 6 x y + 15 x y + 20 x y + 15 x y + 6 x y + x
(%i2) diff (%, x);
5 4 2 3 3 2 4 5
6 y + 30 x y + 60 x y + 60 x y + 30 x y + 6 x
(%i3) integrate (1/(1 + x^3), x);
\frac{2 \cdot x - 1}{6 \sqrt{3}} \cdot \arctan\left(\frac{x}{\sqrt{3}}\right) + \frac{\log(x - x + 1)}{3 \sqrt{3}} + \frac{\log(x + 1)}{3}
\end{verbatim}

Maxima can solve linear systems and cubic equations:

\begin{verbatim}
(%i1) linsolve ([3*x + 4*y = 7, 2*x + a*y = 13], [x, y]);
\frac{7 \cdot a - 52}{3 \cdot a - 8} \quad \frac{25}{3 \cdot a - 8}
(%o1) \left[ x = \frac{7 \cdot a - 52}{3 \cdot a - 8}, \quad y = \frac{25}{3 \cdot a - 8} \right]
(%i2) solve (x^3 - 3*x^2 + 5*x = 15, x);
\{x = - \sqrt{5} \cdot %i, \quad x = \sqrt{5} \cdot %i, \quad x = 3\}
\end{verbatim}

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 ;.

\begin{verbatim}
(%i1) eq_1: x^2 + 3*x*y + y^2 = 0$
(%i2) eq_2: 3*x + y = 1$
(%i3) solve ([eq_1, eq_2]);
\{x = - \frac{\sqrt{5} \cdot %i}{2}, \quad x = \frac{\sqrt{5} \cdot %i}{2} \}
\end{verbatim}

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])$

\sin(\sqrt{y^2+x^2})/\sqrt{y^2+x^2}$
2 Bug Detection and Reporting

2.1 Functions and Variables for Bug Detection and Reporting

\texttt{run_testsuite ([options])}

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.

\texttt{run_testsuite} takes the following optional keyword arguments

- \textit{display_all} Display all tests. Normally, the tests are not displayed, unless the test fails. (Defaults to \texttt{false}).

- \textit{display_known_bugs} Displays tests that are marked as known bugs. (Default is \texttt{false}).

- \textit{tests} This is 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 \texttt{testsuite_files}.

- \textit{time} Display time information. If \texttt{true}, the time taken for each test file is displayed. If \texttt{all}, the time for each individual test is shown if \texttt{display_all} is \texttt{true}. The default is \texttt{false}, so no timing information is shown.

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

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

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

\texttt{run_testsuite} returns \texttt{done}.

\textit{testsuite_files}

\texttt{testsuite_files} is the set of tests to be run by \texttt{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:

\begin{verbatim}
["rtest13s", ["rtest14", 57, 63]]
\end{verbatim}

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

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** ()

Prints out a summary of the parameters of the Maxima build.

**build_info** returns an empty string "".
Chapter 3: Help

3 Help

3.1 Lisp and Maxima

Maxima is written in Lisp, and it is easy to access 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.
\begin{verbatim}
(msetq $foo #\[x, y\])
This has the same effect as entering
(%i1) foo: [x, y];

The Lisp function \texttt{displa} prints an expression in Maxima format.

(%i1) :lisp #\[x, y, z\]$
(\texttt{MLIST SIMP}) \texttt{X Y Z}$
(%i1) :lisp (displa '((\texttt{MLIST SIMP}) \texttt{X Y Z}))
\[x, y, z\] NIL

Functions defined in Maxima are not ordinary Lisp functions. The Lisp function \texttt{mfuncall} calls a Maxima function. For example:

(%i1) foo(x,y) := x*y$
(%i2) :lisp (mfuncall '$foo 'a 'b)
(\texttt{MTIMES SIMP}) A B

Some Lisp functions are shadowed in the Maxima package, namely the following. \texttt{complement}, \texttt{continue}, \texttt{/, float}, \texttt{functionp}, \texttt{array}, \texttt{exp}, \texttt{listen}, \texttt{signum}, \texttt{atan}, \texttt{asin}, \texttt{acos}, \texttt{asinh}, \texttt{acosh}, \texttt{atanh}, \texttt{tanh}, \texttt{cosh}, \texttt{sinh}, \texttt{tan}, \texttt{break}, and \texttt{gcd}.

\section*{3.2 Garbage Collection}

Symbolic computation tends to create a good deal of garbage (temporary or intermediate results that are eventually not used), and effective handling of this can be crucial to successful completion of some programs.

Under GCL, on UNIX systems where the mprotect system call is available (including SUN OS 4.0 and some variants of BSD) a stratified garbage collection is available. This limits the collection to pages which have been recently written to. See the GCL documentation under ALLOCATE and GBC. At the Lisp level doing (setq si::*notify-gbc* t) will help you determine which areas might need more space.

For other Lisps that run Maxima, we refer the reader to the documentation for that Lisp on how to control GC.

\section*{3.3 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 \texttt{?} command (i.e., the \texttt{describe} function). The user’s manual is viewed as info hypertext by the \texttt{info} viewer program and as a web page by any ordinary web browser.

\texttt{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))
\end{verbatim}
Chapter 3: Help

3.4 Functions and Variables for Help

**apropos (string)**  
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:  
```lisp
(%i1) apropos("gamma");  
(%o1) [%gamma, gamma, gammalim, gamma_expand, gamma_greek,  
gamma_incomplete, gamma_incomplete_generalized,  
gamma_incomplete_regularized, Gamma, log_gamma, makegamma,  
prefer_gamma_incomplete,  
gamma_incomplete_generalized_regularized]
```

**demo (filename)**  
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:  
```lisp
(%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)

(%i5) demo ("rncomb");

batching /home/wfs/maxima/share/simplification/rncomb.dem
At the _ prompt, type ';' followed by enter to get next demo

(%i6) load(rncomb)

(%i7) exp1 : ----- + ---------
     y + x  2 (y + x)

(%o7) ----- + ---------
     y + x  2 (y + x)

(%i8) combine(exp1)

(%o8) ----- + ---------
     y + x  2 (y + x)

(%i9) rncombine(%)

(%o9) 2 z + x

2 (y + x)

(%i10) exp2 : - + - + - + -
     3 3 2 2

(%o10) - + - + - + -
     3 3 2 2

(%i11) combine(exp2)
\begin{verbatim}
(%o11) 2 d + 2 c + 3 (b + a) --------------- 6

(%i12) rnc combine(exp2)

(%o12) 2 d + 2 c + 3 b + 3 a --------------- 6

(%i13)

describe (string) Function
describe (string, exact) Function
describe (string, inexact) Function
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 3.3 [Documentation], page 10.

Example:

(%i11) ?? 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: nonnegativeintegerp (Functions and Variables for linearalgebra)
Enter space-separated numbers, ‘all’ or ‘none’: 7 8

-- Function: integrate (<expr>, <x>)
\end{verbatim}
-- 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 (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([x+y,0,-3.2],[2.5E+20,x])
(%o2) [y + x, 0, - 3.2, 2.5E+20, x]
(%o2) done
(%i3) example("lambda");
(%i4) lambda([x,y,z],z^2+y^2+x^2)
2 2 2
(%o4) lambda([x, y, z], z + y + x )
(%i5) (%(1,2,a)
2
(%o5) a + 5
(%i6) a+2+1
(%o6) a + 3
(%o6) done
(%i7) example("allROOTS");
(%i8) (1+2*x)^3 = 13.5*(1+x^5)
3 5
(%o8) (2 x + 1) = 13.5 (x + 1)
\begin{verbatim}
(\%) allroots(%)
(\%) [x = .8296749902129361, x = -1.015755543828121, 
x = .9659625152196369 \%i - .4069597231924075, 
x = - .9659625152196369 \%i - .4069597231924075, x = 1.0]
(\%) done
\end{verbatim}

**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

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 ‘’ and nouns.

Examples:

Applied to a symbol, the single quote prevents evaluation of the symbol.

\[
\begin{align*}
(\%i1) \text{aa: 1024;} & \quad 1024 \\
(\%o1) & \\
(\%i2) \text{aa^2;} & \quad 1048576 \\
(\%o2) & \\
(\%i3) \text{'aa^2;} & \quad 2 \\
(\%o3) & \\
(\%i4) \text{''%;} & \quad 1048576 \\
(\%o4) &
\end{align*}
\]

Applied to a function call, the single quote prevents evaluation of the function call. The result is the noun form of the function call.

\[
\begin{align*}
(\%i1) \text{x0: 5;} & \quad 5 \\
(\%o1) & \\
(\%i2) \text{x1: 7;} & \quad 7 \\
(\%o2) & \\
(\%i3) \text{integrate (x^2, x, x0, x1);} & \quad 218 \\
(\%o3) & -- \\
(\%i4) \text{’integrate (x^2, x, x0, x1);} & \quad 3 \\
(\%o4) &
\end{align*}
\]
Applied to a parenthesized expression, the single quote prevents evaluation of all symbols and function calls in the expression.

\begin{verbatim}
(%i1) aa: 1024;
(%o1) 1024
(%i2) bb: 19;
(%o2) 19
(%i3) sqrt(aa) + bb;
(%o3) 51
(%i4) '(sqrt(aa) + bb);
(%o4) bb + sqrt(aa)
(%i5) ' %;
(%o5) 51
\end{verbatim}

The single quote does not prevent simplification.

\begin{verbatim}
(%i1) sin (17 * %pi) + cos (17 * %pi);
(%o1) - 1
(%i2) '(sin (17 * %pi) + cos (17 * %pi));
(%o2) - 1
\end{verbatim}

Maxima considers floating point operations by its in-built mathematical functions to be a simplification.

\begin{verbatim}
(%i1) sin(1.0);
(%o1) .8414709848078965
(%i2) '(sin(1.0));
(%o2) .8414709848078965
\end{verbatim}

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 the single-quote operator `'` 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.

```lisp
(%i1) expand ((a + b)^3);
     3 2 2 3
(\(%o1\) \(\text{b + 3 a b + 3 a b + a}\))
(%i2) [_, ''_];
     3 3 2 2 3
(\(%o2\) \([\text{expand((b + a) ), b + 3 a b + 3 a b + a}]\))
(%i3) [\%i1, ''\%i1];
     3 3 2 2 3
(\(%o3\) \([\text{expand((b + a) ), b + 3 a b + 3 a b + a}]\))
(%i4) [aa : cc, bb : dd, cc : 17, dd : 29];
   [cc, dd, 17, 29]
(%i5) foo_1 (x) := aa - bb * x;
   foo_1(x) := aa - bb x
(%i6) foo_1 (10);
   cc - 10 dd
(%i7) ''%;
   - 273
(%i8) ''(foo_1 (10));
   - 273
(%i9) foo_2 (x) := ''aa - ''bb * x;
   foo_2(x) := cc - dd x
(%i10) foo_2 (10);
   - 273
(%i11) [x0 : x1, x1 : x2, x2 : x3];
   [x1, x2, x3]
(%i12) x0;
   x1
(%i13) ''x0;
   x2
(%i14) ''''x0;
   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).

```lisp
(%i1) declare (foo, noun);
   done
(%i2) foo (x) := x - 1729;
   `foo(x) := x - 1729
(%i3) foo (100);
   foo(100)
(%i4) '''foo (100);
   - 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);
    foo_1a(x) := x log(x) - x
    foo_1b(x) := integrate(log(x), x)
(%o3) [%t3, %t4]
(%i4) integrate (log (x), x);
(%o4) x log(x) - x
(%i5) foo_2a (x) := ''%;
(%o5) foo_2a(x) := %
(%i6) foo_2b (x) := %;
(%o6) foo_2b(x) := %
(%i7) dispfun (foo_2a, foo_2b);
    foo_2a(x) := x log(x) - x
    foo_2b(x) := %
(%o7) [%t7, %t8]
(%i8) F : lambda ([u], diff (sin (u), u));
(%o8) lambda([u], diff(sin(u), u))
(%i9) G : lambda ([u], ''(diff (sin (u), u)));

(%o9) lambda([u], diff(sin(u), u))
4.2 Functions and Variables for Command Line

**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.

**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.

**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 supresses 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 determin-
     ant 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.
• **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 (see `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 (see `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) := 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 array 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(ev(x)).

5. For each instance of ev in the arguments, steps (3) and (4) are repeated.

Examples:

```lisp
(%i1) sin(x) + cos(y) + (w+1)^2 + 'diff (sin(w), w);
   2
d  2
(%o1) cos(y) + sin(x) + -- (sin(w)) + (w + 1)
dw
(%i2) ev (%o1, 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.

```lisp
(%i3) programmode: false;
(%o3) false
(%i4) x+y, x: a+y, y: 2;
(%o4) y + a + 2
(%i5) 2*x - 3*y = 3$
(%i6) -3*x + 2*y = -4$
(%i7) solve ([%o5, %o6]);
Solution

   1
(%t7) y = - -
       5

   6
(%t8) x = -
       5
(%o8) [%t7, %t8]
(%i9) %o6, %o8;
(%o9) - 4 = - 4
(%i10) x + 1/x > gamma (1/2);
```
1

\( x + \frac{1}{x} > \sqrt{\pi} \)

\( x \)

\%(o9) \% \), numer, x=1/2;
\%(o10) 2.5 > 1.772453850905516
\%(o11) \% \), 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
- ifeval
- isolate_wrt_times
- keepfloat
- letrat
- listarith
- logabs
- logarc
- logexpand
- lognegint
- lognumer
- m1pbranch
- numer_pbranch
- programmode
- radexpand
- ratalgdenom
- ratfac
- ratmx
- ratsimpexpons
- simp
- simpproduct
- simpsum
- sumexpand
- trigexpand

Examples:

\%(i1) sin (1/2);
\%(o1) sin(-)
\%(i2) sin (1/2), float;
\%(o2) sin(1/2) = 0.479425538604203
\%(i3) sin (1/2), float=true;
\%(o3) sin(1/2) = 0.479425538604203
\%(i4) simp : false;
\%(o4) simp = false
\%(i5) 1 + 1;
Chapter 4: Command Line

(%o5) 1 + 1
(%i6) 1 + 1, simp;
(%o6) 2
(%i7) simp : true;
(%o7) true
(%i8) sum (1/k^2, k, 1, inf);
    inf
   > --
   / 2
   ===== k
   k = 1
(%o8)
(%i9) sum (1/k^2, k, 1, inf), simpsum;
   2
%pi
----
6
(%o9)
(%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 \( \text{ev}(\text{expr}, F) \) and \( \text{expr}, F \) (at the interactive prompt) are equivalent to \( F(\text{ev(expr)}) \).

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:

- `bfloat`,
- `factor`,
- `fullratsimp`,
- `logcontract`,
- `polarform`,
- `radcan`,
- `ratexpand`,
- `ratsimp`,
- `rectform`,
- `rootscontract`,
- `trigexpand`, and
- `trigreduce`.

Examples:

(%i1) x^3 - 1;
(%o1) x - 1
(%i2) x^3 - 1, factor;
(%o2) (x - 1) (x + x + 1)
(%i3) factor (x^3 - 1);
(%o3) (x - 1) (x + x + 1)
(%i4) cos(4 * x) / sin(x)^4;
\[
\frac{\cos(4 \ x)}{4} \quad \frac{4}{\sin(x)}
\]

(%i5) \(\cos(4 \ x) / \sin(x)^4\), trigexpand;

\[
\frac{\sin^2(x) - 6 \cos(x) \sin(x) + \cos(x)}{4 \sin(x)^4}
\]

(%o5) \(\cos(4 \ x) / \sin(x)^4\), trigexpand, ratexpand;

\[
\frac{6 \cos(x) \cos(x)}{2 \sin(x)} + \frac{6 \cos(x) \cos(x)}{2 \sin(x)} + 1
\]

(%o6) \(\cos(4 \ x) / \sin(x)^4\), trigexpand, ratexpand;

\[
\frac{6 \cos(x) \cos(x)}{2 \sin(x)} + \frac{6 \cos(x) \cos(x)}{2 \sin(x)} + 1
\]

(%o7) ratexpand (trigexpand (\cos(4 \ x) / \sin(x)^4));

Option variable
\texttt{infeval}

Enables "infinite evaluation" mode. \texttt{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 \texttt{ev}. Of course expressions such as \texttt{ev (X, X=X+1, infeval)} will generate an infinite loop.
kill \((a_1, ..., a_n)\)  
Function

kill \((\text{labels})\)  
Function

kill \((\text{inlabels, outlabels, linelabels})\)  
Function

kill \((n)\)  
Function

kill \([m, n]\)  
Function

kill \((\text{values, functions, arrays, ...})\)  
Function

kill \((\text{all})\)  
Function

kill \((\text{allbut } (a_1, ..., a_n))\)  
Function

Removes all bindings (value, function, array, or rule) from the arguments \(a_1, ..., a_n\). 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., \(\text{kill } (\text{inlabels, functions, allbut } (\text{foo, bar}))\).

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

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 \((\text{infolist})\), where \(\text{infolist}\) is any item in \(\text{infolists}\) (such as \(\text{values, functions, arrays}\)) unbinds all items in \(\text{infolist}\). See also \(\text{infolists}\).

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

kill \((\text{allbut } (a_1, ..., a_n))\) unbinds all items on all infolists except for \(a_1, ..., a_n\). kill \((\text{allbut } \text{infolist}))\) unbinds all items except for the ones on \(\text{infolist}\), where \(\text{infolist}\) is \(\text{values, functions, arrays}\), 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 \((\text{symbol})\) unbinds all properties of \(\text{symbol}\). In contrast, the functions \(\text{remvalue, remfunction, remarray, remrule}\) unbind a specific property.

kill always returns \(\text{done}\), even if an argument has no binding.

\textbf{labels} \((\text{symbol})\)  
Function

\textbf{labels}  
System variable

Returns the list of input, output, or intermediate expression labels which begin with \(\text{symbol}\). Typically \(\text{symbol}\) is the value of \(\text{inchar, outchar, or linechar}\). The label character may be given with or without a percent sign, so, for example, \(i\) and \(\%i\) yield the same result.

If no labels begin with \(\text{symbol}\), \(\text{labels}\) returns an empty list.
The function labels quotes its argument. The quote-quote operator '' defeats quotation. For example, labels (''inchar) returns the input labels which begin with the current input label character.

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

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 flag 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.

**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.
function playback() Displays input, output, and intermediate expressions, without recomputing them.

Function playback(n) 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.

Function playback([m, n]) 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.

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

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

Function playback(output) playback(input) displays all input expressions generated so far.

Function playback(slow) 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.

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

Function playback(grind) 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).

function printprops(a, 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.

function printprops([a_1, ..., a_n], i) function printprops(all, i)
prompt is the prompt symbol of the `demo` function, `playback (slow)` mode, and the Maxima break loop (as invoked by `break`).

**Function**

**quit ()**

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 `debugging`.

**Function**

**remfunction (f_1, ..., f_n)**

**Function**

**remfunction (all)**

Unbinds the function definitions of the symbols `f_1, ..., 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 array functions or subscripted functions. `remarray` applies to those types of functions.

**Function**

**reset ()**

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.

**Option variable**

**showtime**

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`.

**Function**

**to_lisp ()**

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.
Chapter 4: Command Line

(%i1) f(x):=x^2+x;

(%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

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) := x^2]

(%i2) values;
(%o2) [a, b, c]

(%i3) [kill(a), remove(b,value), remvalue(c)];
(%o3) [done, done, [c]]

(%i4) values;
(%o4) []
5 Operators

5.1 nary

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 Syntax.

5.2 nofix

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 Syntax.

5.3 postfix

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 Syntax.

5.4 prefix

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 Syntax.

5.5 Arithmetic operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator</th>
<th>Operator</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>-</td>
<td>*</td>
<td>/</td>
</tr>
<tr>
<td>^</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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, b, x, 2, a, x, 3]);
(%o5) 3 x + 2 a + 19
```
Division and exponentiation are binary, noncommutative operators. The names of these operators are "/" and ".".

Subtraction and division are represented internally in terms of addition and multiplication, respectively.

Computations are carried out on literal numbers. Floating-point contagion applies.

Arithmetic computations are a simplification, not an evaluation.

Arithmetic is carried out element-by-element for lists (depending on listarithmetic) and matrices.
5.6 Relational operators
Chapter 5: Operators

The symbols \(<\), \(\leq\), \(\geq\), and \(>\) represent less than, less than or equal, greater than or equal, and greater than, respectively. The names of these operators are "\(<\)" "\(\leq\)" "\(\geq\)" 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 \(\text{is}\) and \(\text{maybe}\), and the programming constructs \(\text{if}\), \(\text{while}\), and \(\text{unless}\). 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 \text{true} or \text{false}, the behavior of \(\text{is}\) and \(\text{if}\) are governed by the global flag \text{prederror}. When \text{prederror} is \text{true}, \text{is} and \text{if} trigger an error. When \text{prederror} is \text{false}, \text{is} returns \text{unknown}, and \text{if} returns a partially-evaluated conditional expression.

\text{maybe} always behaves as if \text{prederror} were \text{false}, and \text{while} and \text{unless} always behave as if \text{prederror} were \text{true}.

Relational operators do not distribute over lists or other aggregates.

See also \(\text{=}\ #\) \text{equal} and \text{notequal}.

Examples:

Relational expressions are evaluated to Boolean values by some functions and programming constructs.

\begin{verbatim}
(%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
\end{verbatim}

Relational expressions are not otherwise evaluated or simplified to Boolean values, although the arguments of relational expressions are evaluated.

\begin{verbatim}
(%i1) [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]
\end{verbatim}
5.7 General operators

^^  Operator
Noncommutative exponentiation operator. ^^ is the exponentiation operator corresponding to noncommutative multiplication ., just as the ordinary exponentiation operator ^ 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;
3 2 <2> <3>
(%o1) a b + a . b
(\%i2) string (a . a . b . b . b + a * a * a * b * b);
(%o2) a^3*b^2+a^^2 . b^^3

!  Operator
The factorial operator.
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. See factlim, gammalim, minfactorial and factcomb.
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!];
Chapter 5: Operators

105 \sqrt{\pi}

\begin{align*}
(\%o2) & = 1, \text{-----------------, } 40320, 20! \\
(\%i3) & = 4.77!, (1.0+\%i)!; \\
(\%o3) & = 81.44668037931197, \\
& \quad 3.430658398165454 \times 10^{-15} + 6.529654964201665 \times 10^{15} \\
(\%i4) & = 2.86b0!, (1.0b0+\%i)!
\end{align*}

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.

\begin{align*}
(\%i1) & = [(\%i + 1)!, \%pi!, \%e!, (\cos(1) + \sin(1))!]; \\
(\%o1) & = \left[ (\%i + 1)! , \%pi! , \%e! , (\sin(1) + \cos(1))! \right] \\
(\%i2) & = \text{ev}(\%i1, \%num, \%enum); \\
(\%o2) & = \left[ 7.188082728976031, \right. \\
& \quad 4.260820476357003, 1.227580202486819 \\
\end{align*}

Factorials are simplified, not evaluated. Thus \(x!\) may be replaced even in a quoted expression.

\begin{align*}
(\%i1) & = '([0!, (7/2)!, 4.77!, 8!, 20!]); \\
(\%o1) & = \left[ 1, \text{-----------------, } 81.44668037931199, 40320, 16 \right] \\
& \quad 2432902008176640000
\end{align*}

Maxima knows the derivative of the factorial function.

\begin{align*}
(\%i1) & = \text{diff}(x!, x); \\
(\%o1) & = x! \psi(x + 1)
\end{align*}

The option variable \texttt{factorial\_expand} controls expansion and simplification of expressions with the factorial function.

\begin{align*}
(\%i1) & = (n+1)!/n!, \texttt{factorial\_expand}:\text{true}; \\
(\%o1) & = n + 1
\end{align*}

\textbf{!!}

The double factorial operator.

For an integer, float, or rational number \(n, n!!\) evaluates to the product \(n(n-2)(n-4)\ldots(n-2(k-1))\) where \(k\) is equal to \texttt{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 \texttt{genfact}(n, n/2, 2).

\textbf{#}

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;
   a = b
(%o1)

(%i2) is (a = b);
   false
(%o2)

(%i3) a # b;
   a # b
(%o3)

(%i4) not a = b;
   true
(%o4)

(%i5) is (a # b);
   true
(%o5)

(%i6) is (not a = b);
   true
(%o6)
```

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`, `dot0scsimp`, `dot0simp`, `dot1simp`, `dotassoc`, `dotconstrules`, `dotdistrib`, `dotexptsimp`, `dotident`, and `dottcrules`.

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 nonexistent elements.

When the left-hand side is a subscripted element of an undeclared Maxima 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;
   a
(%o1)

(%i2) a : 123;
```
Assignment to an element of a list.

\[
\begin{align*}
\%i1 & : [1, 2, 3]; \\
\%o1 & : [1, 2, 3] \\
\%i2 & : b[3] : 456; \\
\%o2 & : 456 \\
\%i3 & : b; \\
\%o3 & : [1, 2, 456]
\end{align*}
\]

Assignment creates an undeclared array.

\[
\begin{align*}
\%i1 & : c[99] : 789; \\
\%o1 & : 789 \\
\%i2 & : c[99]; \\
\%o2 & : 789 \\
\%i3 & : c; \\
\%o3 & : c \\
\%i4 & : \text{arrayinfo}(c); \\
\%o4 & : \text{hashed, 1, [99]} \\
\%i5 & : \text{listarray}(c); \\
\%o5 & : [789]
\end{align*}
\]

Multiple assignment.

\[
\begin{align*}
\%i1 & : [a, b, c] : [45, 67, 89]; \\
\%o1 & : [45, 67, 89] \\
\%i2 & : a; \\
\%o2 & : 45 \\
\%i3 & : b; \\
\%o3 & : 67 \\
\%i4 & : c; \\
\%o4 & : 89 \\
\end{align*}
\]

Multiple assignment is carried out in parallel. The values of \(a\) and \(b\) are exchanged in this example.

\[
\begin{align*}
\%i1 & : [a, b] : [33, 55]; \\
\%o1 & : [33, 55] \\
\%i2 & : [a, b] : [b, a]; \\
\%o2 & : [55, 33] \\
\%i3 & : a; \\
\%o3 & : 55 \\
\%i4 & : b; \\
\%o4 & : 33 \\
\end{align*}
\]

:: Operator

\[
\begin{align*}
\text{::} \text{ is the same as : (which see) except that :: evaluates its left-hand side as well as its right-hand side.}
\end{align*}
\]

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).

(\%i11) x: \%pi$  
(\%i12) y: 1234$  
(\%i13) z: 1729 * w$  
(\%i14) printq1 (x) ::= block (print ("(1) x is equal to", x),  
'(print ("(2) x is equal to", x)))$  
(\%i15) printq1 (y - z);  
(1) x is equal to y - z  
(2) x is equal to \%pi  
(\%o5)  \%pi
Chapter 5: Operators

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 `''%.

```lisp
(%i1) x: %pi$
(%i2) y: 1234$
(%i3) z: 1729 * w$
(%i4) print1 (x) := block (print ("(1) x is equal to", x),

'(print ("(2) x is equal to", x)))$
(%i5) print1 (y - z);
(1) x is equal to 1234 - 1729 w
(%o5) print((2) x is equal to, x)
(%o6) print((2) x is equal to, x)

macroexpand returns a macro expansion. macroexpand (foo (x)) followed by `''% is equivalent to foo (x) when foo is a macro function.

```lisp
(%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));
(1) x is equal to 1234 - 1729 w
(%o5) print(x is equal to, y - z)
(%o6) print(x is equal to, y - z)
(%i7) g (y - z);
(2) x is equal to %pi

```

Operator `:=`  

The function definition operator. $f(x_1, \ldots, x_n) := expr$ defines a function named $f$ with arguments $x_1, \ldots, x_n$ and function body $expr$. $:= never evaluates the function body (unless explicitly evaluated by quote-quote `'''). The function so defined may be an ordinary Maxima function (with arguments enclosed in parentheses) or an array 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 $:= accepts a variable number of arguments. Actual arguments are assigned one-to-one to formal arguments $x_1, \ldots, x_{(n - 1)}$, 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)

The function defined by ::= may be an ordinary Maxima function or an array function.

(%i1) G1 (x, y) ::= x . y - y . x;
(%o1) G1(x, y) ::= x . y - y . x
(%i2) G2 [x, y] ::= x . y - y . x;
(%o2) G2 ::= x . y - y . x

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

= 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
Chapter 5: Operators

expressions, for which \texttt{is(equal(a, b))} is \texttt{true}. Expressions can be equivalent and not syntactically equal.

The negation of \texttt{=} is represented by \texttt{#}. As with \texttt{=}, an expression \texttt{a \# b}, by itself, is not evaluated. \texttt{is(a \# b)} evaluates \texttt{a \# b} to \texttt{true} or \texttt{false}.

In addition to \texttt{is}, some other operators evaluate \texttt{=} and \texttt{#} to \texttt{true} or \texttt{false}, namely \texttt{if}, \texttt{and}, \texttt{or}, and \texttt{not}.

Note that because of the rules for evaluation of predicate expressions (in particular because \texttt{not expr} causes evaluation of \texttt{expr}), \texttt{not a = b} is equivalent to \texttt{is(a \# b)}, instead of \texttt{a \# b}.

\texttt{rhs} and \texttt{lhs} return the right-hand and left-hand sides, respectively, of an equation or inequation.

See also \texttt{equal} and \texttt{notequal}.

Examples:

An expression \texttt{a = b}, by itself, represents an unevaluated equation, which might or might not hold.

\begin{verbatim}
(\%i1) eq_1 : a * x - 5 * y = 17;
(\%o1) a x - 5 y = 17
(\%i2) eq_2 : b * x + 3 * y = 29;
(\%o2) 3 y + b x = 29
(\%i3) solve ([eq_1, eq_2], [x, y]);
196 29 a - 17 b
[[x = --------, y = 196 5 b + 3 a]
 5 b + 3 a]

(\%i4) subst (%), [eq_1, eq_2]);
196 a 3 (29 a - 17 b)
------- + --------------- = 17,
5 b + 3 a 5 b + 3 a
196 b 3 (29 a - 17 b)
------- + --------------- = 29]
5 b + 3 a 5 b + 3 a

(\%i5) ratsimp (%);
(\%o5) [17 = 17, 29 = 29]
\end{verbatim}

\texttt{is(a = b)} evaluates \texttt{a = b} to \texttt{true} when \texttt{a} and \texttt{b} are syntactically equal (that is, identical). Expressions can be equivalent and not syntactically equal.

\begin{verbatim}
(\%i1) a : (x + 1) * (x - 1);
(\%o1) (x - 1) (x + 1)
(\%i2) b : x^2 - 1;
(\%o2) x - 1
(\%i3) [is (a = b), is (a \# b)];
(\%o3) [false, true]
(\%i4) [is (equal (a, b)), is (notequal (a, b))];
(\%o4) [true, false]
\end{verbatim}

Some operators evaluate \texttt{=} and \texttt{#} to \texttt{true} or \texttt{false}.

\begin{verbatim}
(\%i11) if expand ((x + y)^2) = x^2 + 2 * x * y + y^2 then FOO else BAR;
\end{verbatim}
(%o1) FOO
(%i2) eq_3 : 2 * x = 3 * x;
(%o2) 2 x = 3 x
(%i3) eq_4 : exp (2) = %e^2;
    2  
(%o3) %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

and

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.

or

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.
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 preerror governs the behavior of not when its operand cannot be determined to be true or false. not prints an error message when preerror is true. Otherwise, operands which do not evaluate to true or false are accepted, and the result is a Boolean expression.

abs (expr)  Returns the absolute value expr. If expr is complex, returns the complex modulus of expr.

abs distributes over a list, a matrix, or an equation. See distribute_over.

additive  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).

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

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

while

```lisp
(%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.

```lisp
(%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(a_1, a_2, ...)) has the effect of kill(all) except that it does not kill the symbols a_1, a_2, ....
Declaration

antisymmetric

If \texttt{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 \texttt{symmetric} or \texttt{commutative}, where \( n \) is the number of interchanges of two arguments necessary to convert it to that form.

Function

cabs (expr)

Returns the complex absolute value (the complex modulus) of \( expr \).

Function

ceiling (x)

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), \texttt{ceiling} evaluates \( x \) using big floating point numbers, and applies \texttt{ceiling} to the resulting big float. Because \texttt{ceiling} uses floating point evaluation, it’s possible, although unlikely, that \texttt{ceiling} could return an erroneous value for constant inputs. To guard against errors, the floating point evaluation is done using three values for \texttt{fpprec}.

For non-constant inputs, \texttt{ceiling} tries to return a simplified value. Here are examples of the simplifications that \texttt{ceiling} knows about:

\begin{verbatim}
(%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(n, 6)]
(%i5) assume (x > 0, x < 1)$
(%i6) ceiling (x);
(%o6) 1
(%i7) tex (ceiling (a));
$$\left \lceil a \right \rceil$$
(%o7) false
\end{verbatim}

The function \texttt{ceiling} does not automatically map over lists or matrices. Finally, for all inputs that are manifestly complex, \texttt{ceiling} returns a noun form.

If the range of a function is a subset of the integers, it can be declared to be \texttt{integervalued}. Both the \texttt{ceiling} and \texttt{floor} functions can use this information; for example:

\begin{verbatim}
(%i1) declare (f, integervalued)$
(%i2) floor (f(x));
(%o2) f(x)
(%i3) ceiling (f(x) - 1);
(%o3) f(x) - 1
\end{verbatim}

Function

charfun (p)

Return 0 when the predicate \( p \) evaluates to \texttt{false}; return 1 when the predicate evaluates to \texttt{true}. When the predicate evaluates to something other than \texttt{true} or \texttt{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]

**Declaration**

**commutative**

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`.

**Function**

**compare**(x, y)

Return a comparison operator `op` (`<`, `<=`, `>`, `>=`, `=`, or `#`) such that `is(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:

```lisp
(%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

```lisp
(%i1) compare (acos (x^2 + 1), acos (x^2 + 1) + 1);
%o1) <
```

The real domain of `acos (x^2 + 1)` is empty.

**Function**

**entier**(x)

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.

**Function**

**equal**(a, b)

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
Two expressions may be equivalent even if they are not syntactically equal (i.e., identical).

When \texttt{is} fails to reduce \texttt{equal} to \texttt{true} or \texttt{false}, the result is governed by the global flag \texttt{prederror}. When \texttt{prederror} is \texttt{true}, \texttt{is} complains with an error message. Otherwise, \texttt{is} returns \texttt{unknown}.

In addition to \texttt{is}, some other operators evaluate \texttt{equal} and \texttt{notequal} to \texttt{true} or \texttt{false}, namely \texttt{if}, \texttt{and}, \texttt{or}, and \texttt{not}.

The negation of \texttt{equal} is \texttt{notequal}.

Examples:

By itself, \texttt{equal} does not evaluate or simplify.

\begin{verbatim}
(%i1) equal (x^2 - 1, (x + 1) * (x - 1));
  2
(%o1) equal(x - 1, (x - 1) (x + 1))
(%i2) equal (x, x + 1);
(%o2) equal(x, x + 1)
(%i3) equal (x, y);
(%o3) equal(x, y)
\end{verbatim}

The function \texttt{is} attempts to evaluate \texttt{equal} to a Boolean value. \texttt{is(equal(a, b))} returns \texttt{true} when \texttt{ratsimp(a - b)} returns 0. Two expressions may be equivalent even if they are not syntactically equal (i.e., identical).

\begin{verbatim}
(%i1) ratsimp (x^2 - 1 - (x + 1) * (x - 1));
  2
(%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
\end{verbatim}

When \texttt{is} fails to reduce \texttt{equal} to \texttt{true} or \texttt{false}, the result is governed by the global flag \texttt{prederror}.

\begin{verbatim}
(%i1) [aa : x^2 + 2*x + 1, bb : x^2 - 2*x - 1];
  2
(%o1) [x + 2 x + 1, x - 2 x - 1]
(%i2) ratsimp (aa - bb);
(%o2) 4 x + 2
(%i3) prederror : true;
\end{verbatim}
Maxima was unable to evaluate the predicate:
\[ \text{equal}(x + 2x + 1, x - 2x - 1) \]
-- an error. Quitting. To debug this try debugmode(true);

Some operators evaluate equal and notequal to true or false.

Because \text{not expr} causes evaluation of expr, \text{not equal(a, b)} is equivalent to \text{is(notequal(a, b))}.

\text{floor (x)}

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), \text{floor} evaluates x using big floating point numbers, and applies \text{floor} to the resulting big float. Because \text{floor} uses floating point evaluation, it’s possible, although unlikely, that \text{floor} could return an erroneous value for constant inputs. To guard against errors, the floating point evaluation is done using three values for \text{fpprec}.

For non-constant inputs, \text{floor} tries to return a simplified value. Here are examples of the simplifications that \text{floor} knows about:

\[
\begin{align*}
(\%i1) & \quad \text{floor (ceiling (x));} \\
(\%o1) & \quad \text{ceiling(x)} \\
(\%i2) & \quad \text{floor (floor (x));} \\
(\%o2) & \quad \text{floor(x)} \\
(\%i3) & \quad \text{declare (n, integer)$} \\
(\%i4) & \quad [\text{floor (n), floor (abs (n)), floor (min (n, 6))}]; \\
(\%o4) & \quad [n, \text{abs(n), min(n, 6)}] \\
(\%i5) & \quad \text{assume (x > 0, x < 1)$} \\
(\%i6) & \quad \text{floor (x)}; \\
(\%o6) & \quad 0 \\
(\%i7) & \quad \text{tex (floor (a));}
\end{align*}
\]
The function \texttt{floor} does not automatically map over lists or matrices. Finally, for all inputs that are manifestly complex, \texttt{floor} returns a noun form. If the range of a function is a subset of the integers, it can be declared to be \texttt{integervalued}. Both the \texttt{ceiling} and \texttt{floor} functions can use this information; for example:

\begin{verbatim}
(%i1) declare (f, integervalued)$
(%i2) floor (f(x));
  f(x)
(%o2)
(%i3) ceiling (f(x) - 1);
  f(x) - 1
\end{verbatim}

\texttt{notequal} \((a, b)\)

Represents the negation of \texttt{equal}(a, b).

Examples:

\begin{verbatim}
(%i1) equal (a, b);
  equal(a, b)
(%o1)
(%i2) maybe (equal (a, b));
  unknown
(%o2)
(%i3) notequal (a, b);
  notequal(a, b)
(%o3)
(%i4) not equal (a, b);
  notequal(a, b)
(%o4)
(%i5) maybe (notequal (a, b));
  unknown
(%o5)
(%i6) assume (a > b);
  [a > b]
(%o6)
(%i7) equal (a, b);
  equal(a, b)
(%o7)
(%i8) maybe (equal (a, b));
  false
(%o8)
(%i9) notequal (a, b);
  notequal(a, b)
(%o9)
(%i10) maybe (notequal (a, b));
  true
(%o10)
\end{verbatim}

\texttt{evenp} \((expr)\)

Returns \texttt{true} if \texttt{expr} is an even integer. \texttt{false} is returned in all other cases.

\texttt{fix} \((x)\)

A synonym for \texttt{entier} \((x)\).

\texttt{fullmap} \((f, expr_1, \ldots)\)

Similar to \texttt{map}, but \texttt{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)
```

**fullmapl** $(f, list_1, ...)$  
Similar to **fullmap**, but **fullmapl** only maps onto lists and matrices.

Example:

```
(%i1) fullmapl ("+", [3, [4, 5]], [[a, 1], [0, -1.5]]);
(%o1) [[a + 3, 4], [4, 3.5]]
```

**is** $(expr)$  
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

maybe (expr)
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

isqrt (x)
Returns the "integer square root" of the absolute value of x, which is an integer.

lmax (L)
When L is a list or a set, return apply ('max, args (L)). When L isn’t a list or a set, signal an error.

lmin (L)
When L is a list or a set, return apply ('min, args (L)). When L isn’t a list or a set, signal an error.

max (x_1, ..., x_n)
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).
Function \texttt{min} \((x_1, \ldots, x_n)\)

Return a simplified value for the minimum of the expressions \(x_1\) through \(x_n\). When \texttt{get (trylevel, maxmin)} is 2 or greater, \texttt{min} uses the simplification \(\text{min} (e, -e) \rightarrow -|e|\). When \texttt{get (trylevel, maxmin)} is 3 or greater, \texttt{min} tries to eliminate expressions that are between two other arguments; for example, \(\text{min} (x, 2x, 3x) \rightarrow \text{min} (x, 3x)\). To set the value of \texttt{trylevel} to 2, use \texttt{put (trylevel, 2, maxmin)}.

Function \texttt{polymod} \((p)\)

Function \texttt{polymod} \((p, m)\)

Converts the polynomial \(p\) to a modular representation with respect to the current modulus which is the value of the variable \texttt{modulus}.

\texttt{polymod} \((p, m)\) specifies a modulus \(m\) to be used instead of the current value of \texttt{modulus}.

See \texttt{modulus}.

Function \texttt{mod} \((x, y)\)

If \(x\) and \(y\) are real numbers and \(y\) is nonzero, return \(x - y \times \text{floor}(x / y)\). Further for all real \(x\), we have \(\text{mod} (x, 0) = x\). For a discussion of the definition \(\text{mod} (x, 0) = x\), see Section 3.4, of "Concrete Mathematics," by Graham, Knuth, and Patashnik. The function \(\text{mod} (x, 1)\) is a sawtooth function with period 1 with \(\text{mod} (1, 1) = 0\) and \(\text{mod} (0, 1) = 0\).

To find the principal argument (a number in the interval \((-\pi, \pi]\)) of a complex number, use the function \(x \rightarrow \pi - \text{mod} (\pi - x, 2\pi)\), where \(x\) is an argument.

When \(x\) and \(y\) are constant expressions (10 * \(\pi\), for example), \texttt{mod} uses the same big float evaluation scheme that \texttt{floor} and \texttt{ceiling} uses. Again, it’s possible, although unlikely, that \texttt{mod} could return an erroneous value in such cases.

For nonnumerical arguments \(x\) or \(y\), \texttt{mod} knows several simplification rules:

- \((%i1) \text{ mod} (x, 0); \)\n  \((%o1) x\)\n- \((%i2) \text{ mod} (a*x, a*y); \)\n  \((%o2) a \text{ mod}(x, y)\)\n- \((%i3) \text{ mod} (0, x); \)\n  \((%o3) 0\)

Function \texttt{oddp} \((expr)\)

is \texttt{true} if \(expr\) is an odd integer. \texttt{false} is returned in all other cases.

Function \texttt{psubst} \((list, expr)\)

Function \texttt{psubst} \((a, b, expr)\)

\texttt{psubst}(a, b, expr) is similar to \texttt{subst}. See \texttt{subst}.

In distinction from \texttt{subst} the function \texttt{psubst} makes parallel substitutions, if the first argument \(list\) is a list of equations.

See also \texttt{sublis} for making parallel substitutions.

Example:

The first example shows parallel substitution with \texttt{psubst}. The second example shows the result for the function \texttt{subst}, which does a serial substitution.
56 Maxima 5.24.0 Manual

(%i4) psubst ([a^2=b, b=a], sin(a^2) + sin(b));
(%o4) sin(b) + sin(a)
(%i5) subst ([a^2=b, b=a], sin(a^2) + sin(b));
(%o5) 2 sin(a)

make_random_state (n) Function
make_random_state (s) Function
make_random_state (true) Function
make_random_state (false) Function

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
Covers 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) .2310127244107132
Chapter 5: Operators

(%i8) random (10.0);
(%o8) 4.394553645870825
(%i9) random (100.0);
(%o9) 32.28666704056853
(%i10) set_random_state (s2);
(%o10) done
(%i11) random (1.0);
(%o11) .2310127244107132
(%i12) random (10.0);
(%o12) 4.394553645870825
(%i13) random (100.0);
(%o13) 32.28666704056853

Function

rationalize (expr)

Convert all double floats and big floats in the Maxima expression expr to their exact rational equivalents. If you are not familiar with the binary representation of floating point numbers, you might be surprised that rationalize (0.1) does not equal 1/10. This behavior isn’t special to Maxima – the number 1/10 has a repeating, not a terminating, binary representation.

(%i1) rationalize (0.5);
(%o1) 1
    -
    2
(%i2) rationalize (0.1);
(%o2) --
    10
(%i3) fpprec : 5$
(%i4) rationalize (0.1b0);
(%o4) 209715
    -------
    2097152
(%i5) fpprec : 20$
(%i6) rationalize (0.1b0);
(%o6) 236118324143482260685
    ------------------------
    2361183241434822606848
(%i7) rationalize (sin (0.1*x + 5.6));
(%o7)
   x
   28
  sin(-- + --)
   10  5
(%i1) unitfrac(r) := block([uf : [], q],
   if not(ratnump(r)) then
     error("The input to 'unitfrac' must be a rational number"),
   while r # 0 do (
     uf : cons(q : 1/ceiling(1/r), uf),
     r : r - q),
   reverse(uf))$

(%i2) unitfrac (9/10);

1 1 1
[-, -, --]
2 3 15

(%o2) 9

(%i3) apply ("+", %);

9

(%o3) --
10

(%i4) unitfrac (-9/10);

1
- 1, --
10

(%o4) 9

(%i5) apply ("+", %);

9

(%o5) --
10

(%i6) unitfrac (36/37);

1 1 1 1 1
[-, -, -, --, ----]
2 3 8 69 6808

(%o6) 36

(%i7) apply ("+", %);

37

round (x)

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.

sign (expr)

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).

signum (x)

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 abs(x)/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, \texttt{signum} a simplified but equivalent form. The simplifications include reflection (\texttt{signum(-x)} gives \texttt{-signum(x)}) and multiplicative identity (\texttt{signum(x*y)} gives \texttt{signum(x) * signum(y)}).

The \texttt{signum} function distributes over a list, a matrix, or an equation. See \texttt{distribute_over}.

\begin{verbatim}
function
sort (L, P)  Function
sort (L)  Function

dsort a list \texttt{L} according to a predicate \texttt{P} of two arguments, such that \texttt{P (L[k], L[k + 1])} is \texttt{true} for any two successive elements. The predicate may be specified as the name of a function or binary infix operator, or as a \texttt{lambda} expression. If specified as the name of an operator, the name is enclosed in "double quotes".

The sorted list is returned as a new object; the argument \texttt{L} is not modified. To construct the return value, \texttt{sort} makes a shallow copy of the elements of \texttt{L}.

If the predicate \texttt{P} is not a total order on the elements of \texttt{L}, then \texttt{sort} might run to completion without error, but the result is undefined. \texttt{sort} complains if the predicate evaluates to something other than \texttt{true} or \texttt{false}.

\texttt{sort (L)} is equivalent to \texttt{sort (L, orderlessp)}. That is, the default sorting order is ascending, as determined by \texttt{orderlessp}. All Maxima atoms and expressions are comparable under \texttt{orderlessp}.

The predicate \texttt{ordergreatp} sorts a list in descending order. The predicate \texttt{ordermagnitudep} sorts Maxima numbers, constant symbols with a numerical value, or expressions which can be evaluated to a constant by magnitude. All other elements of the list \texttt{L} are sorted by \texttt{orderlessp}. The predicate "\texttt{<}" allows the ordering by magnitude too, but does not order completely if elements in the list \texttt{L} are not comparable under "\texttt{<}".

Examples:
\begin{verbatim}
(%i1) sort ([11, -17, 29b0, 7.55, 3, -5/2, b + a, 9 * c, 19 - 3 * x]);
5
(%o1) [- 17, - -, 3, 7.55, 11, 2.9b1, b + a, 9 c, 19 - 3 x]
2
(%i2) sort ([11, -17, 29b0, 7.55, 3, -5/2, b + a, 9*c, 19 - 3*x],
    ordergreatp);
5
(%o2) [19 - 3 x, 9 c, b + a, 2.9b1, 11, 7.55, 3, - -, - 17]
2
(%i3) sort ([%pi, 3, 4, %e, %gamma]);
(%o3) [3, 4, %e, %gamma, %pi]
(%i4) sort ([%pi, 3, 4, %e, %gamma], "<");
(%o4) [%gamma, %e, 3, %pi, 4]
(%i5) my_list: [[aa,hh,uu], [ee,cc], [zz,xx,mm,cc], [%pi,%e]];
(%o5) [[aa, hh, uu], [ee, cc], [zz, xx, mm, cc], [%pi, %e]]
(%i6) sort (my_list);
(%o6) [[%pi, %e], [aa, hh, uu], [ee, cc], [zz, xx, mm, cc]]
(%i7) sort (my_list, lambda ([a, b], orderlessp (reverse (a),
reverse (b))));
\end{verbatim}
\end{verbatim}
Order Maxima numbers, constants, and constants expressions by magnitude, and all other elements in ascending sorting order:

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

\[(%o8)\] [minf, 0, sin(1), 1, 1.0, 1.0b0, 2, %e, 3, inf, %i, %i + 1, 2 x]

**sqrt** (x) Function

The square root of \(x\). It is represented internally by \(x^{1/2}\). See also rootscontract.

radexp will cause \(n\)th roots of factors of a product which are powers of \(n\) to be pulled outside of the radical, e.g. \(\sqrt{16*x^2}\) will become \(4*x\) only if radexp is true.

**sqrtdispflag** Option variable

Default value: true

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

**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:

\[(%i11)\] sublis ([a=b, b=a], sin(a) + cos(b));

\%(o11)\] sin(b) + cos(a)

**sublist** (list, p) Function

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

Example:

\%(i11) L: [1, 2, 3, 4, 5, 6];
\%(o11) \] [1, 2, 3, 4, 5, 6]
\%(i12) sublist (L, evenp);
\%(o12) \] [2, 4, 6]

**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.
Chapter 5: Operators

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.

\[
\text{(6i1) subst}(100, g, g[x] + 2);
\]

\[
\text{subst: cannot substitute 100 for operator g in expression g}\quad x
\]

-- an error. To debug this try: debugmode(true);

\[
\text{(6i2) subst}(100, g, g[x] + 2), \text{subnumsimp: true};
\]

\[
\text{(6o2)}
\]

The substitution is done in serial for a list of equations. Compare this with a parallel substitution:

\[
\text{(6i3) subst}([a=\text{b}, \text{b}=\text{c}], a+\text{b});
\]

\[
\text{(6o3)}
\]

\[
\text{(6i4) sublis}([a=\text{b}, \text{b}=\text{c}], a+\text{b});
\]
For further examples, do example (subst).

**substinpart** \((x, \text{expr}, n_1, \ldots, n_k)\)

Function

Similar to substpart, but substinpart works on the internal representation of expr.

Examples:

\[
\text{(\%i1) } x \cdot \text{diff}(f(x), x, 2); \\
\text{(\%o1)} \quad x \cdot \frac{d}{dx} \left(\frac{\text{(\%o1) } x \cdot \text{diff}(f(x), x, 2)}}{2}ight) \\
\]

\[
\text{(\%i2) } \text{substinpart}(d^2, \text{(\%o1)}, 2); \\
\text{(\%o2)} \quad x \cdot d \\
\]

\[
\text{(\%i3) } \text{substinpart}(f1, f[f1](x + 1), 0); \\
\text{(\%o3)} \quad f1(x + 1) \\
\]

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

\[
\text{(\%i1) } \text{part}(x + y + z, [1, 3]); \\
\text{(\%o1)} \quad z + x \\
\]

**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.

\[
\text{(\%i1) } \text{expr: } 27y^3 + 54x^2y + 36xy^2 + y + 8x^3 + x + 1; \\
\text{(\%o1)} \quad 27y^3 + 54x^2y + 36xy^2 + y + 8x^3 + x + 1 \\
\]

\[
\text{(\%i2) } \text{part}(\text{expr}, 2, [1, 3]); \\
\text{(\%o2)} \quad 54y \\
\]

\[
\text{(\%i3) } \text{sqrt} (\text{piece}/54); \\
\text{(\%o3)} \quad \text{abs}(y) \\
\text{(\%i4) } \text{substinpart} (\text{factor}(\text{piece}), \text{expr}, [1, 2, 3, 5])); \\
\text{(\%o4)} \quad (3y + 2x) + y + x + 1 \\
\]

\[
\text{(\%i5) } \text{expr: } 1/x + y/x - 1/z; \\
\text{(\%o5)} \quad 1/y \quad 1 \\
\text{ - - + - + - } \\
\text{z \quad x \quad x} \\
\]

\[
\text{(\%i6) } \text{substinpart} (\text{xthru}(\text{piece}), \text{expr}, [2, 3]); \\
\text{(\%o6)} \quad y + 1 \quad 1 \\
\text{ ----- - - } \\
\text{ x \quad z} \\
\]

Also, setting the option inflag to true and calling part or substpart is the same as calling inpart or substinpart.
Chapter 5: Operators

**Function** substpart \((x, \text{expr}, n_1, \ldots, n_k)\) Substitutes \(x\) for the subexpression picked out by the rest of the arguments as in \text{part}. It returns the new value of \text{expr}. \(x\) may be some operator to be substituted for an operator of \text{expr}. In some cases \(x\) needs to be enclosed in double-quotes " (e.g. substpart \("+", a*b, 0\) yields \(b + a\)).

\[(%i1) \quad 1/(x^2 + 2); \quad \frac{1}{2} \quad \frac{2}{x + 2} \]
\%(o1) \quad \frac{1}{x^2 + 2} \]

\[(%i2) \quad \text{substpart} \left(\frac{3}{2}, \%, 2, 1, 2\right); \quad \frac{1}{2} \quad \frac{3}{2} \quad \frac{1}{x + 2} \]
\%(o2) \quad \text{substpart} \left(\frac{3}{2}, \%, 2, 1, 2\right) \quad \frac{1}{x + 2} \]

\[(%i3) \quad a*x + f(b, y); \quad \frac{a x + f(b, y)}{x + 2} \]
\%(o3) \quad \frac{a x + f(b, y)}{x + 2} \]

\[(%i4) \quad \text{substpart} \left("+", \%, 1, 0\right); \quad x + f(b, y) + a \]
\%(o4) \quad x + f(b, y) + a \]

Also, setting the option \text{inflag} to \text{true} and calling \text{part} or \text{substpart} is the same as calling \text{inpart} or \text{substinpart}.

**Function** subvarp \((\text{expr})\) Returns \text{true} if \text{expr} is a subscripted variable, for example \(a[i]\).

**Function** symbolp \((\text{expr})\) Returns \text{true} if \text{expr} is a symbol, else \text{false}. In effect, \text{symbolp}(x) is equivalent to the predicate \text{atom}(x) \text{ and not numberp}(x).

See also Section 6.4 [Identifiers], page 67.

**Function** vectorpotential \((\text{givencurl})\) Returns the vector potential of a given curl vector, in the current coordinate system. \text{potentialzeroloc} has a similar role as for \text{potential}, but the order of the left-hand sides of the equations must be a cyclic permutation of the coordinate variables.

**Function** xthru \((\text{expr})\) Combines all terms of \text{expr} (which should be a sum) over a common denominator without expanding products and exponentiated sums as \text{ratsimp} does. \text{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 \text{xthru} before \text{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 \text{ratsimped}.

\[(%i1) \quad ((x+2)^{20} - 2*y)/(x+y)^{20} + (x+y)^{(-19)} - x/(x+y)^{20}; \]

\%(o1) \quad \frac{((x+2)^{20} - 2*y)/(x+y)^{20} + (x+y)^{(-19)} - x/(x+y)^{20}}{x + 2} \]

Function

zeroequiv (expr, v)

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.
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.

<table>
<thead>
<tr>
<th>integrate</th>
<th>next</th>
<th>from</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>at</td>
<td>limit</td>
<td>sum</td>
</tr>
<tr>
<td>for</td>
<td>and</td>
<td>elseif</td>
<td>then</td>
</tr>
<tr>
<td>else</td>
<td>do</td>
<td>or</td>
<td>if</td>
</tr>
<tr>
<td>unless</td>
<td>product</td>
<td>while</td>
<td>thru</td>
</tr>
<tr>
<td>step</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

```maxima
(%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.

```maxima
(%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.

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

6.2 Complex

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 $2 + 3\times%i$ and $2 - 3\times%i$. 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.
6.3 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:

```lisp
(%i1) foo (x) := x^2;

(%o1) foo(x) := x

(%i2) foo (42);

(%o2) 1764

(%i3) 'foo (42);

(%o3) foo(42)

(%i4) 'foo (42), nouns;

(%o4) 1764

(%i5) declare (bar, noun);

(%o5) done

(%i6) bar (x) := x/17;

(%o6) ''bar(x) := --
     17

(%i7) bar (52);

(%o7) bar(52)

(%i8) bar (52), nouns;

(%o8) --
     17

(%i9) integrate (1/x, x, 1, 42);

(%o9) log(42)

(%i10) 'integrate (1/x, x, 1, 42);

(%o10) I - dx
    [ 1
    ] x
    1

(%i11) ev (%), nouns;
```
6.4 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`, `FOO`, and `Foo` are distinct. See Section 3.1 [Lisp and Maxima], page 9 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 3.1 [Lisp and Maxima], page 9 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 = FOO), is (FOO = Foo), is (Foo = foo)];
(%o11) [false, false, false]
(%i12) :lisp (defvar *my-lisp-variable* `'$foo)
*MY-LISP-VARIABLE*
(%i12) ?\*my\-lisp\-variable\*;
(%o12) foo
```
6.5 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.";
    This is a string.
(%i2) s_2 : "Embedded \"double quotes\" and backslash \ characters.";
    Embedded "double quotes" and backslash \ characters.
(%i3) s_3 : "Embedded line termination in this string.";
    Embedded line termination in this string.
(%i4) s_4 : "Ignore the \ line termination \ characters in \ this string.";
    Ignore the line termination characters in this string.
(%i5) stringdisp : false;
    false
(%i6) s_1;
    This is a string.
(%i7) stringdisp : true;
    true
(%i8) s_1;
    "This is a string."
```

6.6 Inequality

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

6.7 Syntax

It is possible to define new operators with specified precedence, to redefine 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^b

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");
prefix ("dd")

(%i2) "dd" (a);
  "dd" (a)

(%i3) infix ("<-")
  "<-"

(%i4) a <- dd b;
  a <- dd b

(%i5) "<-" (a, "dd" (b));
   a <- dd b

(%i6) a <- dd b
   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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Prefix/RBP</th>
<th>Infix/LBP/RBP</th>
<th>Nary/LBP/RBP</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefix</td>
<td>rbp=180</td>
<td></td>
<td></td>
</tr>
<tr>
<td>postfix</td>
<td>lbp=180</td>
<td></td>
<td></td>
</tr>
<tr>
<td>infix</td>
<td>lbp=180, rbp=180</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nary</td>
<td>lbp=180, rbp=180</td>
<td></td>
<td></td>
</tr>
<tr>
<td>matchfix</td>
<td>(binding power not applicable)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nofix</td>
<td>(binding power not applicable)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For comparison, here are some built-in operators and their left and right binding powers.

<table>
<thead>
<tr>
<th>Operator</th>
<th>lbp</th>
<th>rbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>180</td>
<td>20</td>
</tr>
<tr>
<td>::</td>
<td>180</td>
<td>20</td>
</tr>
<tr>
<td>::=</td>
<td>180</td>
<td>20</td>
</tr>
<tr>
<td>!</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>!!</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>140</td>
<td>139</td>
</tr>
<tr>
<td>.</td>
<td>130</td>
<td>129</td>
</tr>
<tr>
<td>*</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>+</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>-</td>
<td>100</td>
<td>134</td>
</tr>
<tr>
<td>=</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>#</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>&gt;</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>&gt;=</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>&lt;</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>&lt;=</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>not</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

The `remove` and `kill` functions 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.

```maxima
(%i1) infix ("##");
(%o1) ##
(%i2) "##" (a, b) := a^b;
    b
(%o2) a ## b := a
(%i3) 5 ## 3;
(%o3) 125
(%i4) remove ("##", op);
(%o4) done
(%i5) 5 ## 3;
Incorrect syntax: # is not a prefix operator
5 ##

 uptake
(%i5) "##" (5, 3);
(%o5) 125
(%i6) infix ("##");
(%o6) ##
```
Chapter 6: Expressions

(\%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)

6.8 Functions and Variables for Expressions

at (expr, [eqn_1, ..., eqn_n]) Function
at (expr, eqn) Function
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);
(\%o1) a
(\%i2) atvalue ('diff (f(x,y), x), x = 0, 1 + y);
(\%o2) 1 + y
(\%i3) printprops (all, atvalue);

\[ \begin{align*}
\frac{d}{dx} f(x, y) &= 0 \\
\frac{d}{dx} f(0, 1) &= 0 \\
2 f(0, 1) &= a
\end{align*} \]

(\%o3) done
(\%i4) diff (4*f(x, y)^2 - u(x, y)^2, x);
(\%o4) 8 f(x, y) (\frac{d}{dx} (f(x, y))) - 2 u(x, y) (\frac{d}{dx} (u(x, y)))
(\%i5) at (%, [x = 0, y = 1]);
Function
\texttt{box} (\texttt{expr})
\texttt{box} (\texttt{expr}, \texttt{a})

Returns \texttt{expr} enclosed in a box. The return value is an expression with \texttt{box} as the operator and \texttt{expr} as the argument. A box is drawn on the display when \texttt{display2d} is \texttt{true}.

\texttt{box} (\texttt{expr}, \texttt{a}) encloses \texttt{expr} in a box labelled by the symbol \texttt{a}. The label is truncated if it is longer than the width of the box.

\texttt{box} evaluates its argument. However, a boxed expression does not evaluate to its content, so boxed expressions are effectively excluded from computations.

\texttt{boxchar} is the character used to draw the box in \texttt{box} and in the \texttt{dpart} and \texttt{lpart} functions.

Examples:
\begin{verbatim}
(%i1) box (a^2 + b^2);
2 2
" b + a "

(%i2) a : 1234;
(%o2) 1234

(%i3) b : c - d;
(%o3) c - d

(%i4) box (a^2 + b^2);
2
" (c - d) + 1522756"

(%i5) box (a^2 + b^2, term_1);

term_1
2
" (c - d) + 1522756"

(%i6) 1729 - box (1729);

1729 - "1729"

(%i7) boxchar: "-";

(%i8) box (\sin(x) + \cos(y));
-\cos(y) + \sin(x)-
\end{verbatim}
**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.

**carg (z)**

Function

Returns the complex argument of `z`. The complex argument is an angle `theta` in \((-\pi, \pi]\) such that \(r \exp (theta \cdot i) = z\) 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:

```lisp
(%i1) carg (1);
(%o1) 0
(%i2) carg (1 + %i);
(%o2) %pi
(%i3) carg (exp (%i));
(%o3) 1
(%i4) carg (exp (%pi * %i));
(%o4) %pi
(%i5) carg (exp (3/2 * %pi * %i));
(%o5) - %pi
(%i6) carg (17 * exp (2 * %i));
(%o6) 2
```

**constant**

Special operator

`declare (a, constant)` declares `a` to be a constant. See `declare`.

**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.

Examples:

```lisp
(%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)

Function declare (a_1, p_1, a_2, p_2, ...)
Assigns the atom or list of atoms a_1 the property or list of properties p_1. When a_1 and/or p_1 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. However, featurep does not recognize some flags; this is a bug.
See also features.
declare recognizes the following properties:
evfun Makes a_1 known to ev so that the function named by a_1 is applied when a_1 appears as a flag argument of ev. See evfun.
evflag Makes a_1 known to the ev function so that a_1 is bound to true during the execution of ev when a_1 appears as a flag argument of ev. See evflag.
bindtest Tells Maxima to trigger an error when a_1 is evaluated unbound.
noun Tells Maxima to parse a_1 as a noun. The effect of this is to replace instances of a_1 with 'a_1 or nounify(a_1), depending on the context.
constant Tells Maxima to consider a_1 a symbolic constant.
scalar Tells Maxima to consider a_1 a scalar variable.
nonscalar Tells Maxima to consider a_1 a nonscalar variable. The usual application is to declare a variable as a symbolic vector or matrix.
nonarray Tells Maxima to consider a_1 not an array. This declaration prevents multiple evaluation of a subscripted variable name.
mainvar Tells Maxima to consider a_1 a "main variable". A main variable succeeds all other constants and variables in the canonical ordering of Maxima expressions, as determined by ordergreatp.
alphabetic Tells Maxima to recognize all characters in a_1 (which must be a string) as alphabetic characters.
feature  Tells Maxima to recognize \texttt{a\_i} as the name of a feature. Other atoms may then be declared to have the \texttt{a\_i} property.

\texttt{rassociative, lassociative}
Tells Maxima to recognize \texttt{a\_i} as a right-associative or left-associative function.

\texttt{nary}
Tells Maxima to recognize \texttt{a\_i} as an n-ary function.
The \texttt{nary} declaration is not the same as calling the \texttt{nary} function. The sole effect of \texttt{declare(foo, nary)} is to instruct the Maxima simplifier to flatten nested expressions, for example, to simplify \texttt{foo(x, foo(y, z))} to \texttt{foo(x, y, z)}.

\texttt{symmetric, antisymmetric, commutative}
Tells Maxima to recognize \texttt{a\_i} as a symmetric or antisymmetric function. \texttt{commutative} is the same as \texttt{symmetric}.

\texttt{oddfun, evenfun}
Tells Maxima to recognize \texttt{a\_i} as an odd or even function.

\texttt{outative}
Tells Maxima to simplify \texttt{a\_i} expressions by pulling constant factors out of the first argument.

When \texttt{a\_i} has one argument, a factor is considered constant if it is a literal or declared constant.

When \texttt{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.

\texttt{multiplicative}
Tells Maxima to simplify \texttt{a\_i} expressions by the substitution \texttt{a\_i(x * y * z * \ldots) \rightarrow a\_i(x) * a\_i(y) * a\_i(z) * \ldots}. The substitution is carried out on the first argument only.

\texttt{additive}
Tells Maxima to simplify \texttt{a\_i} expressions by the substitution \texttt{a\_i(x + y + z + \ldots) \rightarrow a\_i(x) + a\_i(y) + a\_i(z) + \ldots}. The substitution is carried out on the first argument only.

\texttt{linear}
Equivalent to declaring \texttt{a\_i} both \texttt{outative} and \texttt{additive}.

\texttt{integer, noninteger}
Tells Maxima to recognize \texttt{a\_i} as an integer or noninteger variable.

\texttt{even, odd}
Tells Maxima to recognize \texttt{a\_i} as an even or odd integer variable.

\texttt{rational, irrational}
Tells Maxima to recognize \texttt{a\_i} as a rational or irrational real variable.

\texttt{real, imaginary, complex}
Tells Maxima to recognize \texttt{a\_i} as a real, pure imaginary, or complex variable.

\texttt{increasing, decreasing}
Tells Maxima to recognize \texttt{a\_i} as an increasing or decreasing function.
**posfun**  
Tells Maxima to recognize \( a_i \) as a positive function.

**integervalued**  
Tells Maxima to recognize \( a_i \) as an integer-valued function.

Examples:

evfun and evflag declarations.

```
(%i1) declare (expand, evfun);
   (%o1) done
(%i2) (a + b)^3;
   (%o2) 3
(%i3) (a + b)^3, expand;
   (%o3) 3 2 2 3
   \( b + 3 a b + 3 a b + a \)
(%i4) declare (demoivre, evflag);
   (%o4) done
(%i5) exp (a + b*%i);
   (%o5) %i b + a
   \%e
(%i6) exp (a + b*%i), demoivre;
   (%o6) %e (%i sin(b) + cos(b))
```

**bindtest declaration.**

```
(%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
```

**noun declaration.**

```
(%i1) factor (12345678);
   (%o1) 2
(%i2) declare (factor, noun);
   (%o2) done
(%i3) factor (12345678);
   (%o3) factor(12345678)
(%i4) ',%, nouns;
   (%o4) 2
```

**constant, scalar, nonscalar, and mainvar declarations.**

**alphabetic declaration.**
Chapter 6: Expressions

(%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~]

feature declaration.

(%i1) declare (FOO, feature);
(%o1) done
(%i2) declare (x, FOO);
(%o2) done
(%i3) featurep (x, FOO);
(%o3) true

rassociative and lassociative declarations.

nary declaration.

(%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)

symmetric and antisymmetric declarations.

(%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)

oddfun and evenfun declarations.

(%i1) o (- u) + o (u);
(%o1) o(u) + o(- u)
(%i2) declare (o, oddfun);
(%o2) done
(%i3) o (- u) + o (u);
(%o3) 0
(\%i4) \text{e}(- \text{u}) - \text{e}(\text{u})
(\%o4) \text{e}(- \text{u}) - \text{e}(\text{u})
(\%i5) \text{declare} (\text{e}, \text{evenfun});
(\%o5) \text{done}
(\%i6) \text{e}(- \text{u}) - \text{e}(\text{u})
(\%o6) 0

\text{outative declaration.}

(\%i1) \text{F1}(100 \times \text{x});
(\%o1) \text{F1}(100 \times \text{x})
(\%i2) \text{declare} (\text{F1}, \text{outative});
(\%o2) \text{done}
(\%i3) \text{F1}(100 \times \text{x});
(\%o3) 100 \text{F1}(\text{x})
(\%i4) \text{declare} (\text{zz}, \text{constant});
(\%o4) \text{done}
(\%i5) \text{F1}(\text{zz} \times \text{y});
(\%o5) \text{zz} \text{F1}(\text{y})

\text{multiplicative declaration.}

(\%i1) \text{F2}(\text{a} \times \text{b} \times \text{c});
(\%o1) \text{F2}(\text{a} \times \text{b} \times \text{c})
(\%i2) \text{declare} (\text{F2}, \text{multiplicative});
(\%o2) \text{done}
(\%i3) \text{F2}(\text{a} \times \text{b} \times \text{c});
(\%o3) \text{F2}(\text{a}) \text{F2}(\text{b}) \text{F2}(\text{c})

\text{additive declaration.}

(\%i1) \text{F3}(\text{a} + \text{b} + \text{c});
(\%o1) \text{F3}(\text{c} + \text{b} + \text{a})
(\%i2) \text{declare} (\text{F3}, \text{additive});
(\%o2) \text{done}
(\%i3) \text{F3}(\text{a} + \text{b} + \text{c});
(\%o3) \text{F3}(\text{c}) + \text{F3}(\text{b}) + \text{F3}(\text{a})

\text{linear declaration.}

(\%i1) \text{'}\text{sum}(\text{F}(\text{k}) + \text{G}(\text{k}), \text{k}, 1, \text{inf});
\hspace{1cm} \text{inf}
\hspace{1cm} \text{====}
\hspace{1cm} \text{\backslash}
\hspace{1cm} \text{====}
\hspace{1cm} \text{\textgreater} \ \ \text{(G}(\text{k}) + \text{F}(\text{k}))
\hspace{1cm} \text{/}
\hspace{1cm} \text{====}
\hspace{1cm} \text{k = 1}
(\%i2) \text{declare} (\text{nounify (sum)}, \text{linear});
(\%o2) \text{done}
(\%i3) \text{'}\text{sum}(\text{F}(\text{k}) + \text{G}(\text{k}), \text{k}, 1, \text{inf});
\[
\text{inf} \quad \text{inf} \\
\frac{G(k)}{F(k)} \\
k = 1 \\
\]

The function `disolate` 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")`.

The function `dispform` 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`.

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\~%", %); ((%SQRTE 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

Function distrib(expr)
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);
1
(%o4) -------------
 b d + a d + b c + a c

Function dpart(expr, n\_1, ..., n\_k)
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"

Function exp(x)
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.

%emode
Option variable
Default value: true
When %emode is true, %e^(%pi %i x) is simplified as follows.
%e^(%pi %i 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^(%pi %i x) simplifies to %e^(%pi %i y) where y is x - 2 k for some integer k such that abs(y) < 1.
When %emode is false, no special simplification of %e^(%pi %i x) is carried out.

%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.

exptisolate
Option variable
Default value: false
exptisolate, when true, causes isolate (expr, var) to examine exponents of atoms (such as %e) which contain var.

exptsubst
Option variable
Default value: false
exptsubst, when true, permits substitutions such as y for %e^x in %e^(a x).

freeof (x_1, ... , x_n, expr)
Freeof (x_1, expr) returns true if no subexpression of expr is equal to x_1 or if x_1 occurs only as a dummy variable in expr, or if x_1 is neither the noun nor verb form of any operator in expr, and returns false otherwise.
freeof (x_1, ..., x_n, expr) is equivalent to freeof (x_1, expr) and ... and freeof (x_n, expr).
The arguments x_1, ..., x_n 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);
   3    d + c  3
(%o1) cos(a ) b z
     1
(%i2) freeof (z, expr);
(%o2) false
(%i3) freeof (cos, expr);
(%o3) false
(%i4) freeof (a[1], expr);
(%o4) false
(%i5) freeof (cos (a[1]), expr);
(%o5) false
(%i6) freeof (b^(c+d), expr);
(%o6) false
(%i7) freeof ("^", expr);
(%o7) false
(%i8) freeof (w, sin, a[2], sin (a[2]), b*(c+d), expr);
(%o8) true
```

`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);
      5  4  2  3  3  2  4  5
(%o2) b + 5 a b + 10 a b + 10 a b + 5 a b + a
(%i3) freeof (a+b, %);
(%o3) true
(%i4) freeof (a+b, expr);
(%o4) false
(%i5) exp (x);
(%o5) %e^x
(%i6) freeof (exp, exp (x));
```
A summation or definite integral is free of its dummy variable. An indefinite integral is not free of its variable of integration.

\[
\begin{align*}
(\%o6) & \quad \text{true} \\
\text{A summation or definite integral is free of its dummy variable. An indefinite integral is not free of its variable of integration.} \\
(\%i1) \quad \text{freeof (i, 'sum (f(i), i, 0, n));} \\
(\%o1) & \quad \text{true} \\
(\%i2) \quad \text{freeof (x, 'integrate (x^2, x, 0, 1));} \\
(\%o2) & \quad \text{true} \\
(\%i3) \quad \text{freeof (x, 'integrate (x^2, x));} \\
(\%o3) & \quad \text{false}
\end{align*}
\]

\textbf{genfact (x, y, z)}

Returns the generalized factorial, defined as \(x (x-z) (x - 2 z) \ldots (x - (y - 1) z)\).

Thus, for integral \(x\), \textit{genfact} \((x, x, 1) = x!\) and \textit{genfact} \((x, x/2, 2) = x!!\).

\textbf{imagpart (expr)}

Returns the imaginary part of the expression \(expr\).

\textit{imagpart} is a computational function, not a simplifying function.

See also \textit{abs}, \textit{carg}, \textit{polarform}, \textit{rectform}, and \textit{realpart}.

\textbf{infix (op)}

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.

\textbf{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 \textit{any}).

\textbf{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 \textit{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: \textit{expr}, \textit{clause}, and \textit{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 \( \textit{rbp} \) makes \( \textit{op} \) left-associative. If \( \textit{lbp} \) is equal to \( \textit{rbp} \), \( \textit{op} \) is left-associative.

See also Syntax.

Examples:

If the left and right binding powers of \( \textit{op} \) are both greater the left and right binding powers of some other operator, then \( \textit{op} \) takes precedence over the other operator.

\[
(\texttt{i1}) \text{lisp (get '$+' lbp)}
100
(\texttt{i1}) \text{lisp (get '$+' rbp)}
100
(\texttt{i1}) \text{infix ("##", 101, 101)};
\texttt{o1}

(\texttt{i2}) "##"(a, b) := sconcat("(" a, ",", b, ")")$;
\texttt{o2}

(\texttt{i3}) 1 + a ## b + 2;
\texttt{o3}

(\texttt{i4}) \text{infix ("##", 99, 99)};
\texttt{o4}

(\texttt{i5}) 1 + a ## b + 2;
\texttt{o5}

Greater \( \textit{lbp} \) makes \( \textit{op} \) right-associative, while greater \( \textit{rbp} \) makes \( \textit{op} \) left-associative.

\[
(\texttt{i1}) \text{infix ("##", 100, 99)};
\texttt{o1}

(\texttt{i2}) "##"(a, b) := sconcat("(" a, ",", b, ")")$;
\texttt{o2}

(\texttt{i3}) foo ## bar ## baz;
\texttt{o3}

(\texttt{i4}) \text{infix ("##", 100, 101)};
\texttt{o4}

(\texttt{i5}) foo ## bar ## baz;
\texttt{o5}

Maxima can detect some syntax errors by comparing the declared part of speech to an actual expression.

\[
(\texttt{i1}) \text{infix ("##", 100, 99, expr, expr, expr)};
\texttt{o1}

(\texttt{i2}) \text{if x ## y then 1 else 0};
\texttt{o2}

\texttt{i2} \text{Incorrect syntax: Found algebraic expression where logical expression expected}
\texttt{i2} \text{if x ## y then}
\texttt{i2} \text{else 0}$

\[
(\texttt{i2}) \text{infix ("##", 100, 99, expr, expr, clause)};
\texttt{o2}

(\texttt{i3}) \text{if x ## y then 1 else 0};
\texttt{o3}

\texttt{i3} \text{if x ## y then 1 else 0}$

\textbf{inflag} \quad \text{Option variable}

\text{Default value: false}

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

Also, setting \texttt{inflag} to \texttt{true} and calling \texttt{part} or \texttt{subpart} is the same as calling \texttt{inpart} or \texttt{subinpart}.

Functions affected by the setting of \texttt{inflag} are: \texttt{part}, \texttt{subpart}, \texttt{first}, \texttt{rest}, \texttt{last}, \texttt{length}, the for ... in construct, \texttt{map}, \texttt{fullmap}, \texttt{maplist}, \texttt{reveal} and \texttt{pickapart}.

\begin{verbatim}
inpart \((expr, n_1, ..., n_k)\)

is similar to \texttt{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). \texttt{part} \((x*y, 0)\) or \texttt{inpart} \((x+y, 0)\) yield +, though in order to refer to the operator it must be enclosed in "s. For example ... if \texttt{inpart} \((%o9,0) = "+" \) then ....

Examples:
\begin{verbatim}
(%i1) x + y + w*z;
(%o1) w z + y + x
(%i2) inpart (%o1, 3, 2);
(%o2) z
(%i3) part (%o3, 1, 2);
(%o3) z
(%i4) 'limit (f(x)^g(x+1), x, 0, minus);
   g(x + 1)
   \texttt{limit} f(x)
(%o4) x \rightarrow 0-
(%i5) inpart (%o5, 1, 2);
(%o5) g(x + 1)
\end{verbatim}
\end{verbatim}

\texttt{isolate} \((expr, x)\)

Returns \texttt{expr} with subexpressions which are sums and which do not contain \texttt{var} replaced by intermediate expression labels (these being atomic symbols like \texttt{%t1}, \texttt{%t2}, \texttt{...}). 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.

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

\texttt{isolate_wrt_times} if \texttt{true}, then \texttt{isolate} will also isolate with respect to products. See \texttt{isolate_wrt_times}.

Do example (isolate) for examples.

\texttt{isolate_wrt_times}

Default value: \texttt{false}
When \texttt{isolate\_wrt\_times} is \texttt{true}, \texttt{isolate} will also isolate with respect to products. E.g. compare both settings of the switch on

\begin{verbatim}
(%i1) isolate\_wrt\_times: true$
(%i2) isolate (expand ((a+b+c)^2), c);

(%t2)
 2 a

(%t3)
 2 b

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

(%o4) c + %t3 c + %t2 c + %t4

(%i4) isolate\_wrt\_times: false$
(%i5) isolate (expand ((a+b+c)^2), c);

(%o5) c + 2 b c + 2 a c + %t4
\end{verbatim}

\textbf{listconstvars} \quad Option variable

Default value: \texttt{false}

When \texttt{listconstvars} is \texttt{true}, it will cause \texttt{listofvars} to include \texttt{e}, \texttt{pi}, \texttt{i}, and any variables declared constant in the list it returns if they appear in the expression \texttt{listofvars} is called on. The default is to omit these.

\textbf{listdummyvars} \quad Option variable

Default value: \texttt{true}

When \texttt{listdummyvars} is \texttt{false}, "dummy variables" in the expression will not be included in the list returned by \texttt{listofvars}. (The meaning of "dummy variables" is as given in \texttt{freeof}. "Dummy variables" are mathematical things like the index of a sum or product, the limit variable, and the definite integration variable.) Example:

\begin{verbatim}
(%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]
\end{verbatim}

\textbf{listofvars} \quad Function

Returns a list of the variables in \texttt{expr}.

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

See also the option variable \texttt{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]

lfreeof (list, expr)  
Function  
For each member \( m \) of \( \text{list} \), calls \( \text{freeof} (m, \text{expr}) \). It returns \( \text{false} \) if any call to \( \text{freeof} \) does and \( \text{true} \) otherwise.

lopow (expr, x)  
Function  
Returns the lowest exponent of \( x \) which explicitly appears in \( \text{expr} \). Thus
(%i1) lopow ((x+y)^2 + (x+y)^a, x+y);
(%o1) \( \min(a, 2) \)

lpart (label, expr, n_1, ..., n_k)  
Function  
is similar to \( \text{dpart} \) but uses a labelled box. A labelled box is similar to the one produced by \( \text{dpart} \) but it has a name in the top line.

multthru (expr)  
Function  
multthru (expr_1, expr_2)  
Function  
Multiplies a factor (which should be a sum) of \( \text{expr} \) by the other factors of \( \text{expr} \). That is, \( \text{expr} \) is \( f_1 f_2 \ldots f_n \) 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. \( \text{Namely all the factors except} \ f_i \). \text{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 \text{multthru} can be used to divide sums by products as well.

\text{multthru} (\text{expr}_1, \text{expr}_2) \text{multiplies each term in} \ \text{expr}_2 \ (\text{which should be a sum or an equation}) \ \text{by} \ \text{expr}_1. \ \text{If} \ \text{expr}_1 \ \text{is not itself a sum \ then this form is equivalent to} \ \text{multthru} (\text{expr}_1*\text{expr}_2).

(%i1) x/(x-y)^2 - 1/(x-y) - f(x)/(x-y)^3;
1
x
-------- + ---------- - ----------
\( x - y \) \( x - y \) \( x - y \)
(%o1) \( \frac{f(x)}{x - y} \)

(%i2) multthru ((x-y)^3, %);  
(%o2) \( - (x - y) + x (x - y) - f(x) \)

(%i3) ratexpand (%);  
(%o3) \( - y + x y - f(x) \)

(%i4) ((a+b)^10*s^2 + 2*a*b*s + (a*b)^2)/(a*b*s^2);
10
2
(b + a) s + 2 a b s + a b
2
\( a b s \)
(%o4) \( \frac{2}{a b s} \)

(%i5) multthru (%); /* note that this does not expand (b+a)^10 */
\[
\frac{2}{s} \frac{a}{s} \frac{b}{s} (b + a)
\]

\[(%o5)\]
\[
\frac{2}{s} \frac{a}{s} \frac{b}{s} \frac{b}{s} + \frac{a}{s} \frac{b}{s} \frac{b}{s} \frac{a}{s} \frac{b}{s} (b + a)
\]

s \quad 2 \quad a \quad b

\[(%i6)\] multthru \((a \cdot (b + c \cdot (d + e) + f)))

\[(%o6)\]
\[
a \cdot f + a \cdot c \cdot (e + d) + a \cdot b
\]

\[(%i7)\] expand \((a \cdot (b + c \cdot (d + e) + f)))

\[(%o7)\]
\[
a \cdot f + a \cdot c \cdot e + a \cdot c \cdot d + a \cdot b
\]

**nonarray**

The command `declare(a, nonarray)` tells Maxima to consider \(a\) not an array. This declaration prevents multiple evaluation, if \(a\) is a subscripted variable.

Example:

\[(%i1)\]
\[
a:'b$ b:'c$ c:'d$
\]

\[(%i4)\]
\[
a[x];
\]

\[(%o4)\]
\[
d
\]

\[(%i5)\] declare(a, nonarray);

\[(%o5)\]
\[
done
\]

\[(%i6)\]
\[
a[x];
\]

\[(%o6)\]
\[
a
\]

**nounify (f)**

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.

**nterms (expr)**

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)**

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 an array 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
```

\textbf{operatorp} \hspace{1em} Function \hspace{1em} \textbf{optimize} \hspace{1em} Function \hspace{1em} \textbf{optimprefix} \hspace{1em} Option variable

\textbf{operatorp} (\textit{expr}, \textit{op})

\textbf{operatorp} returns true if \textit{op} is equal to the operator of \textit{expr}.

\textbf{operatorp} (\textit{expr}, \textit{op}) \hspace{1em} returns \textbf{true} \hspace{1em} if \hspace{1em} \textit{op} \hspace{1em} is \hspace{1em} equal \hspace{1em} to \hspace{1em} the \hspace{1em} operator \hspace{1em} of \hspace{1em} \textit{expr}.

\textbf{optimize} (\textit{expr})

\textbf{optimize} returns an expression that produces the same value and side effects as \textit{expr} but does so more efficiently by avoiding the recomputation of common subexpressions. \textbf{optimize} also has the side effect of "collapsing" its argument so that all common subexpressions are shared. Do \textbf{example (optimize)} for examples.

\textbf{optimprefix}

\textbf{optimprefix} is the prefix used for generated symbols by the \textbf{optimize} command.

Default value: \%

\textbf{optimprefix} is the prefix used for generated symbols by the \textbf{optimize} command.
ordergreat \( (v_1, \ldots, v_n) \)

orderless \( (v_1, \ldots, v_n) \)

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 ordgreatep.

ordergreatp \( (expr_1, expr_2) \)

orderlessp \( (expr_1, expr_2) \)

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.

\[
(\%i1) \text{stringdisp : true;}
(\%o1) \text{true}
(\%i2) \text{sort([\%pi, 3b0, 3.0, x, X, "foo", 3, a, 4, "bar", 4.0, 4b0]);}
\]
Chapter 6: Expressions

(%o2) [3, 3.0, 4, 4.0, 3.0b0, 4.0b0, %pi, "bar", "foo", a, x, X]

Effect of ordergreat and orderless functions.

(%i1) sort ([M, H, K, T, E, W, G, A, P, J, S]);
(%o2) ordergreat (S, J);
(%o3) done
(%o4) orderless (M, H);
(%o5) done
(%o6) sort ([M, H, K, T, E, W, G, A, P, J, S]);

Effect of mainvar, constant, and scalar declarations.

(%i1) sort ([aa, foo, bar, bb, baz, quux, cc, dd, A1, B1, C1]);
(%o1) [aa, bar, baz, bb, cc, dd, foo, quux, A1, B1, C1]
(%i2) declare (aa, mainvar);
(%o2) done
(%i3) declare ([baz, quux], constant);
(%o3) done
(%i4) declare ([A1, B1], scalar);
(%o4) done
(%i5) sort ([aa, foo, bar, bb, baz, quux, cc, dd, A1, B1, C1]);
(%o5) [baz, quux, A1, B1, bar, bb, cc, dd, foo, C1, aa]

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

part (expr, n_1, ..., n_k)

Function

Returns parts of the displayed form of expr. It obtains the part of expr as specified by the indices n_1, ..., n_k. First part n_1 of expr is obtained, then part n_2 of that, etc. The result is part n_k of ... part n_2 of part n_1 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 parts with 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

(%i2) part(z+2*y+a,[1,3]);
(%o2) z + a

(%i3) part(z+2*y+a,2,1);
(%o3) 2

declares additional examples.

Function

partition (expr, x)

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.

(%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)]]

Option variable

partswitch

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.

Function

pickapart (expr, n)

Assigns intermediate expression labels to subexpressions of expr at depth n, an in-
teger. 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

(%i2) pickapart (expr, 0);

(%o2) %t2

(%i3) pickapart (expr, 1);

(%o3) %t3
\[
\frac{2}{3} \sin(x)
\]

\[
\frac{b + a}{2}
\]

\[
\frac{\%t5 + \%t4 + \%t3}{2}
\]

\[
\%t5 + \%t4 + \%t3
\]

\[
\text{pickapart (expr, 2)}
\]

\[
\log(\sqrt{x + 1} + 1)
\]

\[
\frac{2}{3} \sin(x)
\]

\[
\frac{b + a}{2}
\]

\[
\frac{\%t8}{2} \frac{\%t7}{3} - \%t6
\]

\[
\text{pickapart (expr, 3)}
\]

\[
\sqrt{x + 1} + 1
\]

\[
\frac{2}{3} \sin(x)
\]

\[
\frac{b + a}{2} \sin(\%t10)
\]

\[
\frac{\log(\%t9) + \%t10}{3}
\]

\[
\text{pickapart (expr, 4)}
\]

\[
\sqrt{x + 1}
\]

\[
\frac{2}{3} \sin(x)
\]

\[
\frac{b + a}{2} \sin(\%t11 + 1)
\]

\[
\text{pickapart (expr, 5)}
\]

\[
x + 1
\]
\[
\frac{2}{3} \sin(x) b + a \\
\text{(\%o12) } \frac{2}{3} \sin(x) b + a - \log(\sqrt{x + 1} + 1) \\
\text{(\%i12) pickapart (expr, 6);} \\
\frac{2}{3} \sin(x) b + a \\
\text{(\%o12) } \frac{2}{3} \sin(x) b + a - \log(\sqrt{x + 1} + 1)
\]

**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.

**polarform**  
Function  
Returns an expression \( r e^{i \theta} \) equivalent to `expr`, such that \( r \) and \( \theta \) are purely real.

**powers**  
Function  
Gives the powers of \( x \) occurring in `expr`.

`load (powers)` loads this function.

**product**  
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
(%i5) product (k, k, 1, n);

n
/===\  
!!  k  
!!  k = 1
(%i6) product (k, k, 1, n), simpproduct;

n
/===\  
!!  1  
!!  k + 1  
k = 1
(%i7) product (k, k, 1, n), simpproduct;

n
/===\  
!!  1  
!!  -------  
!!  k + 1  
k = 1
(%i8) product (if k <= 5 then a^k else b^k, k, 1, 10);

15 40
(a  b)

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.

rectform (expr) Function
Returns an expression a + b %i equivalent to expr, such that a and b are purely real.

rembox (expr, unlabelled) Function
rembox (expr, label) Function
rembox (expr) Function
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);

(a  d - b  c)

(%o1)  sin(%pi x) + ---------
     2
(%i2) dpart (dpart (expr, 1, 1), 2, 2);  

  h

(%o2) sin("%pi x") + "-" ****

  "2"

  "h"

(%i3) expr2: lpart (BAR, lpart (FOO, %, 1), 2);

(FOO) ****

 "a d - b c"

(%o3) "sin("%pi x")" + "-" ****

  "2"

  "h"

(%i4) rembox (expr2, unlabelled);

(FOO) ****

 "a d - b c"

(%o4) "sin("%pi x")" + "-" ****

  "2"

  "h"

(%i5) rembox (expr2, FOO);

(BAR) ****

 "a d - b c"

(%o5) "sin("%pi x")" + "-" ****

  "2"

  "h"

(%i6) rembox (expr2, BAR);

(FOO) ****

 "a d - b c"

(%o6) "sin("%pi x")" + "-" ****

  "2"

  "h"

(%i7) rembox (expr2);

(%o7) sin("%pi x") + "-" 2

  h
**Function**

`sum(expr, i, i_0, i_1)`

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 `simpsum` 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) `cauchy_sum` 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 `sumcontract`, `intosum`, `bashindices`, `niceindices`, `nouns`, `evflag`, and `zeilberger`.

Examples:

```lisp
(%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
    \    >  a(i)
   /   
  ==
 i = 1
(%i5) sum(2^i + i^2, i, 0, n);
    n
    \  i  2
   /  >  (2 + i )
  ==
 i = 0
(%i6) sum(2^i + i^2, i, 0, n), simpsum;
```
\[
\begin{array}{c}
\text{(%o6)} \\
\frac{n + 1}{2} + \frac{2n + 3}{n} + n - 1
\end{array}
\]

\[
\text{(%i7)} \text{sum} \left(\frac{1}{3^i}, i, 1, \text{inf}\right);
\]

\[
\begin{array}{c}
\text{inf} \\
\frac{1}{i}
\end{array}
\]

\[
\text{(%o7)}
\]

\[
\text{(%i8)} \text{sum} \left(\frac{1}{3^i}, i, 1, \text{inf}\right), \text{simpsum} ;
\]

\[
\frac{1}{2}
\]

\[
\text{(%o8)}
\]

\[
\text{(%i9)} \text{sum} \left(i^2, i, 1, 4\right) * \text{sum} \left(\frac{1}{i^2}, i, 1, \text{inf}\right) ;
\]

\[
\frac{1}{2}
\]

\[
\text{(%o9)}
\]

\[
\text{(%i10)} \text{sum} \left(i^2, i, 1, 4\right) * \text{sum} \left(\frac{1}{i^2}, i, 1, \text{inf}\right), \text{simpsum} ;
\]

\[
\frac{5}{2} \pi
\]

\[
\text{(%o10)}
\]

\[
\text{(%i11)} \text{sum} \left(\text{integrate} \left(x^k, x, 0, 1\right), k, 1, n\right) ;
\]

\[
\frac{1}{k + 1}
\]

\[
\text{(%o11)}
\]

\[
\text{(%i12)} \text{sum} \left(\text{if } k \leq 5 \text{ then } a^k \text{ else } b^k, k, 1, 10\right) ;
\]

\[
5 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2
\]

\[
\text{(%o12)}
\]

\[
\text{Function}
\]

\[
lsum \left(\text{expr}, x, L\right)
\]

Represents the sum of \text{expr} for each element \(x\) in \(L\). A noun form \(lsum\) is returned if the argument \(L\) does not evaluate to a list.

Examples:

\[
\begin{array}{c}
\text{(%i1)} \text{lsum} \left(x^i, i, [1, 2, 7]\right) ; \\
\text{(%o1)} x + x + x
\end{array}
\]

\[
\begin{array}{c}
\text{(%i2)} \text{lsum} \left(i^2, i, \text{roots}\left(x^3 - 1\right)\right) ;
\end{array}
\]
unorder ()

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;
2
(%o2) b x + a
(%i3) ordergreat(a);
(%o3) done
(%i4) b*x+a^2;
2
(%o4) a + b x
(%i5) %th(1)-%th(3);
2 2
(%o5) a - a
(%i6) unorder();
(%o6) [a]
(%i7) %th(2);
2 2
(%o7) _101a - a

verbify (f)

Returns the verb form of the function name f. See also verb, noun, and nounify.

Examples:

(%i1) verbify ('foo);
(%o1) foo
(%i2) :lisp %
$FOO
(%i2) nounify (foo);
(%o2) foo
(%i3) :lisp $%
%FOO
7 Simplification

7.1 Functions and Variables for Simplification

**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.

**askinteger** *(expr, integer)*

Function

**askinteger** *(expr)*

Function

**askinteger** *(expr, even)*

Function

**askinteger** *(expr, odd)*

Function

*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*.

**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.

**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:

```lisp
(%i1) sin([x,1,1.0]);
(%o1) [sin(x), sin(1), .8414709848078965]
```

`mod` is a function with two arguments which maps over lists. Mapping over nested lists is possible too:

```lisp
(%i2) mod([x,11,2*a],10);
(%o2) [mod(x, 10), 1, 2 mod(a, 5)]
(%i3) mod([[x,y,z],11,2*a],10);
(%o3) [[mod(x, 10), mod(y, 10), mod(z, 10)], 1, 2 mod(a, 5)]
```

Mapping of the `floor` function over a matrix and an equation:

```lisp
(%i4) floor(matrix([a,b],[c,d]));
[ floor(a) floor(b) ]
[ floor(c) floor(d) ]
(%i5) floor(a=b);
floor(a) = floor(b)
```

Functions with more than one argument map over any of the arguments or all arguments:

```lisp
(%i6) expintegral_e([1,2],[x,y]);
[ 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`:

```lisp
(%i7) properties(abs);
[ integral, distributes over bags, noun, rule, gradef]
```

**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)`.

**expand**

Function

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);
   3  2
(%o1) (x + 1) (y + 1)

(%i2) expand(expr);
   3 2 3 2 2 2 2 2 3
(%o2) x y + 2 x y + y + 3 x y + 6 x y + 3 y + 3 x y + 6 x y + 3 y + x + 2 x + 1

(%i3) expand(expr, 2);
   3 3
(%o3) x (y + 1) + 2 x (y + 1) + (y + 1)

(%i4) expr:(x+1)^(-2)*(y+1)^(3);
   3
(%o4) (y + 1)
   --------------
       2
       (x + 1)

(%i5) expand(expr);
   3 2
(%o5) -- + -- + -- + --
    2 2 2 2

 x + 2 x + 1 x + 2 x + 1 x + 2 x + 1

(%i6) expand(expr, 2, 2);
Resimplify an expression without expansion:

\[(\%i7)\] \(\text{expr:}\,(1+x)^2\sin(x)\);

\[(\%o7)\] \((x + 1) \sin(x)\)

\[(\%i8)\] exponentialize:true;

\[(\%o8)\] true

\[(\%i9)\] expand(expr,0,0);

\[(\%o9)\] \(\frac{2 \text{i} x - \text{i} x}{\text{i} (x + 1) - \text{e}}\)

**expandwrt** \((\text{expr,}\,x_1,\ldots,\,x_n)\)

Function

Expands expression \(\text{expr}\) with respect to the variables \(x_1,\ldots,\,x_n\). 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. \(x_1,\ldots,\,x_n\) 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** \((\text{expr,}\,x_1,\ldots,\,x_n)\)

Function

is similar to **expandwrt**, but treats expressions that are products somewhat differently. **expandwrt_factored** expands only on those factors of \(\text{expr}\) that contain the variables \(x_1,\ldots,\,x_n\).

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 \(n\).

factlim  
Option variable  
Default value: -1  
factlim specifies the highest factorial which is automatically expanded. If it is -1 then all integers are expanded.

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.

lassociative  
Declaration  
declare (g, lassociative) tells the Maxima simplifier that \(g\) is left-associative. 
E.g., \(g (g (a, b), g (c, d))\) will simplify to \(g (g (a, b), c, d)\).

linear  
Declaration  
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 \(x\). 
linear is equivalent to additive and outative. See also opproperties.

mainvar  
Declaration  
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 \(\text{declare} (x, \text{mainvar}), \text{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 \texttt{mainvar} from one in which \( x \) isn’t a \texttt{mainvar}, resimplification e.g. with \texttt{ev (expr, simp)} may be necessary if cancellation is to occur. Also, if you save an expression in which \( x \) is a \texttt{mainvar}, you probably should also save \( x \).)

\textbf{maxapplydepth}

Option variable

Default value: 10000

\texttt{maxapplydepth} is the maximum depth to which \texttt{apply1} and \texttt{apply2} will delve.

\textbf{maxapplyheight}

Option variable

Default value: 10000

\texttt{maxapplyheight} is the maximum height to which \texttt{applyb1} will reach before giving up.

\textbf{maxnegex}

Option variable

Default value: 1000

\texttt{maxnegex} is the largest negative exponent which will be expanded by the \texttt{expand} command (see also \texttt{maxposex}).

\textbf{maxposex}

Option variable

Default value: 1000

\texttt{maxposex} is the largest exponent which will be expanded with the \texttt{expand} command (see also \texttt{maxnegex}).

\textbf{multiplicative}

Declaration

\texttt{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 \( f(x)*f(y) \).

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 \( f (g(x), x) * f (h(x), x) \).

This simplification does not occur when \( f \) is applied to expressions of the form \texttt{product (x[i], i, m, n)}.

\textbf{negdistrib}

Option variable

Default value: true

When \texttt{negdistrib} is \texttt{true}, \( -1 \) distributes over an expression. E.g., \( -(x + y) \) becomes \( -y - x \). Setting it to \texttt{false} will allow \( -(x + y) \) to be displayed like that. This is sometimes useful but be very careful: like the \texttt{simp} flag, this is one flag you do not want to set to \texttt{false} as a matter of course or necessarily for other than local use in your Maxima.

\textbf{negsumdispflag}

Option variable

Default value: true

When \texttt{negsumdispflag} is \texttt{true}, \( x - y \) displays as \( x - y \) instead of as \( -y + x \). Setting it to \texttt{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.
noeval

*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.

noun

*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.

noundisp

Default value: *false*

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

nouns

*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*.

numer

*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:

```lisp
(%i1) [sqrt(2), sin(1), 1/(1+sqrt(3))];
  1
(%o1) [sqrt(2), sin(1), ----------------]
      sqrt(3) + 1
(%i2) [sqrt(2), sin(1), 1/(1+sqrt(3))],numer;
(%o2) [1.414213562373095, .8414709848078965, .3660254037844387]
```

numerval

Declarations the variables *x_1*, ..., *x_n* to have numeric values equal to *expr_1*, ..., *expr_n*. The numeric value is evaluated and substituted for the variable in any expressions in which the variable occurs if the *numer* flag is *true*. See also *ev*.

The expressions *expr_1*, ..., *expr_n* can be any expressions, not necessarily numeric.

opproperties

*opproperties* is the list of the special operator properties recognized by the Maxima simplifier: *linear*, *additive*, *multiplicative*, *outative*, *evenfun*, *oddfun*, *commutative*, *symmetric*, *antisymmetric*, *nary*, *lassociative*, *rassociative*.

opsubst

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.
outative

\texttt{declare (f, outative)} tells the Maxima simplifier that constant factors in the argument of \texttt{f} can be pulled out.

1. If \texttt{f} is univariate, whenever the simplifier encounters \texttt{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., \texttt{f(a*x)} will simplify to \texttt{a*f(x)} where \texttt{a} is a constant. Non-atomic constant factors will not be pulled out.

2. If \texttt{f} is a function of 2 or more arguments, outativity is defined as in the case of \texttt{sum} or \texttt{integrate}, i.e., \texttt{f (a*g(x), x)} will simplify to \texttt{a * f(g(x), x)} for \texttt{a} free of \texttt{x}.

\texttt{sum}, \texttt{integrate}, and \texttt{limit} are all \texttt{outative}.

posfun

\texttt{declare (f, posfun)} declares \texttt{f} to be a positive function. \texttt{is (f(x) > 0)} yields \texttt{true}.

pred

As an argument in a call to \texttt{ev (expr)}, \texttt{pred} causes predicates (expressions which evaluate to \texttt{true} or \texttt{false}) to be evaluated. See \texttt{ev}.

Example:

\begin{verbatim}
(%i1) 1<2;
(%o1) 1 < 2
(%i2) 1<2,pred;
(%o2) true
\end{verbatim}

radcan \texttt{(expr)}

Simplifies \texttt{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, \texttt{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 \texttt{radcan} to zero.

For some expressions \texttt{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:

\begin{verbatim}
(%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
\end{verbatim}
radexpand
Option variable

Default value: true

radexpand controls some simplifications of radicals.

When radexpand is all, causes nth roots of factors of a product which are powers of n to be pulled outside of the radical. E.g. if radexpand is all, \( \sqrt[n]{16x^2} \) simplifies to \( 4x \).

More particularly, consider \( \sqrt{n^2} \).

- 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 \( \text{abs}(x) \).
- 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.

radsubstflag
Option variable

Default value: false

radsubstflag, if true, permits ratsubst to make substitutions such as \( u \) for \( \sqrt{x} \) in \( x \).

rassociative
Declaration

declare \((g, \text{rassociative})\) 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 \((\text{expr}, \text{rule}_1, \ldots, \text{rule}_n)\)
Function

Sequential Comparative Simplification (method due to Stoute). scsimp attempts to simplify \( \text{expr} \) according to the rules \( \text{rule}_1, \ldots, \text{rule}_n \). 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 standard. 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 can not suppress the simplification which follows the evaluation phase.

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) .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.

(%i4) simp:true;
(%o4) true
(%i5) x:sin(1.0),simp:false;
(%o5) .8414709848078965
(%i6) :lisp $X
((%SIN) 1.0)

simpsum
Option variable
Default value: false

When simpsum is true, the result of a sum is simplified. This simplification may sometimes be able to produce a closed form. If simpsum 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.

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:

(%i11) sumexpand: true$
(%i12) sum (f (i), i, 0, m) * sum (g (j), j, 0, n);
\( f(i1) g(i2) \)

\( i1 = 0 \quad i2 = 0 \)

\( \sum (f(i), i, 0, m)^2; \)

\( i3 = 0 \quad i4 = 0 \)

**sumsplitfact**  
Option variable  
**Default value:** true  
When `sumsplitfact` is false, `minfactorial` is applied after a `factcomb`.

**symmetric**  
Declaration  
`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`.

**unknown**  
Function  
Returns `true` if and only if `expr` contains an operator or function not recognized by the Maxima simplifier.
8 Plotting

8.1 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.interface`, where `interface` is the name of the plotting interface being used (gnuplot, xmaxima, mgnuplot or gnuplot_pipes).

The `maxout.interface` file, in the cases when it is used, is created in the directory specified by the system variable `maxima_tempdir`. That location can be changed; by assigning to that variable a string that represents a valid directory where Maxima can create new files.

After a plot has been created, the file `maxout.interface` can be executed again with the appropriate external program. If a Maxima plotting command fails to show anything, that file can be inspected to look for possible sources of problems.

8.2 Plotting Formats

There are currently two external plotting programs that Maxima use: Gnuplot and Xmaxima. There are various different formats for those programs, which can be selected with the option `plot_format` (see 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.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.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 (see `gnuplot_replot`).

  This format should only be used to plot to the screen; for plotting to files it is better to use the gnuplot format.

- **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, should be installed. If Maxima is being run from xmaxima itself, this format will make the plot functions send the data and commands through the same socket used for the communication between Maxima and Xmaxima. When used from the console or from other interface, the commands and data will be saved in the file maxout.xmaxima, and the xmaxima program will be launched with the name of the location of that file as argument.

In previous versions this format used to be called openmath; that old name will still be accepted as a synonym for xmaxima.

8.3 Functions and Variables for Plotting

\texttt{contour_plot (expr, x\_range, y\_range, options, \ldots)}

Function

It plots the contours (curves of equal value) of \textit{expr} over the region \textit{x\_range} by \textit{y\_range}. Any additional arguments are treated the same as in \texttt{plot3d}.

This function only works when the plot format is either gnuplot or gnuplot\_pipes. The additional package implicit\_plot can also be used to plot contours and it works for any format. See implicit\_plot.

Examples:

\begin{verbatim}
(%i1) contour_plot (x^2 + y^2, [x, -4, 4], [y, -4, 4])$

(%i1) F(x, y) := x^3 + y^2;

(%o1) F(x, y) := x^3 + y^2
\end{verbatim}
You can add any options accepted by \texttt{plot3d}; for instance, the option \texttt{legend} with a value of \texttt{false}, to remove the legend. GnuPlot chooses, by default, 3 contours to show. To increase the number of levels, it is necessary to specify a custom gnuPlot preamble:

\begin{verbatim}
(%i1) contour_plot (u^3 + v^2, [u, -4, 4], [v, -4, 4],
[legend,false],
[gnuplot_preamble, "set cntrparam levels 12"])$
\end{verbatim}

\begin{itemize}
\item \textbf{get\_plot\_option} \texttt{(keyword, index)} \hspace{2cm} \textbf{Function}
\item Returns a value of the option with name \texttt{keyword}, stored in the global variable \texttt{plot\_options}. A value of 1 for the index will return the keyword itself; a value of 2 turns returns the first parameter following the keyword, and so on.
\item See also \texttt{plot\_options}, \texttt{set\_plot\_option} and the section on Plotting Options.
\item \textbf{make\_transform} \texttt{([var1, var2, var3], fx, fy, fz)} \hspace{2cm} \textbf{Function}
\item Returns a function suitable to be used in the option \texttt{transform\_xy} of plot3d. The three variables \texttt{var1}, \texttt{var2}, \texttt{var3} are three dummy variable names, which represent
\end{itemize}
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 $f_x$, $f_y$, $f_z$ 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.

**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 should also be a list with two values that correspond to the $x$ and $y$ coordinates of a point.  

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])$``

\[
\begin{array}{c}
\text{Plot of a common function:} \\
\end{array}
\]

If the functions 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])$``

\[
\begin{array}{c}
\text{If the functions grows too fast, it might be necessary to limit the values in the vertical axis using the *y* option:} \\
\end{array}
\]

The aspect of the plot might be different depending on the plotting program used. For instance, when the plot box is disable, Xmaxima will plot the axes using arrows:
A plot with a logarithmic scale:

```
(%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 $
```
Chapter 8: Plotting

(%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], [nticks, 2000]])$

A “circle” with two turns, when plotted with only 7 points:
(%i1) plot2d ([parametric, cos(t), sin(t),
    [t, -2*%pi, 2*%pi], [nticks, 8]])$

Plot of a common function together with the parametric representation of a circle. The size of the plot has been adjusted with the x and y options, to make the circle look round and not deformed as an ellipse. These values work well for the Postscript terminal used to produce this plot; you might have to adjust the values for your screen.

(%i1) plot2d([[parametric, cos(t), sin(t),
    [t, 0, 2*%pi], [nticks, 80]],
    abs(x)], [x, -2, 2], [y, -1.5, 1.5])$
plot2d: some values were clipped.

A plot of a discrete set of points, defining x and y coordinates separately:
The same points shown in the previous example, defining each point separately and without any lines joining the points:

\begin{verbatim}
(%i1) plot2d([discrete, [[10, .6], [20, .9], [30, 1.1],
                  [40, 1.3], [50, 1.4]]],
              [style, points])$
\end{verbatim}

In this 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:

\begin{verbatim}
(%i1) with_stdout ("data.txt", for x:0 thru 10 do
    print (x, x^2, x^3))$
(%i2) data: read_matrix ("data.txt")$
\end{verbatim}
A plot of experimental data points together with the theoretical function that predicts the data:

```
(%i1) xy: [[10, .6], [20, .9], [30, 1.1], [40, 1.3], [50, 1.4]]$
(%i2) plot2d([[discrete, xy], 2*%pi*sqrt(l/980)], [l,0,50],
   [style, points, lines], [color, red, blue],
   [point_type, asterisk],
   [legend, "experiment", "theory"],
   [xlabel, "pendulum's length (cm)"],
   [ylabel, "period (s)"])$
```

See also the section about Plotting Options.

**plot3d**  

```
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 common function:

```latex
(%i1) plot3d (2^(-u^2 + v^2), [u, -3, 3], [v, -2, 2])$
```

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:

```latex
(%i1) plot3d ( log ( x^2*y^2 ), [x, -2, 2], [y, -2, 2], [z, -8, 4], [palette, false], [color, magenta, blue])$
```

The infinite values of z can also be avoided by choosing a grid that does not fall on any asymptotes; this example also shows how to select one of the predefined palettes, in this case the fourth one:
(%i1) plot3d (log (x^2*y^2), [x, -2, 2], [y, -2, 2], [grid, 29, 29], [palette, get_plot_option(palette,5)])$

Two surfaces in the same plot, sharing the same domain; in gnuplot the two surfaces will use the same palette:

(%i1) plot3d ([2^(-x^2 + y^2), 4*sin(3*(x^2+y^2))/(x^2+y^2), [x, -3, 3], [y, -2, 2]])$

The same two surfaces, but now with different domains; in xmaxima each surface will use a different palette, chosen from the list defined by the option palette:
Plot of a Klein bottle, defined parametrically:

\[
\begin{align*}
(\%i1) \ & \ expr_1:5\cos(x)\cos(x/2)\cos(y)+\sin(x/2)\sin(2y)+3.0-10.0 \\
(\%i2) \ & \ expr_2: -5\sin(x)\cos(x/2)\cos(y) + \sin(x/2)\sin(2y) + 3.0 \\
(\%i3) \ & \ expr_3: 5\sin(x/2)\cos(y) + \cos(x/2)\sin(2y) \\
(\%i4) \ & \ plot3d ([expr_1, expr_2, expr_3], [x, -\pi, \pi], [y, -\pi, \pi], [grid, 40, 40])$
\end{align*}
\]

Plot of a spherical harmonic, using the predefined transformations, spherical_to_xyz, to transform from spherical 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])$

Use of the predefined function polar_to_xy to transform from cylindrical to rectangular coordinates. See the documentation for polar_to_xy. This example also shows how to eliminate the bounding box and the legend.

(%i1) plot3d (r^.33*cos(th/3), [r, 0, 1], [th, 0, 6*%pi],
[grid, 12, 80],
[transform_xy, polar_to_xy], [box, false],
[legend,false])$

Plot of a sphere using the transformation from spherical to rectangular coordinates. In xmaxima the three axes are scaled in the same proportion, maintaining the symmetric shape of the sphere. A palette with different shades of a single color is used:
Chapter 8: Plotting

(\%i1) plot3d ( 5, [theta, 0, %pi], [phi, 0, 2*%pi],
[plot_format,xmaxima],
[transform_xy, spherical_to_xyz],
[palette,[value,0.65,0.7,0.1,0.9]])$

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, 4, 4])$

apply: subscript must be an integer; found: round(x)
By setting the elevation equal to zero, a surface can be seen as a map in which each color represents a different level. The option `colorbox` is used to show the correspondence among colors and levels, and the mesh lines are disabled to make the colors easier to see.

```
(%i1) plot3d (cos (-x^2 + y^3/4), [x, -4, 4], [y, -4, 4],
    [mesh_lines_color, false], [elevation, 0], [azimuth, 0],
    [colorbox, true], [grid, 150, 150])$
```

See also the section about Plotting Options.

**plot_options**

System variable

Elements of this list state the default options for plotting. If an option is present in a `plot2d` or `plot3d` call, that value takes precedence over the default option. Otherwise, the value in `plot_options` is used. Default options are assigned by `set_plot_option`. There are other local options specific to each plotting command, and not included in this list of global options.

Each element of `plot_options` is a list of two or more items. The first item is the name of the option, and the remainder comprises the value or values assigned to the option. In some cases, the assigned value is a list, which may include several items.

See also `set_plot_option`, `get_option` and the section on Plotting Options.

**set_plot_option**(option)

Function

Accepts most of the options listed in the section Plotting Options, and saves them into the global variable `plot_options`.

`set_plot_option` evaluates its argument and returns the complete list `plot_options` (after modifying the option given).

See also `plot_options`, `get_option` and the section on Plotting Options.

Example:

Modification of the `grid` values.

```
(%i1) set_plot_option ([grid, 30, 40]);
```
spherical_to_xyz

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.

8.4 Plotting Options

All options consist of a list starting with one of the keywords in this section, followed by one or more values. Most of the options can be used in any of the plotting commands (`plot2d`, `plot3d`, `contour_plot`, `implicit_plot`) or in the function `set_plot_option`; the exceptions will be specified in the following list.

- **adapth_depth** [adapth_depth, integer]
  
  The maximum number of splittings used by the adaptive plotting routine.

  Default value: 5

- **axes** [axes, symbol]
  
  Where `symbol` can be either `true`, `false`, `x` or `y`. If `false`, no axes will be shown; if equal to `x` or `y` only the `x` or `y` axis will be shown, and if it is equal to `true`, both axes will be shown. This option is used only by `plot2d` and `implicit_plot`.

  Default value: `true`

- **azimuth** [azimuth, number]
  
  A `plot3d` plot can be thought of as starting with its x and y axis in the horizontal and vertical axis, as in `plot2d`, and the z axis coming out of the paper perpendicularly. The z axis is then rotated around the x axis an angle equals to `elevation` and then the xy plane is rotated around the new z axis an angle `azimuth`. This option sets the value for the azimuth, in degrees.

  Default value: 30

  See also `elevation`. 
box [box, symbol]
   Plot option
   If set to true, a bounding box will be drawn for the plot; if set to false, no box will be drawn.
   Default value: true

color [color, color_1, ..., color_n]
   Plot option
   In plot2d and implicit_plot, 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; one side of the surface will have color color_1 and the other color_2 (or the same color if there is only one color).
   If there are more curves or surfaces than colors, the colors will be repeated in sequence. When using gnuplot, the colors could be: blue, red, green, magenta, black, cyan or black; in xmaxima the colors can be those 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 given the name of an unknown color, black will be used instead.
   Default value: blue, red, green, magenta, black, cyan

colorbox [colorbox, symbol]
   Plot option
   Where symbol can be either true or false. If true, whenever plot3d uses a palette of different colors to represent the different values of z, a box will be shown on the right, indicating the colors used according to the scale of values of z. This option does not work in xmaxima.
   Default value: false

elevation [elevation, number]
   Plot option
   A plot3d plot can be thought of as starting with its x and y axis in the horizontal and vertical axis, as in plot2d, and the z axis coming out of the paper perpendicularly. The z axis is then rotated around the x axis an angle equals to elevation and then the xy plane is rotated around the new z axis an angle azimuth. This option sets the value for the elevation, in degrees.
   Default value: 60
   See also azimuth.

grid [grid, integer, integer]
   Plot option
   Sets the number of grid points to use in the x- and y-directions for three-dimensional plotting.
   Default value: 30, 30

legend [legend, string_1, ..., string_n]
   Plot option
   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. This option can not be set with set_plot_option.
**logx** \([\text{logx}]\)  
Plot option  
Makes the horizontal axes to be scaled logarithmically. It can not be used with \textit{set_plot_option}.

**logy** \([\text{logy}]\)  
Plot option  
Makes the vertical axes to be scaled logarithmically. It can not be used with \textit{set_plot_option}.

**mesh_lines_color** \([\text{mesh_lines_color, color}]\)  
Plot option  
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 \textit{color} (see the list of allowed colors in \textit{color}). It can also be given a value \textit{false} to eliminate completely the mesh lines.  
Default value: black

**nticks** \([\text{nticks, integer}]\)  
Plot option  
When plotting functions with plot2d, it is gives the initial number of points used by the adaptive plotting routine for plotting functions. When plotting parametric functions with plot2d or plot3d, it sets the number of points that will be shown for the plot. \textit{nticks} is used to determine whether \textit{plot} is plotted using \textit{plot2d} or \textit{plot3d}.  
Default value: 29

**palette** \([\text{palette}, [\text{palette}_1], ..., [\text{palette}_n]]\)  
Plot option  
It can consist of one palette or a list of several palettes. Each palette is a list with a keyword followed by four 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 \textit{mesh_lines_color}. If \textit{palette} is given the value \textit{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 \textit{color}.  
Default value: \([\text{hue, 0.25, 0.7, 0.8, 0.5}], \text{hue, 0.65, 0.8, 0.9, 0.55}], \text{hue, 0.55, 0.8, 0.9, 0.4}], \text{hue, 0.95, 0.7, 0.8, 0.5}]\)

**plot_format** \([\text{plot_format, format}]\)  
Plot option  
Where \textit{format} is one of the following: gnuplot, xmaxima, mgnuplot or gnuplot_pipes.  
It sets the format to be used for plotting.  
Default value: gnuplot, in Windows systems, or gnuplot_pipes in other systems.
plot_real_part [plot_realpart, symbol]  
Plot option  
When 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 \( \text{realpart}(\text{function}) \). I 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 \( \log(\text{abs}(x)) \); if plot_real_part were true, \( \log(-5) \) would be plotted as \( \log(5) \), while nothing would be plotted if plot_real_part were false.

Default value: 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 or lozenge

Default value: bullet, circle, plus, times, asterisk, box, square, triangle, delta, wedge, nabla, diamond, lozenge

psfile [psfile, string]  
Plot option  
Saves the plot into a Postscript file with name equal to string, rather than showing it in the screen. By default, the file will be created in the directory defined by the variable maxima_tempdir; the value of that variable can be changed to save the file in a different directory.

run_viewer [run_viewer, symbol]  
Plot option  
Controls whether or not the appropriate viewer for the plot format should be run.

Default value: true

style [style, type_1, ..., type_n]  
Plot option

style [style, [style_1], ..., [style_n]]  
Plot option

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:
Chapter 8: Plotting


*lines* accepts up to four parameters: line width, points radius, color and type of object to replace the points.

Default value: *lines* (will plot all sets of points joined with lines of thickness 1 and the first color given by the option *color*).

See also *color* and *point_type*.

**t** [t, min, max]  
Plot option  
Default range for parametric plots.  
Default value: -3, 3

**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.  
Default value: *false*  
See *make_transform*, *polar_to_xy* and *spherical_to_xyz*.

**x** [x, min, max]  
Plot option  
When used as the first option in a 2d-plotting 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*.

**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*.

**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.

8.5 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  
Plot option  
Sets the output terminal type for gnuplot.
- default (default value)  
  Gnuplot output is displayed in a separate graphical window.
- dumb  
  Gnuplot output is displayed in the Maxima console by an "ASCII art" approximation to graphics.
- 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.
- any other valid gnuplot term specification  
  Gnuplot can generate output in many other graphical formats such as png, jpeg, svg etc. To create plot in all these formats the gnuplot_term can be set to any supported gnuplot term name (symbol) or even full gnuplot term specification with any valid options (string). For example [gnuplot_term, png] creates output in PNG (Portable Network Graphics) format while [gnuplot_term, "png size 1000,1000"] creates PNG of 1000x1000 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  
Plot option  
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. If you want to create a Postscript file, you can use the option psfile instead, which will also work in Openmath, and does the same thing with just one option.

  [gnuplot_term, png], [gnuplot_out_file, "graph3.png"]

gnuplot_pm3d  
Plot option  
With a value of false, it can be used to prevent the usage of PM3D mode, which is enabled by default.


\textbf{gnuplot\_preamble} \hspace{1cm} Plot option
Inserts gnuplot commands before the plot is drawn. Any valid gnuplot commands may be used. Multiple commands should be separated with a semi-colon. The example shown produces a log scale plot. The default value for \texttt{gnuplot\_preamble} is the empty string "."

\textbf{gnuplot\_curve\_titles} \hspace{1cm} Plot option
This is an old option that has been replaced by \texttt{legend} described above.

\textbf{gnuplot\_curve\_styles} \hspace{1cm} Plot option
This is an obsolete option that has been replaced by \texttt{style}.

\textbf{gnuplot\_default\_term\_command} \hspace{1cm} Plot option
The gnuplot command to set the terminal type for the default terminal. The default value is \texttt{set term pop}.

\textbf{gnuplot\_dumb\_term\_command} \hspace{1cm} Plot option
The gnuplot command to set the terminal type for the dumb terminal. The default value is "\texttt{set term dumb 79 22}" , which makes the text output 79 characters by 22 characters.

\textbf{gnuplot\_ps\_term\_command} \hspace{1cm} Plot option
The gnuplot command to set the terminal type for the PostScript terminal. The default value is "\texttt{set size 1.5, 1.5; set term postscript eps enhanced color solid 24}" , which sets the size to 1.5 times gnuplot’s default, and the font size to 24, among other things. See the gnuplot documentation for \texttt{set term postscript} for more information.

\section*{8.6 Gnuplot\_pipes Format Functions}

\textbf{gnuplot\_start} () \hspace{1cm} Function
Opens the pipe to gnuplot used for plotting with the \texttt{gnuplot\_pipes} format. Is not necessary to manually open the pipe before plotting.

\textbf{gnuplot\_close} () \hspace{1cm} Function
Closes the pipe to gnuplot which is used with the \texttt{gnuplot\_pipes} format.

\textbf{gnuplot\_restart} () \hspace{1cm} Function
Closes the pipe to gnuplot which is used with the \texttt{gnuplot\_pipes} format and opens a new pipe.

\textbf{gnuplot\_replot} () \hspace{1cm} Function
\textbf{gnuplot\_replot} (s) \hspace{1cm} Function
Updates the gnuplot window. If \texttt{gnuplot\_replot} is called with a gnuplot command in a string \texttt{s}, then \texttt{s} is sent to gnuplot before reploting the window.
gnuplot_reset ()

Resets the state of gnuplot used with the gnuplot_pipes format. To update the gnuplot window call gnuplot_replot after gnuplot_reset.
9 Input and Output

9.1 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

9.2 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:

save, load, loadfile, stringout, batch, demo, writefile, closefile, and appendfile.

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.

9.3 Functions and Variables for Input and Output

-- 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:**

```maxima
(%i1) print ("I was called as", __);
   I was called as print(I was called as, __)
   (%o1) print(I was called as, __)
(%i2) foo (__);
   foo(foo(__))
   (%o2) foo(foo(__))
(%i3) g (x) := (print ("Current input expression =", __), 0);
   g(x) := (print("Current input expression =", __), 0)
   (%o3) g(x) := (print("Current input expression =", __), 0)
(%i4) [aa : 1, bb : 2, cc : 3];
   [1, 2, 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
```

`_` 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:**

```maxima
(%i1) 13 + 29;
   42
   (%o1) 42
(%i2) :lisp $_
   ((MPLUS) 13 29)
   (%o2) 42
(%i3) sin (%pi/2);
   1
   (%o3) 1
(%i4) :lisp $_
   ((%SIN) ((MQUOTIENT) $%PI 2))
   (%o4) _;
```
Chapter 9: Input and Output

```
(%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. For example,

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

yield the same result, namely 21/2.

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

```
block (block (a^n, %%*42), %%/6)
```
yields 7*a^n.

Within a compound statement, the value of %% may be inspected at a break prompt, which is opened by executing the break function. For example, at the break prompt opened by

```
block (a: 42, break ())$
```
entering %%; yields 42.

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 %.
Option variable

%edispflag
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/%e^x.

Example:

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

Function

%th (i)

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 useful in batch files or for referring to a group of output expressions. For example,

```
block (s: 0, for i:1 thru 10 do s: s + %th (i))$
```

sets s to the sum of the last ten output expressions.

%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 %.

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 3.1 [Lisp and Maxima], page 9 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.

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.

Option variable

absboxchar
Default value: !

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

Default value: false

The `file_output_append` variable 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.

**appendfile** *(filename)*

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** *(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 `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 writeln or appendfile.

**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)).

**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.

(%i11) y: 7$
(%i12) z: 88$
(%i13) concat (y, z/2);
(%o13) 744
(%i14) concat ('y, z/2);
(%o14) 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.

(%i15) a: concat ('y, z/2);
(%o15) y44
(%i16) a:: 123;
(%o16) 123
(%i17) y44;
(%o17) 123
(%i18) b^a;
(%o18) y44
(%o18) y
Chapter 9: Input and Output

(%i9) %, numer;  
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, ...)**
Concatenates its arguments into a string. Unlike concat, the arguments do not need to be atoms.

(%i11) sconcat ("xx[", 3, "]":", expand ((x+y)^3));
(%o11) xx[3]:y^3+3*x*y^2+3*x^2*y+x^3

**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.

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

123
2
x - x
.
8414709848078965
done

**dispcon (tensor\_1, tensor\_2, ...)**
Function
Displays the contraction properties of its arguments as were given to defcon. dispcon (all) displays all the contraction properties which were defined.

**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 disp.

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

\( \sin(1.0) \approx 0.8415 \)

\( \text{(%o3)} \text{ done} \)

**display2d**

Option variable

Default value: true

When `display2d` is false, the console display is a string (1-dimensional) form rather than a display (2-dimensional) form.

Example:

\( \text{(%i1)} \ x/(x^2+1); \)

\( \text{(%o1)} \ \frac{x}{2x+1} \)

\( \text{(%i2)} \ \text{display2d: false}$

\( \text{(%i3)} \ x/(x^2+1); \)

\( \text{(%o3)} \ x/(x^2+1) \)

**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:

<table>
<thead>
<tr>
<th>User</th>
<th>part</th>
<th>inpart</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-b;</td>
<td>a-b</td>
<td>a + (-1) b</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>- 1</td>
</tr>
<tr>
<td>a/b;</td>
<td>- a/b</td>
<td>a b</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>1/2</td>
</tr>
<tr>
<td>sqrt(x);</td>
<td>sqrt(x)</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>4 x</td>
<td>4</td>
</tr>
<tr>
<td>x*4/3;</td>
<td>--- x</td>
<td>- x</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**disptersms** (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 \( P_1, P_2, \ldots \) are very large expressions then the display program may run out of storage space in trying to display \( P_1 + P_2 + \ldots \) all at once.
However, \texttt{dispterms} (\texttt{P1 + P2 + \ldots}) displays \texttt{P1}, then below it \texttt{P2}, etc. When not using \texttt{dispterms}, if an exponential expression is too wide to be displayed as \texttt{A^B} it appears as \texttt{expt} (\texttt{A, B}) (or as \texttt{ncexpt} (\texttt{A, B}) in the case of \texttt{A^{^B}}).

\textbf{error\_size} \hspace{2cm} \text{Option variable}

Default value: 10

\texttt{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 \texttt{ERROR\_SIZE}) is greater than \texttt{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 \texttt{error\_syms}.

Otherwise, the expression is smaller than \texttt{error\_size}, and the expression is displayed in the message.

See also \texttt{error} and \texttt{error\_syms}.

Example:
The size of \texttt{U}, as determined by \texttt{ERROR\_SIZE}, is 24.

\begin{verbatim}
(%i1) U: (C^D^E + B + A)/(cos(X-1) + 1)$
(%i2) error\_size: 20$
(%i3) error ("Example expression is", U);
Example expression is errexp1
-- an error. Quitting. To debug this try debugmode(true);
(%i4) errexp1;
E
D
C + B + A
\end{verbatim}

\begin{verbatim}
(%o4)
--------------
cos(X - 1) + 1
(%i5) error\_size: 30$
(%i6) error ("Example expression is", U);
E
D
C + B + A
Example expression is --------------
\end{verbatim}

\begin{verbatim}
--------------
cos(X - 1) + 1
-- an error. Quitting. To debug this try debugmode(true);
\end{verbatim}

\textbf{error\_syms} \hspace{2cm} \text{Option variable}

Default value: \texttt{[errexp1, errexp2, errexp3]}

In error messages, expressions larger than \texttt{error\_size} are replaced by symbols, and the symbols are set to the expressions. The symbols are taken from the list \texttt{error\_syms}. The first too-large expression is replaced by \texttt{error\_syms[1]}, the second by \texttt{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`.

**expt** $(a, b)$

Function

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, e.g., $X^{-1}$ as $1/X$.

**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** $(filename)$

Function

`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,

```maxima
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.
file_search_maxima  
file_search_lisp  
file_search_demo  
file_search_usage  
file_search_tests

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.

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.

file_type (filename)  

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  

Default value: [1, lisp, 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]
file_type_maxima is a list of file extensions that maxima recognizes as denoting a Maxima source file.
See also file_type.

grind (expr)
Function
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.
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.
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]);
(\%o5)
[ \ ]
[ aa 17 ]
[ 29 bb ]
(\%i6) grind (%);
matrix([[aa,17],[29,bb]])$
(\%o6) done
(\%i7) set (aa, 17, 29, bb);
\begin{verbatim}
(%o7)    \{17, 29, aa, bb\}
(%i8) grind (%);
\{17,29,aa,bb\}$
(%o8) done
(%i9) exp (aa / (bb + 17)^29);
\[\frac{aa}{(bb + 17)^{29}}\]
(%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
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);
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)$
(%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));
\end{verbatim}
(\%16) \texttt{cholesky(A)} := \texttt{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)}

\textbf{ibase} \hfill \textbf{Option variable}

Default value: 10

\texttt{ibase} is the base for integers read by Maxima.

\texttt{ibase} may be assigned any integer between 2 and 36 (decimal), inclusive. When \texttt{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 \texttt{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 \texttt{ibase}, when an integer is terminated by a decimal point, it is interpreted in base 10.

See also \texttt{obase}.

Examples:

\texttt{ibase} less than 10.

\begin{verbatim}
(\%i1) ibase : 2 $
(\%i2) obase;
(\%o2) 10
(\%i3) 1111111111111111;
(\%o3) 65535
\end{verbatim}

\texttt{ibase} greater than 10. Letters are interpreted as digits only if the first digit is 0 through 9.

\begin{verbatim}
(\%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
\end{verbatim}

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

\begin{verbatim}
(\%i1) ibase : 36 $
(\%i2) obase;
(\%o2) 10
(\%i3) 1234;
(\%o3) 49360
\end{verbatim}
Option variable

Option variable

inchar

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.

(%i11) inchar: "input";
(%o11) input

(%i12) (input1) expand ((a+b)^3);

    3  2  2  3
(%o12) b + 3 a b + 3 a b + a

(%i13) ldisp (e, f);

    3
(%t13) (b + a)

    3  2  2  3
(%t14) b + 3 a b + 3 a b + a

ldisp (expr_1, ..., expr_n)

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.

(%i11) e: (a+b)^3;

(%o11) (b + a)

(%i12) f: expand (e);

    3  2  2  3
(%o12) b + 3 a b + 3 a b + a

(%i13) ldisp (e, f);

    3
(%t13) (b + a)

    3  2  2  3
(%t14) b + 3 a b + 3 a b + a

ldisplay (expr_1, ..., expr_n)

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 \texttt{ldisplay} and \texttt{rhs} is its value. Typically each argument is a variable. \texttt{ldisp} assigns an intermediate expression label to each equation and returns the list of labels.

See also \texttt{display}.

\begin{verbatim}
(%i1) e: (a+b)^3;
    3
(%o1) (b + a)

(%i2) f: expand (e);
    3 2 2 3
(%o2) b + 3 a b + 3 a b + a

(%i3) ldisplay (e, f);

    3
(%t3) e = (b + a)

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

(%o4) [%t3, %t4]

(%i4) %t3;

    3
(%o4) e = (b + a)

(%i5) %t4;

    3 2 2 3
(%o5) f = b + 3 a b + 3 a b + a
\end{verbatim}

\textbf{linechar} \hspace{1cm} Option variable

Default value: \%t

\texttt{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 \texttt{linechar} and \texttt{linenum}.

\texttt{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 \texttt{inchar}, \texttt{outchar}, and \texttt{linechar} should have a different first char. Otherwise some commands like \texttt{kill(inlables)} do not work as expected.

Intermediate expressions might or might not be displayed. See \texttt{programmode} and \texttt{labels}.

\textbf{linel} \hspace{1cm} Option variable

Default value: 79

\texttt{linel} is the assumed width (in characters) of the console display for the purpose of displaying expressions. \texttt{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 \texttt{describe}, is not affected by \texttt{linel}.

\textbf{lispdisp} \hspace{1cm} Option variable

Default value: \texttt{false}
When \texttt{lispdisp} is \texttt{true}, Lisp symbols are displayed with a leading question mark \texttt{?}. Otherwise, Lisp symbols are displayed with no leading mark.

Examples:

\begin{verbatim}
(%i1) lispdisp: false$
(%i2) ?foo + ?bar;
(%o2) foo + bar
(%i3) lispdisp: true$
(%i4) ?foo + ?bar;
(%o4) ?foo + ?bar
\end{verbatim}

\textbf{load (filename)}

Evaluates expressions in \texttt{filename}, thus bringing variables, functions, and other objects into Maxima. The binding of any existing object is clobbered by the binding recovered from \texttt{filename}. To find the file, \texttt{load} calls \texttt{file_search} with \texttt{file_search_maxima} and \texttt{file_search_lisp} as the search directories. If \texttt{load} succeeds, it returns the name of the file. Otherwise \texttt{load} prints an error message.

\texttt{load} works equally well for Lisp code and Maxima code. Files created by \texttt{save}, \texttt{translate_file}, and \texttt{compile_file}, which create Lisp code, and \texttt{stringout}, which creates Maxima code, can all be processed by \texttt{load}. \texttt{load} calls \texttt{loadfile} to load Lisp files and \texttt{batchload} to load Maxima files.

\texttt{load} does not recognize :lisp constructs in Maxima files, and while processing \texttt{filename}, the global variables \_, \__, \%, and \%th have whatever bindings they had when \texttt{load} was called.

See also \texttt{loadfile}, \texttt{batch}, \texttt{batchload}, and \texttt{demo}. \texttt{loadfile} processes Lisp files; \texttt{batch}, \texttt{batchload}, and \texttt{demo} process Maxima files.

See \texttt{file_search} for more detail about the file search mechanism.

\texttt{load} evaluates its argument.

\textbf{load_pathname} \hspace{1cm} \textbf{System variable}

Default value: \texttt{false}

When a file is loaded with the functions \texttt{load}, \texttt{loadfile} or \texttt{batchload} the system variable \texttt{load_pathname} is bound to the pathname of the file which is processed.

The variable \texttt{load_pathname} can be accessed from the file during the loading.

Example:

Suppose we have a batchfile \texttt{test.mac} in the directory

"/home/dieter/workspace/mymaxima/temp/" with the following commands

\begin{verbatim}
print("The value of load_pathname is: ", load_pathname)$
print("End of batchfile")$
\end{verbatim}

then we get the following output

\begin{verbatim}
(%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
\end{verbatim}
Function

loadfile (filename)

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.

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
(%i2) 2^8 - 1;
(%o10) 11111111
(%i3) obase : 8;
(%o3) 10
(%i4) 8^8 - 1;
(%o4) 77777777
(%i5) obase : 16;
(%o5) 10
(%i6) 16^8 - 1;
(%o6) 0FFFFFFFF
(%i7) obase : 36;
(%o7) 10
(%i8) 36^8 - 1;
(%o8) 0ZZZZZZZZ
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(inlables) do not work as expected.

(%i1) outchar: "output";
(output1) output
(%i2) expand ((a+b)^3);
(output2) b + 3 a b + 3 a^2 + a^3
(%i3)

See also labels.

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.

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

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

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.

(\%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)

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.

**tcl_output** (list, i0, skip)
**tcl_output** (list, i0)
**tcl_output** ([list1, ..., listn], 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 ([list1, ..., listn], i) prints the i’th elements of list1, ..., listn.

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 11.000000000}

**read** (expr1, ..., exprn)

Prints expr1, ..., exprn, 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.

(%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 \((expr_1, \ldots, expr_n)\)

Function

Prints \(expr_1, \ldots, expr_n\), then reads one expression from the console and returns the expression (without evaluation). The expression is terminated with a \(;\) (semicolon) or \($\) (dollar sign).

\[
(\%i1) \quad \text{aa: 7}\$
\]

\[
(\%i2) \quad \text{foo: readonly ("Enter an expression:";)}
\]

Enter an expression:
\[
2^\text{aa};
\]

\[
(\%o2) \quad \text{aa}
\]

\[
(\%o2) \quad 2
\]

\[
(\%i3) \quad \text{foo: read ("Enter an expression:";)}
\]

Enter an expression:
\[
2^\text{aa};
\]

\[
(\%o3) \quad 128
\]

See also read.

reveal \((expr, depth)\)

Function

Replaces parts of \(expr\) at the specified integer \(depth\) with descriptive summaries.

- Sums and differences are replaced by \(\text{Sum}(n)\) where \(n\) is the number of operands of the sum.
- Products are replaced by \(\text{Product}(n)\) where \(n\) is the number of operands of the product.
- Exponentials are replaced by \(\text{Expt}\).
- Quotients are replaced by \(\text{Quotient}\).
- Unary negation is replaced by \(\text{Negterm}\).
- Lists are replaced by \(\text{List}(n)\) where \(n\) is the number of elements of the list.

When \(depth\) is greater than or equal to the maximum depth of \(expr\), \(\text{reveal}\) \((expr, depth)\) returns \(expr\) unmodified.

\(\text{reveal}\) evaluates its arguments. \(\text{reveal}\) returns the summarized expression.

Example:

\[
(\%i1) \quad \text{e: expand ((a - b)^2)/expand ((exp(a) + exp(b))^2);}
\]

\[
\begin{align*}
2 & \quad 2 \\
\text{b - 2 a b + a} & \\
\text{b + a} & \quad 2 \quad \text{b} & \quad 2 \quad \text{a} \\
\quad 2 \quad \text{e} & + \quad \text{e} & + \quad \text{e}
\end{align*}
\]

\[
(\%i2) \quad \text{reveal (e, 1);}
\]

\[
(\%o2) \quad \text{Quotient}
\]

\[
(\%i3) \quad \text{reveal (e, 2);}
\]

\[
(\%o3) \quad \text{Sum}(3)
\]

\[
(\%o3) \quad \text{Sum}(3)
\]

\[
(\%i4) \quad \text{reveal (e, 3);}
\]

\[
\begin{align*}
\text{Expt + Negterm + Expt} & \\
\text{------------------------} & \\
\text{Product(2) + Expt + Expt}
\end{align*}
\]
(%i5) reveal (e, 4);
                               2 2
                2 b - Product(3) + a
(%o5) ---------------------------------------------------------------------
                Product(2)  Product(2)
                2 Expt + %e + %e
(%i6) reveal (e, 5);
                               2 2
                b - 2 a b + a
(%o6) ---------------------------------------------------
                Sum(2)  2 b  2 a
                2 %e + %e + %e
(%i7) reveal (e, 6);
                               2 2
                b - 2 a b + a
(%o7) ---------------------------------------------------
                b + a  2 b  2 a
                2 %e + %e + %e

rmxchar

    Default value: ]

rmxchar is the character drawn on the right-hand side of a matrix.
See also lmxchar.

save (filename, name_1, name_2, name_3, ...)
    Function
save (filename, values, functions, labels, ...)
    Function
save (filename, [m, n])
    Function
save (filename, name_1=expr_1, ...)
    Function
save (filename, all)
    Function
save (filename, name_1=expr_1, name_2=expr_2, ...)
    Function

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. The data stored by save may be
recovered by load (filename).

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 la-
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.

### 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.

### 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.

### showratvars (expr)

Function

Returns a list of the canonical rational expression (CRE) variables in expression `expr`. See also `ratvars`.

### stardisp

Option variable

Default value: `false`

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

### 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.

### 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."

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.
Function \texttt{tex} \((\text{expr})\)  
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.

\texttt{tex} \((\text{expr})\) prints a TeX representation of \texttt{expr} on the console.

\texttt{tex} \((\text{label})\) prints a TeX representation of the expression named by \texttt{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.

\texttt{destination} may be an output stream or file name. When \texttt{destination} is a file name, \texttt{tex} appends its output to the file. The functions \texttt{openw} and \texttt{opena} create output streams.

\texttt{tex} \((\text{expr}, \text{false})\) and \texttt{tex} \((\text{label}, \text{false})\) return their TeX output as a string.

\texttt{tex} evaluates its first argument after testing it to see if it is a label. Quote-quote \texttt{''} forces evaluation of the argument, thereby defeating the test and preventing the label.

See also \texttt{texput}.

Examples:

\begin{verbatim}
(%i1) integrate (1/(1+x^3), x);
\end{verbatim}

\begin{verbatim}
2 x - 1
log(x - x + 1) atan(-------)
2 sqrt(3)
\end{verbatim}

\begin{verbatim}
(%o1) - ------------------- + ------------- + --------------------
6 sqrt(3) 3
\end{verbatim}

\begin{verbatim}
(%i2) tex (%o1);\end{verbatim}

\begin{verbatim}
$$-{{\log \left(x^2-x+1\right)}\over{6}}+{{\arctan \left({{2\,x-1}\over{\sqrt{3}}}\right)}\over{\sqrt{3}}}+{{\log \left(x+1\right)}\over{3}}\leqno{\tt (%o1)}$$
\end{verbatim}

\begin{verbatim}
(%o2) (%o1)\end{verbatim}

\begin{verbatim}
(%i3) tex (integrate (sin(x), x));\end{verbatim}

\begin{verbatim}
$$-\cos x$$
\end{verbatim}

\begin{verbatim}
(%o3) false\end{verbatim}

\begin{verbatim}
(%i4) tex (%o1, "foo.tex");\end{verbatim}

\begin{verbatim}
(%o4) (%o1)\end{verbatim}

\texttt{tex} \((\text{expr}, \text{false})\) returns its TeX output as a string.

\begin{verbatim}
(%i1) S : tex (x * y * z, false);
\end{verbatim}

\begin{verbatim}
(%o1) $$x\,y\,z$$
\end{verbatim}

\begin{verbatim}
(%i2) S;
\end{verbatim}

\begin{verbatim}
(%o2) $$x\,y\,z$$
\end{verbatim}
Function \text{tex1}(e)

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:

\begin{verbatim}
(%i1) tex1 (\sin(x) + \cos(x));
(%o1) \sin x + \cos x
\end{verbatim}

Function \text{texput}(a, s)

Function \text{texput}(a, f)

Function \text{texput}(a, [s_1, s_2], \text{matchfix})

Function \text{texput}(a, [s_1, s_2, s_3], \text{matchfix})

Assign the TeX output for the atom \(a\), which can be a symbol or the name of an operator.

\text{texput}(a, s)\) causes the \text{tex} function to interpolate the string \(s\) into the TeX output in place of \(a\).

\text{texput}(a, f)\) causes the \text{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 \text{tex1} to generate TeX output for the arguments of the input expression.

\text{texput}(a, s, \text{operator \_ type})\), where \text{operator \_ type} is \text{prefix}, \text{infix}, \text{postfix}, \text{nary}, or \text{nofix}, causes the \text{tex} function to interpolate \(s\) into the TeX output in place of \(a\), and to place the interpolated text in the appropriate position.

\text{texput}(a, [s_1, s_2], \text{matchfix})\) causes the \text{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.

\text{texput}(a, [s_1, s_2, s_3], \text{matchfix})\) causes the \text{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.

\begin{verbatim}
(%i1) texput (me, "\\mu_e");
(%o1) \mu_e
(%i2) tex (me);
$$\mu_e$$
(%o2) false
\end{verbatim}

Assign TeX output for an ordinary function (not an operator).

\begin{verbatim}
(%i1) texput (lcm, "\\text{\texttt{lcm}}");
(%o1) \text{\texttt{lcm}}
(%i2) tex (lcm (a, b));
$$\text{\texttt{lcm}}(a, b)$$
(%o2) false
\end{verbatim}

Call a function to generate TeX output.
Assign TeX output for a prefix operator.

(%i1) prefix ("grad");
(%o1) grad
(%i2) texput ("grad", " \nabla ", prefix);
(%o2) \nabla
(%i3) tex (grad f);
$$\nabla f$$
(%o3) false

Assign TeX output for an infix operator.

(%i1) infix ("~");
(%o1) ~
(%i2) texput ("~", " \times ", infix);
(%o2) \times
(%i3) tex (a ~ b);
$$a \times b$$
(%o3) false

Assign TeX output for a postfix operator.

(%i1) postfix ("##");
(%o1) ##
(%i2) texput ("##", " !! ", postfix);
(%o2) !!
(%i3) tex (x ##);
$$x!!$$
(%o3) false

Assign TeX output for a nary operator.

(%i1) nary ("@@");
(%o1) @@
(%i2) texput ("@@", " \circ ", nary);
(%o2) \circ
(%i3) tex (a @@ b @@ c @@ d);
$$a \circ b \circ c \circ d$$
(%o3) false

Assign TeX output for a nofix operator.

(%i1) nofix ("foo");
(%o1) foo
(%i2) texput ("foo", "\mathsc{foo}", nofix);
(%o2) \mathsc{foo}
(%i3) tex (foo);
$$\mathsc{foo}$$
(%o3) false
Assign TeX output for a matchfix operator.

(%i1) matchfix ("<<", ">>");
(%o1) "<<
(%i2) texput ("<<", [" \langle ", " \rangle "], matchfix);
(%o2) [ \langle , \rangle ]
(%i3) tex (<<a>>);
$$ \langle a \rangle $$
(%o3) false
(%i4) tex (<<a, b>>);
$$ \langle a , b \rangle $$
(%o4) false
(%i5) texput ("<<", [" \langle ", " \rangle ", " \, | \,", matchfix);
(%o5) [ \langle , \rangle , \, | \,]
(%i6) tex (<<a>>);
$$ \langle a \rangle $$
(%o6) false
(%i7) tex (<<a, b>>);
$$ \langle a \, | \,b \rangle $$
(%o7) false

get_tex_environment (op)

set_tex_environment (op, before, after)

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) \text{tex} \ (f \ (x) \ := \ 1 - x);  \\
\$\$f(x):=1-x\$
(\%o4) \ \text{false}

get\_tex\_environment\_default () \quad \text{Function}

set\_tex\_environment\_default (before, after) \quad \text{Function}

Customize the TeX environment output by \text{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) \text{tex} \ (f(x) \ + \ g(x));  \\
\$g\left(x\right)+f\left(x\right)\$
(\%o2) \ \text{false}

(\%i3) set\_tex\_environment\_default ("\begin{equation}
", "
\end{equation}"");  \\
(\%o3) \[
\begin{equation}
\end{equation}

(\%i4) \text{tex} \ (f(x) \ + \ g(x));  \\
\begin{equation}
g\left(x\right)+f\left(x\right)\end{equation}
(\%o4) \ \text{false}

system (command) \quad \text{Function}

Executes \text{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; \ \text{rm} -f _hist.out)&")
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.

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)
  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 under-
lying 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.
10 Floating Point

10.1 Functions and Variables for Floating Point

**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.

**algepsilon**
Option variable
Default value: $10^{-8}$
algepsilon is used by algsys.

**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.

**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.

**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.

**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 $1.0000000000000000B0$. Otherwise, this is displayed as $1.0B0$. 
cbffac \((z, \text{fpprec})\)  
Complex bigfloat factorial.
load ("bffac") loads this function.

float \((\text{expr})\)  
Converts integers, rational numbers and bigfloats in expr to floating point numbers.
It is also an \texttt{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). The default value is true.

floatnump \((\text{expr})\)  
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 \texttt{bfloat} and \texttt{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. fpprintprec cannot be 1.

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},\text{numer\_pbranch:}\text{true}; \\
\%(o2) \ 1.189207115002721*\text{i}-1.189207115002721 \\

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

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

\%(i5) \ (-2)^{(3/4)},\text{numer,}\text{numer\_pbranch:}\text{true}; \\
\%(o5) \ 1.189207115002721*\text{i}-1.189207115002721
11 Contexts

11.1 Functions and Variables for Contexts

activate \( (\text{context}_1, \ldots, \text{context}_n) \)

Function

Activates the contexts \( \text{context}_1, \ldots, \text{context}_n \). The facts in these contexts are then available to make deductions and retrieve information. The facts in these contexts are not listed by \text{facts} () .

The variable \text{activecontexts} is the list of contexts which are active by way of the \text{activate} function.

activecontexts

System variable

Default value: \[
\]

\text{activecontexts} is a list of the contexts which are active by way of the \text{activate} function, as opposed to being active because they are subcontexts of the current context.

assume \( (\text{pred}_1, \ldots, \text{pred}_n) \)

Function

Adds predicates \( \text{pred}_1, \ldots, \text{pred}_n \) 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 \text{assume}.

\text{assume} returns a list whose elements are the predicates added to the context or the atoms \text{redundant} or \text{inconsistent} where applicable.

The predicates \( \text{pred}_1, \ldots, \text{pred}_n \) can only be expressions with the relational operators \( < \leq \text{equal} \not= \geq \) and \( > \). Predicates cannot be literal equality \( = \) or literal inequality \( \# \) expressions, nor can they be predicate functions such as \text{integerp}.

Compound predicates of the form \text{pred}_1 \text{and} \ldots \text{and} \text{pred}_n \text{are recognized, but not} \text{pred}_1 \text{or} \ldots \text{or} \text{pred}_n. \text{not} \text{pred}_k \text{is recognized if} \text{pred}_k \text{is a relational predicate. Expressions of the form} \text{not} (\text{pred}_1 \text{and} \text{pred}_2) \text{and} \text{not} (\text{pred}_1 \text{or} \text{pred}_2) \text{are not recognized.}

Maxima’s deduction mechanism is not very strong; there are many obvious consequences which cannot be determined by \text{is}. This is a known weakness.

\text{assume} does not handle predicates with complex numbers. If a predicate contains a complex number \text{assume} returns \text{inconsistent} or \text{redundant}.

\text{assume} evaluates its arguments.

See also \text{is}, \text{facts}, \text{forget}, \text{context}, and \text{declare}.

Examples:

\begin{verbatim}
(\%i1) assume (xx > 0, yy < -1, zz >= 0);
(\%o1) \[xx > 0, - 1 > yy, zz >= 0\]
(\%i2) assume (aa < bb and bb < cc);
(\%o2) \[bb > aa, cc > bb\]
(\%i3) facts ();
(\%o3) \[xx > 0, - 1 > yy, zz >= 0, bb > aa, cc > bb\]
\end{verbatim}
\( (%i4) \text{is (xx > yy);} \) 
\( (%o4) \text{true} \)

\( (%i5) \text{is (yy < -yy);} \)
\( (%o5) \text{true} \)

\( (%i6) \text{is (sinh (bb - aa) > 0);} \)
\( (%o6) \text{true} \)

\( (%i7) \text{forget (bb > aa);} \)
\( (%o7) [bb > aa] \)

\( (%i8) \text{prederror : false;} \)
\( (%o8) \text{false} \)

\( (%i9) \text{is (sinh (bb - aa) > 0);} \)
\( (%o9) \text{unknown} \)

\( (%i10) \text{is (bb^2 < cc^2);} \)
\( (%o10) \text{unknown} \)

**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 \([\text{expr, 2 expr, 3 expr}]\) 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 \( \textit{symbolp}(x) \) or \( \textit{subvarp}(x) \).

See also **assume** and **assume_pos**.

Examples:

- (%i11) assume_pos: true$
  (%i12) assume_pos_pred: symbolp$
  (%i13) sign (a);
  (%o13) pos
  (%i14) sign (a[1]);
  (%o14) pnz
  (%i15) assume_pos_pred: lambda ([x], display (x), true)$
  (%i16) asksign (a); x = a

- (%o6) pos
  (%i17) asksign (a[1]);
    x = a
      1

- (%o7) pos
  (%i18) asksign (foo (a));
    x = foo(a)

- (%o8) pos
  (%i19) asksign (foo (a) + bar (b));
    x = foo(a)
    x = 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?

\[ p; \]
\[ (%o11) \text{ neg} \]

**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 (item)

Function

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.

features

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, commutative, lassociative, rassociative, symmetric, and antisymmetric, 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.

forget (pred_1, ..., pred_n)

Function

forget (L)

Function

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.

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 unkill 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.
Function \texttt{newcontext} \hspace{1em} \texttt{(name)}

Creates a new, empty context, called \texttt{name}, which has \texttt{global} as its only subcontext. The newly-created context becomes the currently active context. \texttt{newcontext} evaluates its argument. \texttt{newcontext} returns \texttt{name}.

Function \texttt{supcontext} \hspace{1em} \texttt{(name, context)}

Function \texttt{supcontext} \hspace{1em} \texttt{(name)}

Creates a new context, called \texttt{name}, which has \texttt{context} as a subcontext. \texttt{context} must exist.

If \texttt{context} is not specified, the current context is assumed.
12 Polynomials

12.1 Introduction to Polynomials

Polynomials are stored in Maxima either in General Form or as Cannonical 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 3*X^2-1 is (X 2 3 0 -1) and that of 2*X*Y+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.

12.2 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.

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 (expr, x, n)

coeff (expr, x)

Function

Returns the coefficient of x^n in expr, where expr is a polynomial or a monomial term in x.

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.

Examples:

coeff returns the coefficient x^n in expr.

(%i1) coeff (b^3*a^3 + b^2*a^2 + b*a + 1, a^3);

3
  b

(%o1)

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);

3
  - c

(%o1)

(%i2) coeff (c[4]*z^4 - c[3]*z^3 - c[2]*z^2 + c[1]*z, z^3);

3
  - c

(%o2)

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);
   3    3
(%o1)   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
(%o1) - 2 %pi

(%i2) coeff (v[1]^4 - 2*%pi*v[1]^2 + 1, v[1], 2);
(\%o2)
(\%o2) - 2 %pi

(%i3) coeff (sin(1+x)*sin(x) + sin(1+x)^3*sin(x)^3, sin(1+x)^3);
   3
(%o3) sin(x)

(%i4) coeff ((d - a)^2*(b + c)^3 + (a + b)^4*(c - d), a + b, 4);
   %o4
(%o4) c - d

coeff itself does not apply expand or factor or any other function.

(%i1) coeff (c*(a + b)^3, a);
   %o1
(%o1) 0

(%i2) expand (c*(a + b)^3);
   %o2
(%o2) 3 b c + 3 a b c + 3 a b c + a c

(%i3) coeff (%, a);
   %o3
(%o3) 3 b c

(%i4) coeff (b^3*c + 3*a*b^2*c + 3*a^2*b*c + a^3*c, (a + b)^3);
   %o4
(%o4) 0

(%i5) factor (b^3*c + 3*a*b^2*c + 3*a^2*b*c + a^3*c);
   %o5
(%o5) (b + a) c

(%i6) coeff (%, (a + b)^3);
   %o6
(%o6) c
coeff distributes over lists, matrices, and equations.

(%i1) coeff ([4*a, -3*a, 2*a], a);
   %o1
(%o1) [4, - 3, 2]

(%i2) coeff (matrix ([a*x, b*x], [-c*x, -d*x]), x);
   %o2
(%o2) [a b]
(\-% c - d)

(%i3) coeff (a*u - b*v = 7*u + 3*v, u);
   %o3
(%o3) a = 7

\textbf{combine} (expr)     \textbf{Function}
Simplifies the sum expr by combining terms with the same denominator into a single
term.

\textbf{content} (p_1, x_1, ..., x_n)     \textbf{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]

Function
denom (expr)
Returns the denominator of the rational expression expr.

divide (p_1, p_2, x_1, ..., x_n)
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.

Function
eliminate ([eqn_1, ..., eqn_n], [x_1, ..., x_k])
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 \( x_1, ..., x_k \) 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 \( x_1, ..., x_k \). 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^2 + x - y^2 + 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]

Function
ezgcd (p_1, p_2, p_3, ...)
Returns a list whose first element is the g.c.d of the polynomials \( p_1, p_2, p_3, ... \) and whose remaining elements are the polynomials divided by the g.c.d. This always uses the ezgcd algorithm.
facexpand
Option variable

Default value: true

facexpand controls whether the irreducible factors returned by factor are in expanded (the default) or recursive (normal CRE) form.

factcomb (expr)
Function
Tries to combine the coefficients of factorials in expr with the factorials themselves by converting, for example, \((n + 1)\cdot n!\) into \((n + 1)!\).

sumsplitfact if set to false will cause minfactorial to be applied after a factcomb.

factor (expr)
Function
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.

Examples:

\[
\begin{align*}
(\text{o1}) \text{factor} \ (2^{63} \ - \ 1); \\
& 2 \\
(\text{o2}) \text{factor} \ (-8y - 4x + z^2*(2*y + x)); \\
& (2 \ y \ + \ x) \ (z - 2) \ (z + 2) \\
(\text{o3}) \ -1 - 2*x - x^2 + y^2 + 2*y*x + x*y^2 + x^2*y^2; \\
& 2 \ 2 \ 2 \ 2 \\
(\text{o4}) \text{block} (\text{[dontfactor: [x]]}, \text{factor} \ (%/36/(1 + 2*y + y^2))); \\
& (x + 2 \ x + 1) \ (y - 1) \\
(\text{o4}) \ -\ ------------------ \\
& 36 \ (y + 1)
\end{align*}
\]
(%i5) factor (1 + %e^(3*x));
   x  2 x  x
(%o5) (%e + 1) (%e - %e + 1)
(%i6) factor (1 + x^4, a^2 - 2);
   2  2
(%o6) (x - a x + 1) (x + a x + 1)
(%i7) factor (-y^2*z^2 - x*z^2 + x^2*y^2 + x^3);
   2
(%o7) - (y + x) (z - x) (z + x)
(%i8) (2 + x)/(3 + x)/(b + x)/(c + x)^2;
   x + 2
(%o8) ------------------------
       2
       (x + 3) (x + b) (x + c)
(%i9) ratsimp (%);
   4  3
(%o9) (x + 2)/(x + (2 c + b + 3) x
   2  2
+ (c + (2 b + 6) c + 3 b) x + ((b + 3) c + 6 b c) x + 3 b c )
(%i10) partfrac (%); %, x;
   2  4  3
(%o10) -(c - 4 c - b + 6)/((c + (-2 b - 6) c
   2  2
+ (b + 12 b + 9) c + (-6 b - 18 b) c + 9 b) (x + c))
   c - 2
- -----------------------------
   2
   (c + (-b - 3) c + 3 b) (x + c)
   b - 2
+ ---------------------------------------
   2
   2
   3  2
   ((b - 3) c + (6 b - 2 b) c + b - 3 b) (x + b)
   1
- ---------------------------------------
   2
   ((b - 3) c + (18 - 6 b) c + 9 b - 27) (x + 3)
(%i11) map ('factor, %);
\[
\frac{2}{c - 4} - \frac{c - b + 6}{(c - 3)(c - b)(c + c)} + \frac{c - 2}{(c - 3)(c - b)(c + c)}
\]

\[
\frac{b - 2}{(b - 3)(c - b)(x + c)} + \frac{1}{(b - 3)(c - b)(x + c)}
\]

\[
\texttt{ratsimp ((x^5 - 1)/(x - 1));}
\]

\[
\frac{4}{x + x + x + x + 1}
\]

\[
\texttt{subst (a, x, %);}
\]

\[
\frac{4}{a + a + a + a + 1}
\]

\[
\texttt{factor (%th(2), %);} \quad 2 \quad 3 \quad 3 \quad 2
\]

\[
\texttt{(x - a)(x - a)(x + a + a + a + 1)}
\]

\[
\texttt{factor (1 + x^12);} \quad 4 \quad 8 \quad 4
\]

\[
\texttt{(x + 1)(x - x + 1)}
\]

\[
\texttt{factor (1 + x^99);} \quad 2 \quad 6 \quad 3
\]

\[
\texttt{(x + 1)(x - x + 1)(x - x + 1)}
\]

\[
(x - x + x - x + x - x + x - x + x - x + x - x + 1)
\]

\[
(x + x - x - x + x + x - x - x + x + x - x - x + x + x - x)
\]

\[
(x + x - x - x + x + x - x - x - x - x + x + x - x)
\]

\[
30 \quad 27 \quad 21 \quad 18 \quad 12 \quad 9 \quad 3
\]

\[
- x - x + x + x - x - x + x + x + 1)
\]

\textbf{factorflag} \quad \text{Option variable}

Default value: \texttt{false}

When \texttt{factorflag} is \texttt{false}, suppresses the factoring of integer factors of rational expressions.

\textbf{factorout (expr, x_1, x_2, ...)} \quad \text{Function}

Rearranges the sum \texttt{expr} into a sum of terms of the form \( f(x_1, x_2, \ldots) \cdot g \) where \( g \) is a product of expressions not containing any \( x_i \) and \( f \) is factored.

Note that the option variable \texttt{keepfloat} is ignored by \texttt{factorout}.

Example:
(%i1) expand (a*(x+1)*(x-1)*(u+1)^2);
   \[ a u x^2 + 2 a u x - a u - 2 a u - a \]
(%o1) \[ a u x^2 + 2 a u x - a u - 2 a u - a \]
(%i2) factorout(%o1,x);
   \[ 2 \]
(%o2) \[ a u (x - 1) (x + 1) + 2 a u (x - 1) (x + 1) + a (x - 1) (x + 1) \]

**factorsum** (expr)

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));
   \[ a x z^2 + a z + 2 a w x z + 2 a w z + a w x + v x^2 \]
   \[ + 2 u v x + u x + a w + v + 2 u v + u \]
(%o1) \[ a x z^2 + a z + 2 a w x z + 2 a w z + a w x + v x^2 \]
   \[ + 2 u v x + u x + a w + v + 2 u v + u \]
(%i2) factorsum (%);
   \[ 2 \]
(%o2) \[ (x + 1) (a (z + w) + (v + u)) \]

**fasttimes** (p_1, p_2)

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 \(\text{max}(n_1, n_2)^{1.585}\).

**fullratsimp** (expr)

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);
   \[ \frac{a/2^2 a/2^2}{x-1} \]
(%o1) \[ \frac{a/2^2 a/2^2}{x-1} \]
\%(i2)\ \texttt{ratsimp (expr);} \\
\frac{2 \ a \ a}{x \ - \ 2 \ x \ + \ 1} \\
\frac{a}{a \ x \ - \ 1} \\
\%(o2) \ \\
\%(i3) \ \texttt{fullratsimp (expr);} \\
a \\
\%(o3) \\
\%(i4) \ \texttt{rat (expr);} \\
\frac{a/2 \ 4 \ a/2 \ 2}{(x \ ) \ - \ 2 \ (x \ ) \ + \ 1} \\
\frac{a}{a \ x \ - \ 1} \\
\%(o4) \\
\textbf{fullratsubst} \ (a, b, c) \ \\
\begin{itemize}
\item \texttt{fullratsubst} \ is \ the \ same \ as \ \texttt{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. \\
\item \texttt{fullratsubst} \ will \ also \ accept \ its \ arguments \ in \ the \ format \ of \ \texttt{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. \\
\item \texttt{load ("lrats")} \ loads \ \texttt{fullratsubst} \ and \ \texttt{lratsubst}. \\
\end{itemize} 

\begin{itemize}
\item \texttt{fullratsubst} \ is \ equivalent \ to \ \texttt{ratsubst} \ except \ that \ it \ recurses \ until \ its \ result \ stops \ changing. \\
\item \texttt{fullratsubst} \ also \ accepts \ a \ list \ of \ equations \ or \ a \ single \ equation \ as \ first \ argument. \\
\end{itemize}
fullratsubst ([a^2 = b, b^2 = c, c^2 = a], a^3*b*c);

fullratsubst (a^2 = b*a, a^3);

fullratsubst may cause an indefinite recursion.

errcatch (fullratsubst (b*a^2, a^2, a^3));

--- Lisp stack overflow. RESET

\section*{gcd \textit{(p1, p2, x1, ...)}}

Returns the greatest common divisor of \textit{p1} and \textit{p2}. The flag \texttt{gcd} determines which algorithm is employed. Setting \texttt{gcd to ez, subres, red, or spmod} selects the \texttt{ezgcd, subresultant prs, reduced, or modular} algorithm, respectively. If \texttt{gcd false} then \texttt{gcd} \texttt{(p1, p2, x)} always returns 1 for all \textit{x}. Many functions (e.g. \texttt{ratsimp, factor, etc.}) cause \texttt{gcd}'s to be taken implicitly. For homogeneous polynomials it is recommended that \texttt{gcd equal to subres} be used. To take the \texttt{gcd} when an algebraic is present, e.g., \texttt{gcd (x^2 - 2*sqrt(2)*x + 2, x - sqrt(2))}, \texttt{algebraic} must be \texttt{true} and \texttt{gcd} must not be \texttt{ez}.

The \texttt{gcd} flag, default: \texttt{spmod}, if \texttt{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 gcdis are not required.

\section*{gcdex \textit{(f, g)}}

\section*{gcdex \textit{(f, g, x)}}

Returns a list \([a, b, u]\) where \textit{u} is the greatest common divisor (gcd) of \textit{f} and \textit{g}, and \textit{u} is equal to \(a f + b g\). The arguments \textit{f} and \textit{g} should be univariate polynomials, or else polynomials in \textit{x} a supplied main variable since we need to be in a principal ideal domain for this to work. The \texttt{gcd} means the \texttt{gcd} regarding \textit{f} and \textit{g} as univariate polynomials with coefficients being rational functions in the other variables.

\texttt{gcdex} implements the Euclidean algorithm, where we have a sequence of \texttt{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 \texttt{L[i+2]}: \texttt{L[i] - q L[i+1]}, and it terminates at \texttt{L[i+1]} when the remainder \texttt{r[i+2]} is zero.

\begin{verbatim}
(%i1) gcdex (x^2 + 1, x^3 + 4);
\end{verbatim}

\begin{verbatim}
2
\%o1)/R/ x + 4 x - 1 x + 4
[- ------------, -----, 1]
17 17
\%o2)/R/ . [x^2 + 1, x^3 + 4, -1];
\%o2)/R/ 0
\end{verbatim}

Note that the \texttt{gcd} in the following is 1 since we work in \(k(y)[x]\), not the \(y+1\) we would expect in \(k[y, x]\).

\begin{verbatim}
(%i1) gcdex (x*(y + 1), y^2 - 1, x);
\end{verbatim}
Function \texttt{gcfactor} ($n$)

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.

Example:

\begin{verbatim}
(%i1) gcfactor (x^4 - 1);
(%o1) (x - 1) (x + 1) (x - \%i) (x + \%i)
\end{verbatim}

Function \texttt{gfactor} ($\text{expr}$)

Factors the polynomial $\text{expr}$ over the Gaussian integers (that is, the integers with the imaginary unit $\%i$ adjoined). This is like \texttt{factor ($\text{expr}, a^2+1$)} where $a$ is $\%i$.

Example:

\begin{verbatim}
(%i1) gfactor (x^4 - 1);
(%o1) (x - 1) (x + 1) (x - \%i) (x + \%i)
\end{verbatim}

Function \texttt{gfactorsum} ($\text{expr}$)

is similar to \texttt{factorsum} but applies \texttt{gfactor} instead of \texttt{factor}.

Function \texttt{hipow} ($\text{expr}$, $x$)

Returns the highest explicit exponent of $x$ in $\text{expr}$. $x$ may be a variable or a general expression. If $x$ does not appear in $\text{expr}$, \texttt{hipow} returns 0.

\texttt{hipow} does not consider expressions equivalent to $\text{expr}$. In particular, \texttt{hipow} does not expand $\text{expr}$, so \texttt{hipow ($\text{expr}$, $x$)} and \texttt{hipow (expand ($\text{expr}$, $x$))} may yield different results.

Examples:

\begin{verbatim}
(%i1) hipow (y^3 * x^2 + x * y^4, x);
(%o1) 2
(%i2) hipow ((x + y)^5, x);
(%o2) 1
(%i3) hipow (expand ((x + y)^5), x);
(%o3) 5
(%i4) hipow ((x + y)^5, x + y);
(%o4) 5
(%i5) hipow (expand ((x + y)^5), x + y);
(%o5) 0
\end{verbatim}

Option variable \texttt{intfaclim}

Default value: \texttt{true}

If \texttt{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 \texttt{intfaclim} is \texttt{false} (this is the case when the user calls \texttt{factor} explicitly), complete factorization will be attempted. \texttt{intfaclim} is set to \texttt{false} when factors are computed in \texttt{divisors}, \texttt{divsum} and \texttt{totient}.

Internal calls to \texttt{factor} respect the user-specified value of \texttt{intfaclim}. Setting \texttt{intfaclim} to \texttt{true} may reduce the time spent factoring large integers.
keepfloat

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:

(\%)1rats(x/2.0);

'rat' replaced 0.5 by 1/2 = 0.5

(\%)1R/ x

(\%)2rats(x/2.0), keepfloat;

(\%)2R/ 0.5 x

solve ignores keepfloat:

(\%)3solve(1.0-x,x), keepfloat;

'rat' replaced 1.0 by 1/1 = 1.0

(\%)3[ x = 1]

lratsubst \((L, expr)\)

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:

(\%)1load("lrats")$

• subst can carry out multiple substitutions. lratsubst is analogous to subst.

(\%)2subst \([a = b, c = d], a + c\);

(\%)2d + b

(\%)3lratsubst \([a^2 = b, c^2 = d], (a + e)\cdot c\cdot (a + c)\);

(\%)3(d + a c) e + a d + b c

• If only one substitution is desired, then a single equation may be given as first argument.

(\%)4lratsubst \((a^2 = b, a^3)\);

(\%)4a b

modulus

Default value: false

When modulus is a positive number \(p\), operations on rational numbers (as returned by rat and related functions) are carried out modulo \(p\), using the so-called "balanced"
modulus system in which \( n \mod p \) is defined as an integer \( k \) in \([-((p-1)/2), \ldots, 0, \ldots, (p-1)/2]\) when \( p \) is odd, or \([-((p/2-1)/2), \ldots, 0, \ldots, p/2]\) when \( p \) is even, such that \( a p + k \) equals \( n \) for some integer \( a \).

If \( \text{expr} \) is already in canonical rational expression (CRE) form when \( \text{modulus} \) is reset, then you may need to re-rat \( \text{expr} \), e.g., \( \text{expr: rat (ratdisrep (expr))} \), in order to get correct results.

Typically \( \text{modulus} \) is set to a prime number. If \( \text{modulus} \) is set to a positive non-prime integer, this setting is accepted, but a warning message is displayed. Maxima will allow zero or a negative integer to be assigned to \( \text{modulus} \), although it is not clear if that has any useful consequences.

\[\textbf{num} \ (\text{expr})\]

Function

Returns the numerator of \( \text{expr} \) if it is a ratio. If \( \text{expr} \) is not a ratio, \( \text{expr} \) is returned.

\[\text{num} \ \text{evaluates its argument.}\]

\[\textbf{polydecomp} \ (p, x)\]

Function

Decomposes the polynomial \( p \) in the variable \( x \) into the functional composition of polynomials in \( x \). \( \text{polydecomp} \) returns a list \([p_1, \ldots, p_n] \) such that

\[\lambda([x], p_1) (\lambda([x], p_2) (\ldots (\lambda([x], p_n) (x)) \ldots))\]

is equal to \( p \). The degree of \( p_i \) is greater than 1 for \( i \) less than \( n \).

Such a decomposition is not unique.

Examples:

\[
\begin{align*}
\text{(%i1) polydecomp (x^210, x);} \\
\ & \begin{bmatrix} 7 & 5 & 3 & 2 \end{bmatrix} \\
\text{(%o1) [x, x, x, x]} \\
\text{(%i2) p : expand (subst (x^3 - x - 1, x, x^2 - a));} \\
\ & \begin{bmatrix} 6 & 4 & 3 & 2 \end{bmatrix} \\
\text{(%o2) x - 2 x - 2 x + x + 2 x - a + 1} \\
\text{(%i3) polydecomp (p, x);} \\
\ & \begin{bmatrix} 2 & 3 \end{bmatrix} \\
\text{(%o3) [x - a, x - x - 1]} \\
\end{align*}
\]

The following function composes \( L = [e_1, \ldots, e_n] \) as functions in \( x \); it is the inverse of \( \text{polydecomp} \):

\[
\text{compose} (L, x) := \text{block ([r : x], for e in L do r : subst (e, x, r), r)} \$
\]

Re-express above example using \( \text{compose} \):

\[
\begin{align*}
\text{(%i3) polydecomp (compose ([x^2 - a, x^3 - x - 1], x), x);} \\
\ & \begin{bmatrix} 2 & 3 \end{bmatrix} \\
\text{(%o3) [x - a, x - x - 1]} \\
\end{align*}
\]

Note that though \( \text{compose (polydecomp (p, x), x) \ always \ returns \ p \ (unexpanded), \ polydecomp (compose ([p_1, \ldots, p_n], x), x) \ does \ not \ necessarily \ return \ [p_1, \ldots, p_n]} \)
(%i4) polydecomp (compose ([x^2 + 2*x + 3, x^2], x), x);
   2  2
(%o4) [x + 2, x + 1]

(%i5) polydecomp (compose ([x^2 + x + 1, x^2 + x + 1], x), x);
   2  2
   x + 3  x + 5
(%o5) [--, --, 2 x + 1]
   4   2

quotient (p_1, p_2) Function
quotient (p_1, p_2, x_1, ..., x_n) Function

Returns the polynomial p_1 divided by the polynomial p_2. The arguments x_1, ..., x_n are interpreted as in ratvars.

quotient returns the first element of the two-element list returned by divide.

rat (expr) Function
rat (expr, x_1, ..., x_n) Function

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, ..., 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), 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:

(%i11) ((x - 2*y)^4/(x^2 - 4*y^2)^2 + 1)*(y + a)*(2*y + x) /
   (4*y^2 + x^2);
\[ \frac{4}{(x - 2y)(y + a)(2y + x)\left(\frac{-4y + 1}{x - 4y}\right)} \]

\[ \frac{2}{4y + x} \]

\( \text{rat} \) (%, y, a, x);
\[ \frac{2a + 2y}{x + 2y} \]

\textbf{ratalgdenom}

Option variable

Default value: true

When \texttt{ratalgdenom} is true, allows rationalization of denominators with respect to radicals to take effect. \texttt{ratalgdenom} has an effect only when canonical rational expressions (CRE) are used in algebraic mode.

\textbf{ratcoef}\ (\texttt{expr, x, n})
Function

\textbf{ratcoef}\ (\texttt{expr, x})
Function

Returns the coefficient of the expression \( x^n \) in the expression \( \text{expr} \). 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.

\texttt{ratcoef} expands and rationally simplifies its first argument and thus it may produce answers different from those of \texttt{coeff} which is purely syntactic. Thus \texttt{ratcoef \ ((x + 1)/y + x, x)} returns \((y + 1)/y\) whereas \texttt{coeff} returns 1.

\texttt{ratcoef}\ (\texttt{expr, x, 0}), viewing \texttt{expr} as a sum, returns a sum of those terms which do not contain \( x \). Therefore if \( x \) occurs to any negative powers, \texttt{ratcoef} should not be used.

Since \texttt{expr} is rationally simplified before it is examined, coefficients may not appear quite the way they were envisioned.

Example:

\begin{verbatim}
(%i1) s: a*x + b*x + 5$
(%i2) ratcoef (s, a + b);
(%o2) \[ x \]
\end{verbatim}

\textbf{ratdenom}\ (\texttt{expr})
Function

Returns the denominator of \( \text{expr} \), after coercing \texttt{expr} to a canonical rational expression (CRE). The return value is a CRE.

\texttt{expr} is coerced to a CRE by \texttt{rat} if it is not already a CRE. This conversion may change the form of \texttt{expr} by putting all terms over a common denominator.

\texttt{denom} is similar, but returns an ordinary expression instead of a CRE. Also, \texttt{denom} does not attempt to place all terms over a common denominator, and thus some expressions which are considered ratios by \texttt{ratdenom} are not considered ratios by \texttt{denom}. 

\section{Chapter 12: Polynomials}
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);
   2
   x + x + 1
   --------
   2
   y + 7

(%i2) ratdenomdivide: true$

(%i3) ratexpand (expr);
   2
   x x 1
   ------- + ------- + -------
   2 2 2
   y + 7 y + 7 y + 7

(%i4) ratdenomdivide: false$%

(%i5) ratexpand (expr);
   2
   x x + 1
   --------
   2
   y + 7

(%i6) expr2: a^2/(b^2 + 3) + b/(b^2 + 3);
   2
   b a
   ------- + -------
   2 2
   b + 3 b + 3

(%i7) ratexpand (expr2);
   2
   b + a
   -------
   2
   b + 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:

```lisp
(%i1) expr: (4*x^3 + 10*x - 11)/(x^5 + 5);
3
 4 x + 10 x - 11
 5
---
 x + 5

(%i2) ratdiff (expr, x);
7 5 4 2
 8 x + 40 x - 55 x - 60 x - 50
---
10 5
 x + 10 x + 25

(%i3) expr: f(x)^3 - f(x)^2 + 7;
3 2
f (x) - f (x) + 7

(%i4) ratdiff (expr, f(x));
2
3 f (x) - 2 f(x)

(%i5) expr: (a + b)^3 + (a + b)^2;
3 2
(b + a) + (b + a)

(%i6) ratdiff (expr, a + b);
2
3 b + (6 a + 2) b + 3 a + 2 a
```

**Function ratdisrep**( `expr` )

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`.

**Option variable ratepsilon**

Default value: 2.0e-8

`ratepsilon` is the tolerance used in the conversion of floating point numbers to rational numbers.

**Function ratexpand**( `expr` )

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:

```lisp
(%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) ratexpand (expr);
     2
(%o3) --------- + ---------
     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 `rat ((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 \texttt{ratfac} and \texttt{ratweight} schemes are incompatible and may not both be used at the same time.

\textbf{\texttt{ratnumerator (expr)}} \\
Returns the numerator of \texttt{expr}, after coercing \texttt{expr} to a canonical rational expression (CRE). The return value is a CRE.

\texttt{expr} is coerced to a CRE by \texttt{rat} if it is not already a CRE. This conversion may change the form of \texttt{expr} by putting all terms over a common denominator.

\texttt{num} is similar, but returns an ordinary expression instead of a CRE. Also, \texttt{num} does not attempt to place all terms over a common denominator, and thus some expressions which are considered ratios by \texttt{ratnumerator} are not considered ratios by \texttt{num}.

\textbf{\texttt{ratnump (expr)}} \\
Returns \texttt{true} if \texttt{expr} is a literal integer or ratio of literal integers, otherwise \texttt{false}.

\textbf{\texttt{ratp (expr)}} \\
Returns \texttt{true} if \texttt{expr} is a canonical rational expression (CRE) or extended CRE, otherwise \texttt{false}.

CRE are created by \texttt{rat} and related functions. Extended CRE are created by \texttt{taylor} and related functions.

\textbf{\texttt{ratprint}} \\
Default value: \texttt{true}

When \texttt{ratprint} is \texttt{true}, a message informing the user of the conversion of floating point numbers to rational numbers is displayed.

\textbf{\texttt{ratsimp (expr)}} \\
\textbf{\texttt{ratsimp (expr, x_{1}, \ldots, x_{n})}} \\
Simplifies the expression \texttt{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., \texttt{sin (x^2 + 1)}) and the arguments to any such functions are also rationally simplified.

\textbf{\texttt{ratsimp (expr, x_{1}, \ldots, x_{n})}} enables rational simplification with the specification of variable ordering as in \texttt{ratvars}.

When \texttt{ratsimpexpons} is \texttt{true}, \texttt{ratsimp} is applied to the exponents of expressions during simplification.

See also \texttt{ratexpand}. Note that \texttt{ratsimp} is affected by some of the flags which affect \texttt{ratexpand}.

Examples:

\begin{verbatim}
(%i1) sin (x/(x^2 + x)) = exp ((log(x) + 1)^2 - log(x)^2); 2 2
\hspace{2cm} x (log(x) + 1) - log (x) 2
(%o1) sin(------) = %e 2
\hspace{2cm} x + x
\end{verbatim}
(%i2) ratsimp (%);

(%o2) \frac{1}{\sqrt{x + 1}} = e^{\frac{2}{x + 1}}

(%i3) ((x - 1)^{3/2} - (x + 1)*sqrt(x - 1))/sqrt((x - 1)*(x + 1));

(%o3) \frac{(x - 1)^{3/2} - \sqrt{x - 1} \cdot \sqrt{x + 1}}{\sqrt{(x - 1) \cdot (x + 1)}}

(%i4) ratsimp (%);

(%o4) \frac{2 \cdot \sqrt{x - 1} - \sqrt{x + 1}}{\sqrt{x + 1}}

(%i5) x^{a + 1/a}, ratsimpexpons: true;

(%o5) x^{\frac{2}{a + 1}}

ratsimpexpons

Option variable

Default value: false

When ratsimpexpons is true, ratsimp is applied to the exponents of expressions during simplification.

ratsubst (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 a + z.

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);

(%o1) a x y^3 + a x^4 y^8

(%i2) ratsubst (1 - sin(x)^2, cos(x)^2, cos(x)^2 + cos(x) + 1);

(%o2) cos(x) + cos(x) + cos(x) + 1

(%i3) ratsubst (1 - sin(x)^2, cos(x)^2, %);

(%o3) \frac{\sin(x) - 3 \cdot \sin(x) + \cos(x) \cdot (2 - \sin(x)) + 3}{\sin(x)^2 - \cos(x)^2}

(%i4) ratsubst (1 - cos(x)^2, sin(x)^2, sin(x)^2);

(%o4) \frac{2}{\sin(x)^2 - \cos(x)^2}
Chapter 12: Polynomials

\[
\begin{align*}
\text{(i4)} & \quad \cos^4(x) - 2 \cos^2(x) + 1 \\
\text{(i5)} & \quad \text{radsubstflag: false}\$
\text{(i6)} & \quad \text{ratsubst}(u, \sqrt{x}, x); \\
\text{(i7)} & \quad \text{radsubstflag: true}\$
\text{(i8)} & \quad \text{ratsubst}(u, \sqrt{x}, x);
\end{align*}
\]

Function \text{ratvars}(x_1, \ldots, x_n)

Declares main variables \(x_1, \ldots, x_n\) 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 \text{ratvars} list, it is given a lower priority than \(x_1\).

The arguments to \text{ratvars} can be either variables or non-rational functions such as \(\sin(x)\).

The variable \text{ratvars} is a list of the arguments of the function \text{ratvars} when it was called most recently. Each call to the function \text{ratvars} resets the list. \text{ratvars()} clears the list.

Option variable \text{ratvarswitch}

Default value: \text{true}

Maxima keeps an internal list in the Lisp variable \text{VARLIST} of the main variables for rational expressions. If \text{ratvarswitch} is \text{true}, every evaluation starts with a fresh list \text{VARLIST}. This is the default behavior. Otherwise, the main variables from previous evaluations are not removed from the internal list \text{VARLIST}.

The main variables, which are declared with the function \text{ratvars} are not affected by the option variable \text{ratvarswitch}.

Examples:

If \text{ratvarswitch} is \text{true}, every evaluation starts with a fresh list \text{VARLIST}.

\[
\begin{align*}
\text{(i1)} & \quad \text{ratvarswitch: true}\$
\text{(i2)} & \quad \text{rat}(2+x+y^-2); \\
\text{(i3)} & \quad \text{:lisp varlist} \quad (\$X \quad \$Y) \\
\text{(i4)} & \quad \text{rat}(2+a+b^-2);
\end{align*}
\]

\%o3/R/ \quad 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);

2
(%o5)/R/ y + 2 x
(%i6) :lisp varlist
($X $Y)
2
(%o6)/R/ b + 2 a
(%i7) :lisp varlist
($A $B $X $Y)
Function
ratweight
(x_1, w_1, ..., x_n, w_n)
Function
ratweight ()
Assigns a weight w_i to the variable x_i. This causes a term to be replaced by 0 if its weight exceeds the value of the variable ratwtlvl (default yields no truncation). The weight of a term is the sum of the products of the weight of a variable in the term times its power. For example, the weight of 3 x_1^2 x_2 is 2 w_1 + w_2. Truncation according to ratwtlvl is carried out only when multiplying or exponentiating canonical rational expressions (CRE).
ratweight () returns the cumulative list of weight assignments.
Note: The ratfac and ratweight schemes are incompatible and may not both be used at the same time.
Examples:
(%i1) ratweight (a, 1, b, 1);
(%o1) [a, 1, b, 1]
(%i2) expr1: rat(a + b + 1)$
(%i3) expr1^2;
2 2
(%o3)/R/ b + (2 a + 2) b + a + 2 a + 1
(%i4) ratwtlvl: 1$
(%i5) expr1^2;
(%o5)/R/ 2 b + 2 a + 1
ratweights
System variable
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.
kil (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 (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, ..., 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)
resultant

Variable

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 variable resultant controls which algorithm will be used to compute the resultant. subres for subresultant prs, mod for modular resultant algorithm, and red for reduced prs. On most problems subres should be best. On some large degree univariate or bivariate problems mod may be better.

The function bezout takes the same arguments as resultant and returns a matrix. The determinant of the return value is the desired resultant.

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.

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);
                                 2  2
                       2
(%o1) (2 x + 1) (x - 1)
Function `tellrat (p<sub>1</sub>, ..., p<sub>n</sub>)`

Adds to the ring of algebraic integers known to Maxima the elements which are the solutions of the polynomials `p<sub>1</sub>`, ..., `p<sub>n</sub>`. Each argument `p<sub>i</sub>` is a polynomial with integer coefficients.

Function `tellrat ()`

Effective 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<sup>2</sup> - y<sup>2</sup>)`, there would be an ambiguity as to whether to substitute `y<sup>2</sup>` for `x<sup>2</sup>` or vice versa. Maxima picks a particular ordering, but if the user wants to specify which, e.g. `tellrat (y<sup>2</sup> = x<sup>2</sup>)` provides a syntax which says replace `y<sup>2</sup>` by `x<sup>2</sup>`.

Examples:

```
(%i1) 10*(%i + 1)/(%i + 3^(1/3));

(%o1) 10 (%i + 1)
     1/3
     %i + 3

(%i2) ev (ratdisrep (rat(%)), algebraic);

2/3 1/3 2/3 1/3
(%o2) (4 3 - 2 3 - 4) %i + 2 3 + 4 3 - 2

(%i3) tellrat (1 + a + a<sup>2</sup>);

(%o3) [a + a + 1]

(%i4) 1/(a*sqrt(2) - 1) + a/(sqrt(3) + sqrt(2));

1 a
(%o4) ------------------- + -------------------
       sqrt(2) a - 1 sqrt(3) + sqrt(2)

(%i5) ev (ratdisrep (rat(%)), algebraic);

(7 sqrt(3) - 10 sqrt(2) + 2 a) - 2 sqrt(2) - 1
(%o5) ----------------------------------

(%i6) tellrat (y<sup>2</sup> = x<sup>2</sup>);

(%o6) [y - x , a + a + 1]
```

Function `totaldisrep (expr)`

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, ..., x_n)

Removes tellrat properties from x_1, ..., x_n.
13 Constants

13.1 Functions and Variables for Constants

%e

%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

%i represents the imaginary unit, sqrt(-1).

false

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

ind

ind represents a bounded, indefinite result. See also limit.

Example:

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

inf

inf represents real positive infinity.

infinity

infinity represents complex infinity.

minf

minf represents real minus (i.e., negative) infinity.

%phi

%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) \phi^n - (1 - \phi) / (2 \phi - 1)

(%i2) fib (n-1) + fib (n) - fib (n+1);

(%o2) - fib(n + 1) + fib(n) + fib(n - 1)

(%i3) fibtophi (%);

(%o3) \phi^n - (1 - \phi) / (2 \phi - 1) + \phi^{n-1} - (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));

(%o1) %phi^2 A - %phi^2 A - %phi A + %phi - %phi - 1

(%i2) ratsimp (e);

(%o2) (%phi - %phi - 1) A + %phi - %phi - 1

(%i3) tellrat (%phi^2 - %phi - 1);

(%o3) [\phi^2 - \phi - 1]

(%i4) algebraic : true;

(%o4) true

(%i5) ratsimp (e);

(%o5) 0

%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 represents the Boolean constant of the same name. Maxima implements true by the value T in Lisp.

und represents an undefined result.

See also limit.

Example:
zeroa

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

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

See also zeroa and limit.
14 Logarithms

14.1 Functions and Variables for Logarithms

\%e_to_numlog

Option variable

Default value: false

When true, \( r \) some rational number, and \( x \) some expression, \( %e^{-r\log(x)} \) will be simplified into \( x^r \). It should be noted that the \texttt{radcan} command also does this transformation, and more complicated transformations of this ilk as well. The \texttt{logcontract} command "contracts" expressions containing \texttt{log}.

\textbf{li} \[ s \] \( (z) \)

Function

Represents the polylogarithm function of order \( s \) and argument \( z \), defined by the infinite series

\[
\text{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}
\]

\( \text{li} \ [1] \) is \( - \log (1 - z) \). \( \text{li} \ [2] \) and \( \text{li} \ [3] \) 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 \texttt{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 \texttt{numer} evaluation flag is present.

Examples:

(\%i1) assume (x > 0);
(\%o1) \[ x > 0 \]
(\%i2) integrate ((log (1 - t)) / t, t, 0, x);
(\%o2) \[ - \text{li} (x) \]
(\%i3) \text{li} \ [2] \ (7);
(\%o3) \[ \text{li} (7) \]
(\%i4) \text{li} \ [2] \ (7), \text{numer};
(\%o4) \[ 1.2482731783392 - 6.113257021832577 \ %i \]
(\%i5) \text{li} \ [3] \ (7);
(\%o5) \[ \text{li} (7) \]
(\%i6) \text{li} \ [2] \ (7), \text{numer};
(\%o6) \[ 1.2482731783392 - 6.113257021832577 \ %i \]
(\%i7) L : makelist (i / 4.0, i, 0, 8);
(\%o7) \[ 0.0, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0 \]
(\%i8) \text{map} (\text{lambda} ([x], \text{li} \ [2] \ (x)), L);
\( \text{log} \ (x) \) 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. \( \text{log10}(x) := \text{log}(x) / \text{log}(10) \) 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, \( \log(a^b) \) 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 \( a/b \), \( a\neq1 \). (\( \log(1/b) \), 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.

- **lognumber**
  - if true then negative floating point arguments to \( \log \) will always be converted to their absolute value before the \( \log \) is taken. If numer is also true, then negative integer arguments to \( \log \) will also be converted to their absolute value.

- **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 \( \%e^{r \cdot \log(x)} \) will be simplified into \( x^r \). It should be noted that the \( \text{radcan} \) command also does this transformation, and more complicated transformations of this as well. The \( \text{logcontract} \) command "contracts" expressions containing \( \log \).

**logabs** Option variable

Default value: false

When doing indefinite integration where logs are generated, e.g. \( \text{integrate}(1/x, x) \), the answer is given in terms of \( \log(\text{abs}(\ldots)) \) if \( \text{logabs} \) is true, but in terms of...
\(\log(\ldots)\) if \texttt{logabs} is \texttt{false}. For definite integration, the \texttt{logabs:true} setting is used, because here "evaluation" of the indefinite integral at the endpoints is often needed.

\textbf{logarc} \hspace{1cm} \textbf{Option variable}

\textbf{logarc (expr)} \hspace{1cm} \textbf{Function}

When the global variable \texttt{logarc} is \texttt{true}, inverse circular and hyperbolic functions are replaced by equivalent logarithmic functions. The default value of \texttt{logarc} is \texttt{false}.

The function \texttt{logarc(expr)} carries out that replacement for an expression \texttt{expr} without setting the global variable \texttt{logarc}.

\textbf{logconcoeffp} \hspace{1cm} \textbf{Option variable}

Default value: \texttt{false}

Controls which coefficients are contracted when using \texttt{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 \texttt{logconcoeffp:'logconfun$ logconfun(m):=featurep(m,integer) or ratnump(m)$}. Then \texttt{logcontract(1/2*log(x))}; will give \texttt{log(sqrt(x))}.

\textbf{logcontract (expr)} \hspace{1cm} \textbf{Function}

Recursively scans the expression \texttt{expr}, transforming subexpressions of the form \(a1*\log(b1) + a2*\log(b2) + c\) into \(\log(\text{ratsimp}(b1^{a1} * b2^{a2})) + c\)

\begin{verbatim}
(%i1) 2*(a*log(x) + 2*a*log(y))$
(%i2) logcontract(%);
   2 4
(%o2) a log(x y )
\end{verbatim}

The declaration \texttt{declare(n,integer)} causes \texttt{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 \texttt{featurep(coeff,integer)}. The user can control which coefficients are contracted by setting the option \texttt{logconcoeffp} to the name of a predicate function of one argument. E.g. if you like to generate SQRTs, you can do \texttt{logconcoeffp:'logconfun$ logconfun(m):=featurep(m,integer) or ratnump(m)$}. Then \texttt{logcontract(1/2*log(x))}; will give \texttt{log(sqrt(x))}.

\textbf{logexpand} \hspace{1cm} \textbf{Option variable}

Default value: \texttt{false}

If \texttt{true} causes \(\log(a^b)\) to become \(b*\log(a)\). If it is set to \texttt{all}, \(\log(a*b)\) will also simplify to \(\log(a)+\log(b)\). If it is set to \texttt{super}, then \(\log(a/b)\) will also simplify to \(\log(a)-\log(b)\) for rational numbers \(a/b, a\#1\). (\(\log(1/b)\), for integer \(b\), always simplifies.) If it is set to \texttt{false}, that is the default value, all of these simplifications will be turned off.

\textbf{lognegint} \hspace{1cm} \textbf{Option variable}

Default value: \texttt{false}

If \texttt{true} implements the rule \(\log(-n) \rightarrow \log(n)+\%i*\%pi\) for \(n\) a positive integer.
lognumeral

Option variable
Default value: false
If true then negative floating point arguments to log will always be converted to their absolute value before the log is taken. If numer is also true, then negative integer arguments to log will also be converted to their absolute value.

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 < carg(x) <= +%pi.
Chapter 15: Trigonometric

15 Trigonometric

15.1 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: \( \text{acos} \), \( \text{acosh} \), \( \text{acot} \), \( \text{acoth} \), \( \text{acsc} \), \( \text{acsch} \), \( \text{asec} \), \( \text{asech} \), \( \text{asin} \), \( \text{asinh} \), \( \text{atan} \), \( \text{atanh} \), \( \text{cos} \), \( \text{cosh} \), \( \text{cot} \), \( \text{coth} \), \( \text{csc} \), \( \text{csch} \), \( \text{sec} \), \( \text{sech} \), \( \text{sin} \), \( \text{sinh} \), \( \text{tan} \), and \( \text{tanh} \). There are a number of commands especially for handling trigonometric functions, see \texttt{trigexpand}, \texttt{trigreduce}, and the switch \texttt{trigsign}. Two share packages extend the simplification rules built into Maxima, \texttt{ntrig} and \texttt{atrig1}. Do \texttt{describe(command)} for details.

15.2 Functions and Variables for Trigonometric

\%piargs

Option variable

Default value: \texttt{true}

When \%piargs is \texttt{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:

\begin{verbatim}
(%i1) %piargs : false$
(%i2) [sin(\%pi), sin(\%pi/2), sin(\%pi/3)];
(%o2) [sin(\%pi), sin(---), sin(---)]
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, -------]
2
(%i6) [sin(\%pi/4), sin(\%pi/5), sin(\%pi/6)];
(%o6) [-------, sin(---), -]
5 2
sqrt(2)
(%i7) [cos(\%pi/3), cos(10*\%pi/3), tan(10*\%pi/3), cos(sqrt(2)*\%pi/3)];
(%o7) [-, -
2
sqrt(3), cos(-------)]
3

Some identities are applied when \( \pi \) and \( \pi/2 \) are multiplied by an integer variable.
\end{verbatim}
(%i1) declare (n, integer, m, even)
(%i2) [sin (%pi * n), cos (%pi * m), sin (%pi/2 * m),
       cos (%pi/2 * m)];
          m/2
(%o2) [0, 1, 0, (- 1) ]

%iargs

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)];
       [sin( %i x), cos( %i x), tan( %i x)]
(%i3) %iargs : true$
(%i4) [sin (%i * x), cos (%i * x), tan (%i * x)];
       [%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)];
       [true, false]
(%i3) sin (%i * x);
       %i sinh(x)

acos (x)
   - Arc Cosine.

acosh (x)
   - Hyperbolic Arc Cosine.

acot (x)
   - Arc Cotangent.

acoth (x)
   - Hyperbolic Arc Cotangent.

acs (x)
   - Arc Cosecant.

acsch (x)
   - Hyperbolic Arc Cosecant.

asec (x)
   - Arc Secant.
asech \( (x) \)  
- Hyperbolic Arc Secant.

asin \( (x) \)  
- Arc Sine.

asinh \( (x) \)  
- Hyperbolic Arc Sine.

atan \( (x) \)  
- Arc Tangent.

atan2 \( (y, x) \)  
- yields the value of \( \tan(y/x) \) in the interval \(-\pi\) to \(\pi\).

atanh \( (x) \)  
- Hyperbolic Arc Tangent.

atrig1  
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 \texttt{load(atrig1)}; to use them.

cos \( (x) \)  
- Cosine.

cosh \( (x) \)  
- Hyperbolic Cosine.

cot \( (x) \)  
- Cotangent.

coth \( (x) \)  
- Hyperbolic Cotangent.

csc \( (x) \)  
- Cosecant.

csch \( (x) \)  
- 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:

\[
\frac{\text{realpart}(z)}{\text{floor}(-\frac{\text{imagpart}(z)}{2 \pi})}
\]

\[
(1 - \text{unit_step}(\text{imagpart}(z)))
\]

\[
\frac{\text{realpart}(z)}{\text{floor}(-\frac{\text{imagpart}(z)}{2 \pi})} - \frac{\text{realpart}(z)}{\text{ceiling}(-\frac{\text{imagpart}(z)}{2 \pi})}
\]

\[
((-1) + 1)
\]

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:

\[
\begin{align*}
(\%i1) \text{halfangles}\text{false;} & \quad \text{false} \\
(\%i2) \sin(x/2); & \quad \sin(-) \\
(\%i3) \text{halfangles}\text{true;} & \quad \text{true} \\
(\%i4) \sin(x/2); & \quad \frac{\sqrt{1 - \cos(x)}}{\sqrt{2}} \\
(\%i5) \text{assume(x>0, x<2*%pi)} & \\
(\%i6) \sin(x/2); & \quad \frac{\sqrt{1 - \cos(x)}}{\sqrt{2}}
\end{align*}
\]
SECTION 15: TRIGONOMETRIC

The `ntrig` package contains a set of simplification rules that are used to simplify trigonometric functions whose arguments are of the form $f(n \times \pi/10)$ 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. $\sin(2 \times x)$) 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));
   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(2*x)) 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 (expr, x)
Function
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);
     1       1
(%o1) --------- + 3 (--------- + -) + x - -
     2       2

trigsignt
Option variable
Default value: true
When trigsign is true, it permits simplification of negative arguments to trigono-
metric 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 \( \frac{\pi}{n} \) (\( n \) 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.

\[
\text{(i1)} \quad \text{trigrat}(\sin(3a)/\sin(a+\pi/3));
\]
\[
\text{(o1)} \quad \text{sqrt}(3) \sin(2a) + \cos(2a) - 1
\]

The following example is taken from Davenport, Siret, and Tournier, *Calcul Formel*, Masson (or in English, Addison-Wesley), section 1.5.5, Morley theorem.

\[
\text{(i1)} \quad c : \frac{\pi}{3} - a - b$
\]
\[
\text{(i2)} \quad bc : \sin(a)*\sin(3c)/\sin(a+b);
\]
\[
\text{(o2)} \quad \frac{\sin(a) \sin(3(-b-a+\frac{\pi}{3}))}{\sin(b+a)}
\]

\[
\text{(i3)} \quad ba : bc, c=a, a=c;
\]
\[
\text{(o3)} \quad \frac{\sin(3a) \sin(b+a-\frac{\pi}{3})}{\sin(a-\frac{\pi}{3})}
\]

\[
\text{(i4)} \quad ac2 : ba^-2 + bc^-2 - 2*bc*ba*cos(b);
\]
\[
\text{(o4)} \quad \frac{\sin(3a) \sin(b+a-\frac{\pi}{3})}{\sin(a-\frac{\pi}{3})} + \frac{\sin(b+a-\frac{\pi}{3})/(\sin(a-\frac{\pi}{3}) \sin(b+a))}{\sin(a-\frac{\pi}{3})} + \frac{\sin(a) \sin(3(-b-a+\frac{\pi}{3}))}{\sin(b+a)}
\]
(%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
16 Special Functions

16.1 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()` Gamma function
- `gammagreek(a, z)` Incomplete gamma function
- `gammaincomplete(a, z)` Tail of incomplete gamma function
- `hypergeometric(l1, l2, z)` Hypergeometric function
- `slommel` 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

16.2 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)**

The Bessel function of the second kind of order \( v \) and argument \( z \).

\( \texttt{bessel\_y} \) 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)**

The modified Bessel function of the first kind of order \( v \) and argument \( z \).

\( \texttt{bessel\_i} \) 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)**

The modified Bessel function of the second kind of order \( v \) and argument \( z \).

\( \texttt{bessel\_k} \) 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)**

The Hankel function of the first kind of order \( v \) and argument \( z \) (A&S 9.1.3).

\( \texttt{hankel\_1} \) is defined as

\[ \texttt{bessel\_j}(v,z) + \%i \ast \texttt{bessel\_y}(v,z) \]

Maxima evaluates \( \texttt{hankel\_1} \) numerically for a real order \( v \) and complex argument \( z \) in float precision. The numerical evaluation in bigfloat precision and for a complex order \( v \) is not supported.

When \( \texttt{besselexpand} \) is \texttt{true}, \( \texttt{hankel\_1} \) is expanded in terms of elementary functions when the order \( v \) is half of an odd integer. See \( \texttt{besselexpand} \).

Maxima knows the derivative of \( \texttt{hankel\_1} \) wrt the argument \( z \).

Examples:

Numerical evaluation:
(%i1) hankel_1(1,0.5);
(%o1) 0.2422684576748738 - 1.471472392670243 %i
(%i2) hankel_1(1,0.5+%i);
(%o2) -0.2558287994862166 %i - 0.239575601883016

A complex order \( v \) is not supported. Maxima returns a noun form:

(%i3) hankel_1(%i,0.5+%i);
(%o3) hankel_1(%i, %i + 0.5)

Expansion of \( \text{hankel}_1 \) when \( \text{besslexpand} \) is \( \text{true} \):

(%i4) hankel_1(1/2,z),besslexpand:true;
(%o4) \frac{\sqrt{2} \sin(z) - \sqrt{2} %i \cos(z)}{\sqrt{\pi} \sqrt{z}}

Derivative of \( \text{hankel}_1 \) wrt the argument \( z \). The derivative wrt the order \( v \) is not supported. Maxima returns a noun form:

(%i5) diff(hankel_1(v,z),z);
(%o5) \frac{\text{hankel}_1(v - 1, z) - \text{hankel}_1(v + 1, z)}{2}
(%i6) diff(hankel_1(v,z),v);
(%o6) \frac{d}{dv} (\text{hankel}_1(v, z))

Function \( \text{hankel}_2 \)

The Hankel function of the second kind of order \( v \) and argument \( z \) (A&S 9.1.4).

\( \text{hankel}_2 \) is defined as

\[ \text{bessel}_j(v,z) - %i \ast \text{bessel}_y(v,z) \]

Maxima evaluates \( \text{hankel}_2 \) numerically for a real order \( v \) and complex argument \( z \) in float precision. The numerical evaluation in bigfloat precision and for a complex order \( v \) is not supported.

When \( \text{besslexpand} \) is \( \text{true} \), \( \text{hankel}_2 \) is expanded in terms of elementary functions when the order \( v \) is half of an odd integer. See \( \text{besslexpand} \).

Maxima knows the derivative of \( \text{hankel}_2 \) wrt the argument \( z \). For examples see \( \text{hankel}_1 \).

Option variable \( \text{besslexpand} \)

Default value: \( \text{false} \)

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 \( \text{besslexpand} \) is \( \text{true} \), the Bessel function is expanded.
(%i3) besselexpand: true$
(%i4) bessel_j (3/2, z);
    sin(z)   cos(z)
  sqrt(2) sqrt(z) (-- -------)
  2   z
-----------

(%o4) ---------------------------------

sqrt(%pi)

Function

scaled_bessel_i (v, z)
The scaled modified Bessel function of the first kind of order $v$ and argument $z$. That is, $scaled\_bessel\_i(v, z) = \exp(-\text{abs}(z)) \times bessel\_i(v, z)$. 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 $\exp(-\text{abs}(z)) \times bessel\_i(v, z)$.

scaled_bessel_i0 (z)
Identical to $scaled\_bessel\_i(0, z)$.

scaled_bessel_i1 (z)
Identical to $scaled\_bessel\_i(1, z)$.

%s [u,v] (z)
Lommel's little $s[u,v](z)$ function. Probably Gradshteyn & Ryzhik 8.570.1.

16.3 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)
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)
The derivative of the Airy function $Ai$ $airy\_ai(x)$.
See $airy\_ai$.

airy_bi (x)
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 \( (x) \)

The derivative of the Airy Bi function \( \text{airy} \_\text{bi}(x) \).
See \text{airy}_\text{ai} \ and \text{airy}_\text{bi}.

16.4 Gamma and factorial Functions

The gamma function and the related beta, psi and incomplete gamma functions are defined in Abramowitz and Stegun, \textit{Handbook of Mathematical Functions}, Chapter 6.

gamma \( (z) \)

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 \text{factlim}. For integers greater than \text{factlim} the numerical result of the factorial function, which is used to calculate gamma, will overflow. The simplification for rational numbers is controlled by \text{gammalim} to avoid internal overflow. See \text{factlim} and \text{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:

\begin{verbatim}
(%i1) map('gamma,[1,2,3,4,5,6,7,8,9]);
(%o1) [1, 1, 2, 6, 24, 120, 720, 5040, 40320]
(%i2) map('gamma,[1/2,3/2,5/2,7/2]);
[ sqrt(\pi), 3 sqrt(\pi), 15 sqrt(\pi) ]
(%o2) [sqrt(\pi), --------, -----------, -----------]
     2          4          8
(%i3) map('gamma,[2/3,5/3,7/3]);
2 1
[ gamma(-), 4 gamma(-) ]
  2
(%o3) [ gamma(-), -----------, -----------]
     3          3

Numerical evaluation for real and complex values:

(%i4) map('gamma,[2.5,2.5b0]);
(%o4) [1.329340388179137, 1.3293403881791370205b0]
\end{verbatim}
(%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 \( \gamma(z+n) \) and \( \gamma(z-n) \), when \( \gamma_{\text{expand}} \) 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\}
    z - 1

The derivative of \( \gamma \):
(%i10) diff(gamma(z),z); %gamma
(%o10) \psi (z) \gamma(z)

0
See also \( \text{makegamma} \).

The Euler-Mascheroni constant is %gamma.

log_gamma  \( (z) \)  
Function

The natural logarithm of the gamma function.

gamma_incomplete  \( (a, z) \)  
Function

The incomplete upper gamma function A&S 6.5.2:
\[
\Gamma(a, z) = \int_z^\infty t^{a-1} e^{-t} dt
\]

gamma_incomplete_regularized  \( (a, z) \)  
Function

The regularized incomplete upper gamma function A&S 6.5.1:
\[
Q(a, z) = \frac{\Gamma(a, z)}{\Gamma(a)}
\]

gamma_incomplete_generalized  \( (a, z1, z2) \)  
Function

The generalized incomplete gamma function.
\[
\Gamma(a, z1, z2) = \int_{z1}^{z2} t^{a-1} e^{-t} dt
\]

gammalim  
Option variable

Default value: 1000000

\( \text{gammalim} \) controls simplification of the gamma function for integral and rational number arguments. If the absolute value of the argument is not greater than \( \text{gammalim} \), then simplification will occur. Note that the \( \text{factorlim} \) switch controls simplification of the result of \( \gamma \) of an integer argument as well.
**makegamma**  \((\text{expr})\)  Function
Transforms instances of binomial, factorial, and beta functions in \(\text{expr}\) 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 \(\text{beta}(a+n,b)\) and \(\text{beta}(a-n,b)\) or \(\text{beta}(a,b+n)\) and \(\text{beta}(a,b-n)\) 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 \(\log_{\text{gamma}}\):
\[
- \log_{\text{gamma}}(b + a) + \log_{\text{gamma}}(b) + \log_{\text{gamma}}(a) \quad \%e
\]
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:
\[
(\%i1) \quad \text{beta}(2,3), \text{beta}(2,1/3), \text{beta}(2,a);
\]
\[
(\%o1) \quad \frac{1}{12}, \frac{9}{4}, \frac{1}{a (a + 1)}
\]
Simplification for two rational numbers as arguments which sum to an integer:
\[
(\%i2) \quad \text{beta}(1/2,5/2), \text{beta}(1/3,2/3), \text{beta}(1/4,3/4));
\]
\[
(\%o2) \quad \frac{3 \, \%pi}{8}, \frac{2 \, \%pi}{\sqrt{3}}
\]
When setting **beta_args_sum_to_integer** to true more general expression are simplified, when the sum of the arguments is an integer:
\[
(\%i3) \quad \text{beta_args_sum_to_integer}:\text{true}$
(\%i4) \quad \text{beta}(a+1,-a+2);
\]
\[
(\%o4) \quad \frac{\%pi (a - 1) a}{2 \, \sin(\%pi (2 - a))}
\]
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\}
\beta(a+n,b) or \beta(a-n) with n an integer simplifies when \texttt{beta\_expand} is \texttt{true}:
(\%16) \texttt{beta\_expand: true}\$
(\%17) \{beta(a+1,b), beta(a-1,b), beta(a+1,b)/beta(a,b+1)\};
(\%o7) \{\frac{a \beta(a, b)}{b + a}, \frac{a \beta(a, b)}{b - a}, -\}
\beta is not defined, when one of the arguments is zero:
(\%17) beta(0,b);
beta: expected nonzero arguments; found 0, b
-- an error. To debug this try debugmode(true);

\textbf{Numerical evaluation for real and complex arguments in float or bigfloat precision:}
(\%18) beta(2.5,2.3);
(\%o8) 0.8694748611299981
(\%19) beta(2.5,1.4+%i);
(\%o9) 0.0640144950796695 - .1502078053286415 %i
(\%110) beta(2.5b0,2.3b0);
(\%o10) 8.694748611299969b-2
(\%111) beta(2.5b0,1.4b0+%i);
(\%o11) 6.401449507966944b-2 - 1.502078053286415b-1 %i

\beta is symmetric and has mirror symmetry:
(\%14) 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))

\textbf{beta\_incomplete} \(a, b, z\) \hspace{1cm} \textbf{Function}

The basic definition of the incomplete beta function (A&S 6.6.1) is:

\[
z / \left[ \frac{b - 1}{1 - t} \frac{a - 1}{t} \right] dt \]
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:

\[
\text{gamma}(a) \ \text{hypergeometric\_generalized}\left([a, 1 - b], [a + 1], z\right) \ z
\]

(See 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

\[
\sum_{k=0}^{m-1} \frac{(1 - m) z^n}{(n - k) k! (n - k)}
\]

Maxima uses this definition to simplify \( \text{beta\_incomplete} \) for \( a \) a negative integer.

For \( a \) a positive integer, \( \text{beta\_incomplete} \) 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 \( \text{realpart}(a) > 0 \), \( \text{beta\_incomplete} \) has the specific value zero. For \( z=1 \) and \( \text{realpart}(b) > 0 \), \( \text{beta\_incomplete} \) simplifies to the beta function \( \text{beta}(a,b) \).

Maxima evaluates \( \text{beta\_incomplete} \) 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 \( \text{beta\_expand} \) is true, Maxima expands expressions like \( \text{beta\_incomplete}(a+n,b,z) \) and \( \text{beta\_incomplete}(a-n,b,z) \) where \( n \) is a positive integer.

Maxima knows the derivatives of \( \text{beta\_incomplete} \) 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:

\[
\begin{align*}
(\%i1) & \text{beta\_incomplete}(2,b,z); \\
(\%o1) & \frac{b}{1 - (1 - z) (b z + 1)}
\end{align*}
\]

Simplification for \( b \) a positive integer:

\[
\begin{align*}
(\%i2) & \text{beta\_incomplete}(a,2,z); \\
(\%o2) & \frac{a}{(a (1 - z) + 1) z}
\end{align*}
\]

Simplification for \( a \) and \( b \) a positive integer:

\[
(\%i3) \text{beta\_incomplete}(3,2,z);
\]
\[
\frac{3 (1 - z) + 1}{12} z
\]

\%(o3)

\(a\) is a negative integer and \(b \leq (-a)\), Maxima simplifies:

\%(i4) beta_incomplete(-3,1,z);

\%(o4)

\[\frac{1}{3 z}\]

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) \text{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 \(\text{abs}(z) > 1\) \text{beta_incomplete} returns a complex result:

\%(i11) beta_incomplete(0.25,0.50,1.7);

\%(o11) \text{5.244115108584249} - \text{1.45518047787844 \text{\%i}}

Results for more general complex arguments:

\%(i14) beta_incomplete(0.25+%\%i,1.0+%\%i,1.7+%\%i);

\%(o14) \text{2.726960675662536} - \text{.3831175704269199 \text{\%i}}

\%(i15) beta_incomplete(1/2,5/4+%\%i,2.8+%\%i);

\%(o15) \text{13.0469635168716 \text{\%i} - 5.802067956270001}

\%(i16)

Expansion, when \text{beta_expand} is true:

\%(i23) beta_incomplete(a+1,b,z),beta_expand:true;

\%(o23)

\[\frac{b a}{b + a} \text{beta_incomplete}(a, b, z) \frac{(1 - z)}{z} \frac{(b + a)}{b + a}\]

\%(i24) beta_incomplete(a-1,b,z),beta_expand:true;

\%(o24)

\[\frac{b a - 1}{1 - a} \text{beta_incomplete}(a, b, z) \frac{(- b - a + 1)}{1 - a} \frac{(1 - z)}{z} \frac{(1 - a)}{1 - a}\]

Derivative and integral for \text{beta_incomplete}:

\%(i34) diff(beta_incomplete(a, b, z), z);
Chapter 16: Special Functions 231

\[ (1 - z) \frac{a - 1}{z} \]

(integrate(beta_incomplete(a, b, z), z);

\[ \frac{b - 1}{a - 1} \]

(b a)

(1 - z) z

\[ \frac{b}{b + a} \]

+ beta_incomplete(a, b, z) z

\[ \frac{a}{b + a} \]

a beta_incomplete(a, b, z)

- --------------------------

\[ b + a \]

(beta_incompleteRegularized(a, b, z))

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 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:

(1) beta_incomplete_regularized(2, b, z);

\[ 1 - (1 - z) (b z + 1) \]

(a (1 - z) + 1) z

(3) beta_incomplete_regularized(3, 2, z);

For the specific values \( z = 0 \) and \( z = 1 \), Maxima simplifies:

(14) assume(a>0, b>0)$

(15) beta_incomplete_regularized(a, b, 0);
Numerical evaluation for real and complex arguments in float and bigfloat precision:

```
(%i17) beta_incomplete_regularized(0.12, 0.43, 0.9);  
(%o17) .9114011367359802
(%i18) fpprec:32$  
(%i19) beta_incomplete_regularized(0.12, 0.43, 0.9b0);  
(%o19) 9.1140113673598075519946998779975b-1
(%i20) beta_incomplete_regularized(1+%i, 3/3, 1.5*%i);  
(%o20) .2865367499935403  %i - 0.122995963334684
(%i21) fpprec:20$  
(%i22) beta_incomplete_regularized(1+%i, 3/3, 1.5b0*%i);  
(%o22) 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) - ----------  
       a beta(a, b)
(%i14) beta_incomplete_regularized(a-1,b,z);  
(%o14) beta_incomplete_regularized(a, b, z) - ----------  
       beta(a, b) (b + a - 1)
```

The derivative and the integral wrt z:

```
(%i15) diff(beta_incomplete_regularized(a,b,z),z);  
(%o15) beta(a, b)  
(%i16) integrate(beta_incomplete_regularized(a,b,z),z);  
(%o16) beta(a, b) (b + a - 1)
```

**beta_incomplete_generalized** (a, b, z1, z2)

The basic definition of the generalized incomplete beta function is
Maxima simplifies \texttt{beta_incomplete_regularized} for \(a\) and \(b\) a positive integer. For \(\text{realpart}(a) > 0\) and \(z1 = 0\) or \(z2 = 0\), Maxima simplifies \texttt{beta_incomplete_generalized} to \texttt{beta_incomplete}. For \(\text{realpart}(b) > 0\) and \(z1 = 1\) or \(z2=1\), Maxima simplifies to an expression with \texttt{beta} and \texttt{beta_incomplete}.

Maxima evaluates \texttt{beta_incomplete_regularized} for real and complex values in float and bigfloat precision.

When \texttt{beta_expand} is true, Maxima expands \texttt{beta_incomplete_generalized} for \(a + n\) and \(a - n\), \(n\) a positive integer.

Maxima knows the derivative of \texttt{beta_incomplete_generalized} with respect to the variables \(a\), \(b\), \(z1\), and \(z2\) and the integrals with respect to the variables \(z1\) and \(z2\).

Examples:

Maxima simplifies \texttt{beta_incomplete_generalized} for \(a\) and \(b\) a positive integer:

\begin{verbatim}
(%i1) beta_incomplete_generalized(2,b,z1,z2);
\end{verbatim}
\begin{equation*}
\frac{b (b + 1)}{b b (1 - z1) (b z1 + 1) - (1 - z2) (b z2 + 1)}
\end{equation*}

\begin{verbatim}
(%i2) beta_incomplete_generalized(a,2,z1,z2);
\end{verbatim}
\begin{equation*}
\frac{a (a + 1)}{a (a (1 - z2) + 1) z2 - (a (1 - z1) + 1) z1}
\end{equation*}

\begin{verbatim}
(%i3) beta_incomplete_generalized(3,2,z1,z2);
\end{verbatim}
\begin{equation*}
\frac{12}{(1 - z1) (3 z1 + 2 z1 + 1) - (1 - z2) (3 z2 + 2 z2 + 1)}
\end{equation*}

Simplification for specific values \(z1 = 0\), \(z2 = 0\), \(z1 = 1\), or \(z2 = 1\):

\begin{verbatim}
(%i4) assume(a > 0, b > 0)$
(%i5) beta_incomplete_generalized(a,b,z1,0);
\end{verbatim}
\begin{equation*}
- \texttt{beta_incomplete}(a, b, z1)
\end{equation*}

\begin{verbatim}
(%i6) beta_incomplete_generalized(a,b,0,z2);
\end{verbatim}
\begin{equation*}
- \texttt{beta_incomplete}(a, b, z2)
\end{equation*}

\begin{verbatim}
(%i7) beta_incomplete_generalized(a,b,z1,1);
\end{verbatim}
\begin{equation*}
\texttt{beta}(a, b) - \texttt{beta_incomplete}(a, b, z1)
\end{equation*}

\begin{verbatim}
(%i8) beta_incomplete_generalized(a,b,1,z2);
\end{verbatim}
Numerical evaluation for real arguments in float or bigfloat precision:

\( \text{beta_incomplete_generalized}(1/2,3/2,0.25,0.31); \)
\( \text{beta_incomplete_generalized}(1/2,3/2,0.25,0.31b0); \)

Numerical evaluation for complex arguments in float or bigfloat precision:

\( \text{beta_incomplete_generalized}(1/2+\%i,3/2+\%i,0.25,0.31); \)
\( \text{beta_incomplete_generalized}(1/2+\%i,3/2+\%i,0.25,0.31b0); \)

Expansion for \( a + n \) or \( a - n \), \( n \) a positive integer, when beta_expand is true:

\( \text{beta_incomplete_generalized}(a+1,b,z1,z2); \)
\( \text{beta_incomplete_generalized}(a-1,b,z1,z2); \)

Derivative wrt the variable \( z1 \) and integrals wrt \( z1 \) and \( z2 \):

\( \text{diff(beta_incomplete_generalized(a,b,z1,z2),z1);} \)
\( \text{integrate(beta_incomplete_generalized(a,b,z1,z2),z1);} \)
\( \text{integrate(beta_incomplete_generalized(a,b,z1,z2),z2);} \)
**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 bffac 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)  
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);  
\[ \frac{15 \sqrt{\pi}}{8} \]  
(%o1) \[ \frac{15 \sqrt{\pi}}{8} \]  
(%i2) numfactor (%);  
\[ \frac{15}{8} \]  
(%o2) \[ \frac{15}{8} \]  

16.5 Exponential Integrals

The Exponential Integral and related functions are defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Chapter 5

expintegral_e1 (z)  
The Exponential Integral E1(z) (A&S 5.1.1)

expintegral_ei (z)  
The Exponential Integral Ei(z) (A&S 5.1.2)

expintegral_li (z)  
The Exponential Integral Li(z) (A&S 5.1.3)

expintegral_e (n,z)  
The Exponential Integral En(z) (A&S 5.1.4)

expintegral_si (z)  
The Exponential Integral Si(z) (A&S 5.2.1)

expintegral_ci (z)  
The Exponential Integral Ci(z) (A&S 5.2.2)

expintegral_shi (z)  
The Exponential Integral Shi(z) (A&S 5.2.3)

expintegral_chи (z)  
The Exponential Integral Chi(z) (A&S 5.2.4)

expintrep  
Default value: false  
Change the representation of the Exponential Integral to gamma_incomplete, expintegral_e1, expintegral_ei, expintegral_li, expintegral_trig, expintegral_hyp
Chapter 16: Special Functions

expintexpand

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

16.6 Error Function

The Error function and related functions are defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Chapter 7

**erf** (z)

The Error Function erf(z) (A&S 7.1.1)

See also flag erf\_flag.

**erfc** (z)

The Complementary Error Function erfc(z) (A&S 7.1.2)

erfc(z) = 1 - erf(z)

**erfi** (z)

The Imaginary Error Function.

erfi(z) = -i*erf(i*z)

**erf**\_generalized (z1,z2)

Generalized Error function Erf(z1,z2)

**fresnel\_c** (z)

The Fresnel Integral C(z) = integrate(cos((%pi/2)*t^2),t,0,z). (A&S 7.3.1)

The simplification fresnel\_c(-x) = -fresnel\_c(x) is applied when flag trigsign is true.

The simplification fresnel\_c(%i*x) = %i*fresnel\_c(x) is applied when flag \%iargs is true.

See flags erf\_representation and hypergeometric\_representation.

**fresnel\_s** (z)

The Fresnel Integral S(z) = integrate(sin((%pi/2)*t^2),t,0,z). (A&S 7.3.2)

The simplification fresnel\_s(-x) = -fresnel\_s(x) is applied when flag trigsign is true.

The simplification fresnel\_s(%i*x) = %i*fresnel\_s(x) is applied when flag \%iargs is true.

See flags erf\_representation and hypergeometric\_representation.

**erf**\_representation

Default value: false

When T erfc, erfi, erf\_generalized, fresnel\_s and fresnel\_c are transformed to erf.

**hypergeometric**\_representation

Default value: false

Enables transformation to a Hypergeometric representation for fresnel\_s and fresnel\_c
16.7 Struve Functions

The Struve functions are defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Chapter 12.

\texttt{struve\_h}(v, z) \\
\quad The Struve Function H of order v and argument z. (A&S 12.1.1)

\texttt{struve\_l}(v, z) \\
\quad The Modified Struve Function L of order v and argument z. (A&S 12.2.1)

16.8 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 \texttt{hgfred}.

\texttt{%m}[k,u](z) \\
\quad Whittaker M function \(M[k,u](z) = \exp(-z/2)z^{1/2+u}(1/2+u)*M(1/2+u-k,1+2*u,z)\). (A&S 13.1.32)

\texttt{%w}[k,u](z) \\
\quad Whittaker W function. (A&S 13.1.33)

\texttt{%f}[p,q][a, b, z] \\
\quad The \(pFq(a_1,a_2,..ap;b_1,b_2...bq;z)\) hypergeometric function, where \(a\) a list of length \(p\) and \(b\) a list of length \(q\).

\texttt{hypergeometric}([a1, ..., ap],[b1, ... ,bq], x) \\
\quad The hypergeometric function. Unlike Maxima’s \%f hypergeometric function, the function \texttt{hypergeometric} is a simplifying function; also, \texttt{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 \texttt{expand_hypergeometric} is true (default is false) and one of the arguments \(a1\) through \(ap\) is a negative integer (a polynomial case), \texttt{hypergeometric} returns an expanded polynomial.

Examples:

\begin{verbatim}
(%i1) hypergeometric([],[],x);
(%o1) %e^x
\end{verbatim}

Polynomial cases automatically expand when \texttt{expand_hypergeometric} is true:
```lisp
(%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:

```lisp
(%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 - .001049813688578673 %i 
(%i6) hypergeometric([5,6],[8], 5.7b0 - %i), fpprec : 30;
(%o6) 7.3758240097749467450642010824b-3
```

16.9 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`.

```lisp
parabolic_cylinder_d (v, z) 
```

The parabolic cylinder function `parabolic_cylinder_d(v,z)`. (A&s 19.3.1)

16.10 Functions and Variables for Special Functions

```lisp
specint (exp(- s*t) * expr, t) 
```

Compute the Laplace transform of expr with respect to the variable t. The integrand expr may contain special functions.

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);  
\text{sqrt}(\%pi) \frac{a^{3/2}}{2 (p + -) 4}  
(%o2) \frac{\text{sqrt}(a)}{2 p} \frac{\%e^{-a/p}}{2}  

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));  
\frac{\log(s)}{s - a}  
(%o5) \frac{\log(s)}{2 s + 1}  

(%i6) logarc:true$
(%i7) gamma_expand:true$
radcan(specint((\cos(t)*expintegral_si(t)  
- \sin(t)*expintegral_ci(t))/%e^(-s*t),t));  
\frac{2}{s} \frac{\log(s + a)}{2}  
(%o8) \frac{2}{s} \frac{\log(s + a)}{2}  

Results when using the expansion of \text{gamma_incomplete} and when changing the representation to \text{expintegral_e1}:
(%i10) assume(s>0)$
(%i11) specint(1/sqrt(\%pi*t)*unit_step(t-k)*%e^(-s*t),t);  
\frac{1}{2} \text{gamma_incomplete}(-, k s) \frac{1}{\text{sqrt}(\%pi)} \frac{\text{sqrt}(s)}{2}  
(%o11) \frac{1}{\text{sqrt}(\%pi)} \frac{\text{sqrt}(s)}{2}  

Chapter 16: Special Functions

(%i12) gamma_expand: true$
(%i13) specint(1/sqrt(%pi*t)*unit_step(t-k)*%e^(-s*t), t);
   erfc(sqrt(k) sqrt(s))
   --------------------
   sqrt(s)
(%o13)

(%i14) expintrep: expintegral_e1$
(%i15) ratsimp(specint(1/(t+a)^2*%e^(-s*t), t));
   a s %e expintegral_e1(a s) - 1
   ------------------------------
   a
(%o15)

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) \texttt{[notequal(z, 0)]}
(%i2) hgfred([v+1/2], [2*v+1], 2*%i*z);
   v/2 %i z
   4 bessel_j(v, z) gamma(v + 1) %e
   ---------------------------------------
   v z
(%o2)

(%i3) hgfred([1, 1], [2], z);
   log(1 - z)
   - ---------
   z
(%o3)

(%i4) hgfred([a, a+1/2], [3/2], z^2);
   1 - 2 a
   (z + 1) - (1 - z)
   ----------------------------------
   2 (1 - 2 a) z
(%o4)

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);
\[
\frac{(a - 1) 2 L (z)}{2} \quad \text{(\%o6)}
\]
\[
- \frac{\text{ev}(\%)}{a (a + 1)}
\]
\[
\text{(\%i7)}
\]
\[
\frac{2 z}{a (a + 1)} - \frac{2 z}{a} + 1 \quad \text{(\%o7)}
\]

**lambert_w (z)**

The principal branch of Lambert’s W function \( W(z) \), the solution of \( z = W(z) \cdot \exp(W(z)) \).

**nzeta (z)**

The Plasma Dispersion Function \( \text{nzeta}(z) = %i*\text{sqrt}(\%pi)*\exp(-z^2)*(1-\text{erf}(-%i*z)) \).

**nzetar (z)**

Returns \( \text{realpart}(\text{nzeta}(z)) \).

**nzetai (z)**

Returns \( \text{imagpart}(\text{nzeta}(z)) \).
17 Elliptic Functions

17.1 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)

$$\frac{u - \text{elliptic_e(asin(jacobi_sn(u, m)), m)}}{2 m}
+ \frac{2}{(1 - m)}$$

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(--- + ---))
       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(-)
       2

(%i7) makegamma (%);
(%o7) gamma (-)
       4

(%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))
   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.

17.2 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 \(dn(u, m)\).
The Jacobian elliptic function $ns(u,m) = 1/sn(u,m)$.

The Jacobian elliptic function $sc(u,m) = sn(u,m)/cn(u,m)$.

The Jacobian elliptic function $sd(u,m) = sn(u,m)/dn(u,m)$.

The Jacobian elliptic function $nc(u,m) = 1/cn(u,m)$.

The Jacobian elliptic function $cs(u,m) = cn(u,m)/sn(u,m)$.

The Jacobian elliptic function $cd(u,m) = cn(u,m)/dn(u,m)$.

The Jacobian elliptic function $nd(u,m) = nc(u,m) = 1/cn(u,m)$.

The Jacobian elliptic function $ds(u,m) = dn(u,m)/sn(u,m)$.

The Jacobian elliptic function $dc(u,m) = dn(u,m)/cn(u,m)$.

The inverse of the Jacobian elliptic function $sn(u,m)$.

The inverse of the Jacobian elliptic function $cn(u,m)$.

The inverse of the Jacobian elliptic function $dn(u,m)$.

The inverse of the Jacobian elliptic function $ns(u,m)$.

The inverse of the Jacobian elliptic function $sc(u,m)$.

The inverse of the Jacobian elliptic function $sd(u,m)$.
inverse_jacobi_nc \ (u, m) 

The inverse of the Jacobian elliptic function \( nc(u, m) \).

inverse_jacobi_cs \ (u, m) 

The inverse of the Jacobian elliptic function \( cs(u, m) \).

inverse_jacobi_cd \ (u, m) 

The inverse of the Jacobian elliptic function \( cd(u, m) \).

inverse_jacobi_nd \ (u, m) 

The inverse of the Jacobian elliptic function \( nc(u, m) \).

inverse_jacobi_ds \ (u, m) 

The inverse of the Jacobian elliptic function \( ds(u, m) \).

inverse_jacobi_dc \ (u, m) 

The inverse of the Jacobian elliptic function \( dc(u, m) \).

17.3 Functions and Variables for Elliptic Integrals

elliptic_f \ (\phi, m) 

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 246 and \[elliptic_kc\], page 247.

elliptic_e \ (\phi, m) 

The incomplete elliptic integral of the second kind, defined as

\[
\int_0^\phi \sqrt{1 - m \sin^2 \theta} d\theta
\]

See also \[elliptic_e\], page 246 and \[elliptic_ec\], page 247.

elliptic_eu \ (u, m) 

The incomplete elliptic integral of the second kind, defined as

\[
\int_0^u \text{dn}(v, m) dv = \int_0^\tau \sqrt{1 - mt^2} dt
\]

where \( \tau = \text{sn}(u, m) \).

This is related to \( elliptic_e \) by

\[
E(u, m) = E(\phi, m)
\]

where \( \phi = \sin^{-1} \text{sn}(u, m) \).

See also \[elliptic_e\], page 246.
**elliptic\_pi (n, phi, m)**

The incomplete elliptic integral of the third kind, defined as

\[
\int_0^\phi \frac{d\theta}{(1 - n \sin^2 \theta)\sqrt{1 - m \sin^2 \theta}}
\]

Only the derivative with respect to \( \phi \) is known by Maxima.

**elliptic\_kc (m)**

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 \texttt{makegamma} to evaluate them.

**elliptic\_ec (m)**

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 \texttt{makegamma} to evaluate them.
18 Limits

18.1 Functions and Variables for Limits

lhospitallim  
Option variable

Default: 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 (expr, x, val, dir)  
Function

limit (expr, x, val)  
Function

limit (expr)  
Function

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).

eexample (limit) displays some examples.


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 (expr, x, val, dir) Function

Take the limit of the Taylor series expansion of expr in x at val from direction dir.

Option variable
tlimswitch 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.
19 Differentiation

19.1 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);
   z(x) d
   y(x) %e (-- (z(x)))
   dx
(%i3) a1: antid (expr, x, z(x));
   z(x) d
   [y(x) %e , - %e (-- (y(x)))]
   dx
(%i4) a2: antidiff (expr, x, z(x));
   z(x) d
   [y(x) %e , - %e (-- (y(x)))]
   dx
(%i5) a2 - (first (a1) + 'integrate (second (a1), x));
(%o5) 0
(%i6) antid (expr, x, y(x));
   z(x) d
   [0, y(x) %e (-- (z(x)))]
   dx
(%i7) antidiff (expr, x, y(x));
   z(x) d
   [y(x) %e (-- (z(x)))]
   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);
   %e (--- (z(x)))
   dx
(%o2) y(x) %e
(%i3) a1: antid (expr, x, z(x));
   z(x) d
   [y(x) %e , - %e (-- (y(x)))]
   dx
(%o3) [y(x) %e , - %e (-- (y(x)))]
   dx
(%i4) a2: antidiff (expr, x, z(x));
   z(x) d
   [y(x) %e , - %e (-- (y(x)))]
   dx
(%o4) [y(x) %e , - %e (-- (y(x)))]
   dx
(%i5) a2 - (first (a1) + 'integrate (second (a1), x));
(%o5) 0
(%i6) antid (expr, x, y(x));
   z(x) d
   [0, y(x) %e (-- (z(x)))]
   dx
(%o6) [0, y(x) %e (-- (z(x)))]
   dx
(%i7) antidiff (expr, x, y(x));
   z(x) d
   [y(x) %e (-- (z(x)))]
   dx
(%o7) [y(x) %e (-- (z(x)))]
   dx

atomgrad

Property

atomgrad is the atomic gradient property of an expression. This property is assigned by gradef.
atvalue (expr, [x_1 = a_1, ..., x_m = a_m], c)  
Function
atvalue (expr, x_1 = a_1, c)  
Function

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, ..., x_m), or a derivative, diff (f(x_1, ..., x_m), x_1, n_1, ..., x_m, n_m) in which the function arguments explicitly appear. n_i is the order of differentiation with respect to x_i.

The point at which the atvalue is established is given by the list of equations [x_1 = a_1, ..., x_m = a_m]. If there is a single variable x_1, the sole equation may be given without enclosing it in a list.

printprops ([f_1, f_2, ...], atvalue) displays the atvalues of the functions f_1, f_2, ... 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_1, x_2, ... when atvalues are displayed.

atvalue evaluates its arguments. atvalue returns c, the atvalue.

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);  
\[ @2 + 1 \]
(%o2) @2 + 1

(%i3) printprops (all, atvalue);  
\[ \frac{\text{d}}{\text{d}\@1} \]  
--- (f(@1, @2)) = @2 + 1
\[ \text{d@1} \]  
!@1 = 0

\[ 2 \]
\[ f(0, 1) = a \]

(%i3) done

(%i4) diff (4*f(x,y)^2 - u(x,y)^2, x);  
\[ \frac{\text{d}}{\text{d}x} \]
(%o4) 8 f(x, y) (- (f(x, y))) - 2 u(x, y) (- (u(x, y)))
\[ \text{d}x \]

(%i5) at (%, [x = 0, y = 1]);  
\[ 2 \]
\[ \frac{\text{d}}{\text{d}x} \]
(%o5) 16 a - 2 u(0, 1) (- (u(x, y)))
\[ \text{d}x \]
!x = 0, y = 1
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 \( \wedge \) (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.

\[ \text{del} \ (x) \]

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:

\[
\begin{align*}
(\%i1) & \quad \text{diff (log (x));} \\
(\%o1) & \quad \frac{1}{x} \\
(\%i2) & \quad \text{diff (exp (x*y));} \\
(\%o2) & \quad x \ y \ e^{x \ y} \ d\!e\!l(y) + y \ e^{x \ y} \ d\!e\!l(x) \\
(\%i3) & \quad \text{diff (x*y*z);} \\
(\%o3) & \quad x \ y \ d\!e\!l(z) + x \ z \ d\!e\!l(y) + y \ z \ d\!e\!l(x)
\end{align*}
\]

delta \ (t)  Function

The Dirac Delta function.

Currently only laplace knows about the delta function.

Example:

\[
(\%i1) \quad \text{laplace (delta (t - a) * sin(b*t), t, s);} \\
\text{Is a positive, negative, or zero?} \\
\text{(p);} \\
(\%o1) \quad \frac{-a \ s \ \text{sin(a \ b)}}{\ e}
\]

dependencies  System variable

Default value: []

dependencies is the list of atoms which have functional dependencies, assigned by depends or gradef. The dependencies list is cumulative: each call to depends or gradef appends additional items.

See depends and gradef.

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, \text{diff} \( (f, x) \) yields zero. If
depends \( f, x \) is declared, \texttt{diff} \( f, x \) yields a symbolic derivative (that is, a diff noun).

Each argument \( f_1, x_1 \), 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 \( x_i \).

\texttt{diff} recognizes indirect dependencies established by \texttt{depends} and applies the chain rule in these cases.

\texttt{remove} \( f, \) dependency removes all dependencies declared for \( f \).

\texttt{depends} returns a list of the dependencies established. The dependencies are appended to the global variable \texttt{dependencies}. \texttt{depends} evaluates its arguments.

\texttt{diff} is the only Maxima command which recognizes dependencies established by \texttt{depends}. Other functions (\texttt{integrate}, \texttt{laplace}, etc.) only recognize dependencies explicitly represented by their arguments. For example, \texttt{integrate} does not recognize the dependence of \( f \) on \( x \) unless explicitly represented as \texttt{integrate} \( (f(x), x) \).

\begin{verbatim}
(%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); 
\[ \frac{dr}{du} \cdot s + r \cdot \frac{ds}{du} \]
(%i6) diff (r.s, t); 
\[ \frac{dr}{du} \cdot \frac{ds}{dt} + \frac{du}{dt} \cdot s + r \cdot \frac{du}{dt} \]
(%i7) remove (r, dependency); 
(%o7) done
(%i8) diff (r.s, t); 
\[ r \cdot \frac{ds}{dt} \]
\end{verbatim}

\texttt{derivabbrev} \hspace{1cm} \texttt{Option variable}

Default value: \texttt{false}

When \texttt{derivabbrev} is \texttt{true}, symbolic derivatives (that is, diff nouns) are displayed as subscripts. Otherwise, derivatives are displayed in the Leibniz notation \( dy/dx \).

\texttt{derivdegree} \( (\text{expr}, y, x) \) \hspace{1cm} \texttt{Function}

Returns the highest degree of the derivative of the dependent variable \( y \) with respect to the independent variable \( x \) occurring in \text{expr}. 
Example:

(%i1) 'diff (y, x, 2) + 'diff (y, z, 3) + 'diff (y, x) * x^2;

3 2
--- + --- + x --
3 2 dx

(%o1) %e (- (f(x))) + %e (-- (f(x)))
2
dx

(%o2) derivdegree (%y, %y, %x);
2

derivlist (var_1, ..., var_k)
Causes only differentiations with respect to the indicated variables, within the ev command.

derivsubst
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 (expr, x_1, n_1, ..., x_m, n_m)
Function
diff (expr, x, n)
Function
diff (expr, x)
Function
diff (expr)
Function
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 of its variables times the differential \( \text{del} \) of each variable. No further simplification of \( \text{del} \) is offered.
The noun form of diff is required in some contexts, such as stating a differential equation. In those 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
(%o1) %e (- (f(x))) + %e (-- (f(x)))
2
dx

(%o2) derivdegree (%y, %y, %x);
2
Chapter 19: Differentiation

(%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 (%o3, x);
   h(x)
   /
   [  
   (%o4) I f(x, y) dy + f(x, h(x)) h(x) - f(x, g(x)) g(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 \( x_j \) appended as additional arguments. Then all the derivative indices will be sorted.
(2) The \( x_i \) may be integers from 1 up to the value of the variable dimension [default value: 4]. This will cause the differentiation to be carried out with respect to the \( x_i \)'th member of the list coordinates which should be set to a list of the names of the coordinates, e.g., \([x, y, z, t]\). If coordinates is bound to an atomic variable, then that variable subscripted by \( x_i \) will be used for the variable of differentiation. This permits an array of coordinate names or subscripted names like \( X[1], X[2], ... \) to be used. If coordinates has not been assigned a value, then the variables will be treated as in (1) above.

**diff**

Special symbol

When **diff** is present as an **evflag** in call to **ev**, all differentiations indicated in **expr** are carried out.

**dscalar** \((f)\)

Function

 Applies the scalar d'Alembertian to the scalar function \( f \).

**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** ("ctensor") loads this function.

**load** ("vect") loads this function.

Examples:
(%i1) load ("vect")$
(%i2) grad (x^2 + y^2 + z^2);
(%o2) grad (z + y + x )
(%i3) express (%);
    d 2 2 2 d 2 2 2 d 2 2 2
(2) [-- (z + y + x ), -- (z + y + x ), -- (z + y + x )]
    dz  dy  dx
(%i4) ev (% , diff);
(%o4) [2 x, 2 y, 2 z]
(%i5) div ([x^2, y^2, z^2]);
(%o5) div [x, y, z ]
(%i6) express (%);
    d 2 2 2 d 2 2 2 d 2 2 2
(2) -- (z ) + -- (y ) + -- (x )
    dz  dy  dx
(%i7) ev (% , diff);
(%o7) 2 z + 2 y + 2 x
(%i8) curl ([x^2, y^2, z^2]);
(%o8) curl [x, y, z ]
(%i9) express (%);
    d 2 2 2 d 2 2 2 d 2 2 2
(2) [-- (z ) - -- (y ), -- (x ) - -- (z ), -- (y ) - -- (x )]
    dy  dz  dx  dy
(%i10) ev (% , diff);
(%o10) [0, 0, 0]
(%i11) laplacian (x^2 * y^2 * z^2);
(%o11) laplacian (x y z )
(%i12) express (%);
    d 2 2 2 d 2 2 2 d 2 2 2
(2) --- (x y z ) + --- (x y z ) + --- (x y z )
    dz  dy  dx
(%i13) ev (% , diff);
(%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** $(f(x_1, ..., x_n), g_1, ..., g_m)$  
**Function**

Defines the partial derivatives (i.e., the components of the gradient) of the function $f$ or variable $a$. 
gradopt (f(x, x_1, ... x_n), g_1, ... g_m) defines \( \frac{df}{dx_i} \) 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.

gradopt (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 \text{depends} \( a, x \)).

The first argument \( f(x_1, \ldots, x_n) \) or \( a \) is quoted, but the remaining arguments \( g_1, \ldots, g_m \) are evaluated. \text{gradopt} returns the function or variable for which the partial derivatives are defined.

\text{gradopt} can redefine the derivatives of Maxima’s built-in functions. For example, \text{gradopt} (sin(x), sqrt (1 - sin(x)^2)) redefines the derivative of \text{sin}.

\text{gradopt} cannot define partial derivatives for a subscripted function.

\text{printprops ([f_1, \ldots, f_n], \text{gradopt})} displays the partial derivatives of the functions \( f_1, \ldots, f_n \), as defined by \text{gradopt}.

\text{printprops ([a_n, \ldots, a_n], \text{atomgrad})} displays the partial derivatives of the variables \( a_n, \ldots, a_n \), as defined by \text{gradopt}.

\text{gradopt} is the list of the functions for which partial derivatives have been defined by \text{gradopt}. \text{gradopt} does not include any variables for which partial derivatives have been defined by \text{gradopt}.

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.

\text{gradopt}

System variable

Default value: []

\text{gradopt} is the list of the functions for which partial derivatives have been defined by \text{gradopt}. \text{gradopt} does not include any variables for which partial derivatives have been defined by \text{gradopt}.

\text{laplace}(\text{expr}, t, s)

Function

Attempts to compute the Laplace transform of \text{expr} with respect to the variable \( t \) and transform parameter \( s \).

\text{laplace} recognizes in \text{expr} the functions \text{delta}, \text{exp}, \text{log}, \text{sin}, \text{cos}, \text{sinh}, \text{cosh}, and \text{erf}, as well as \text{derivative}, \text{integrate}, \text{sum}, and \text{ilt}. If \text{laplace} fails to find a transform the function \text{specint} is called. \text{specint} can find the laplace transform for expressions with special functions like the bessel functions \text{bessel_j}, \text{bessel_i}, ... and can handle the \text{unit_step} function. See also \text{specint}.

If \text{specint} cannot find a solution too, a noun \text{laplace} is returned.

\text{expr} may also be a linear, constant coefficient differential equation in which case \text{atvalue} of the dependent variable is used. The required \text{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 \( \int f(x) * g(t - x), \)
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 ift, the inverse Laplace transform.

Examples:

\[
\begin{align*}
(\%i1) \quad \text{laplace (exp (2*t + a) * sin(t) * t, t, s);} \\
(\%o1) & \quad \frac{a \text{e}^{2 s - 4}}{2} \frac{2}{(s - 4 s + 5)} \\
(\%i2) \quad \text{laplace ('diff (f (x), x), x, s);} \\
(\%o2) & \quad s \text{ laplace}(f(x), x, s) - f(0) \\
(\%i3) \quad \text{diff (diff (delta (t), t), t);} \\
(\%o3) & \quad \frac{-d}{2} \text{ (delta(t))} \\
(\%i4) \quad \text{laplace (%, t, s);} \\
(\%o4) & \quad \frac{d}{2} \text{ (delta(t))} + s - \text{delta(0) s} \\
(\%i5) \quad \text{assume(a>0)$} \\
(\%i6) \quad \text{laplace(gamma_incomplete(a,t),t,s),gamma_expand:true;} \\
(\%o6) & \quad \frac{\text{gamma}(a)}{s} \frac{\text{gamma}(a)}{\frac{a}{s} + 1} \\
(\%i7) \quad \text{factor(laplace(gamma_incomplete(1/2,t),t,s));} \\
(\%o7) & \quad \frac{\text{sqrt(\%pi) (sqrt(s) sqrt(\frac{a}{s}) + 1)} - 1}{s} \\
(\%i8) \quad \text{assume(exp(\%pi*s)>1)$} \\
(\%i9) \quad \text{laplace(sum(((-1)^n*unit_step(t-n*\%pi)*sin(t),n,0,inf),t,s)}, \text{simpsum};}
\end{align*}
\]
Chapter 19: Differentiation

(%o9) \[
\frac{\pi i}{(s + \pi i) (1 - e^{i \pi}) (s - \pi i) (1 - e^{i \pi})} \quad \frac{\pi i}{2} \\
\]

(%i9) factor(%);

(%o9) \[
\frac{\pi i}{(s - \pi i) (s + \pi i) (e^{i \pi} - 1)} \\
\]

(%i9) factor(%);
20 Integration

20.1 Introduction to Integration

Maxima has several routines for handling integration. The \texttt{integrate} function makes use of most of them. There is also the \texttt{ 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 \texttt{quad_qag}, \texttt{quad_qags}, etc., which are described under the heading QUADPACK. Hypergeometric functions are being worked on, see \texttt{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 $h(x)$.

20.2 Functions and Variables for Integration

\begin{verbatim}
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 \( y \).

(%i1) assume(a > 0)$
(%i2) 'integrate (%e**sqrt(a*y), y, 0, 4);
   4
   /  sqrt(a) sqrt(y)
   I %e dy
   /
   0
(%o2)

(%i3) changevar (%o2, 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 \texttt{ integrate} above, may be evaluated by \texttt{ ev} with the \texttt{nouns} flag. For example, the expression returned by \texttt{ changevar} above may be evaluated by \texttt{ ev} (%o3, nouns).

\texttt{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., $i = j + \ldots$, not a higher degree function. E.g.,
\end{verbatim}
(%i4) sum (a[i]*x^(i-2), i, 0, inf);
    inf
   ===
   \    i - 2
   > a x
  /   i
 ===
   i = 0
(%o4) > 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: \( \text{dblint}_x \) and \( \text{dblint}_y \), both of
which are initially 10, and can be changed independently to other integer values (there
are \( 2*\text{dblint}_x+1 \) points computed in the x direction, and \( 2*\text{dblint}_y+1 \) 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 \( \text{dblint}_x \) and \( \text{dblint}_y \) 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 \( \text{dblint} \). This will result
in orders of magnitude speed improvement over interpreted code in many cases!

demo (dblint) executes a demonstration of \( \text{dblint} \) 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 /                                               2
   [        ]
 (%o1)     I f(t - x) sinh(a x) dx + b f(t) = t
          0

(%i2) laplace (%o1, t, s);
   a laplace(f(t), t, s)                  2
   b laplace(f(t), t, s) + ----------- = --
   2  2  3
   s - a s

(%i3) linsolve ([%o2, %o1], [laplace(f(t), t, s)]);
   2  2
   2 s - 2 a

(%o3) [laplace(f(t), t, s) = ------------------]
   5  2  3
   b s + (a - a b) s
```
(%i4) ilt (rhs (first (%)), s, t);
Is  \(ab (ab - 1)\) positive, negative, or zero?
pos;

\[
\sqrt{ab (ab - 1)} \frac{t}{2 \cosh\left(\frac{1}{b} - \frac{a}{t}\right)} + \frac{2}{3} \frac{ab - 2 (ab + a)}{a b - 2 a b + a} + \frac{2}{3} \frac{ab - 2 (ab + a)}{a b - 2 a b + a}
\]

**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);

\[
\frac{1}{\sqrt{x} + 1} \, dx
\]

(\%o1) \[
\]

(\%i2) integrate(1/(sqrt(x)+1),x,0,1),intanalysis:false;

(\%o2) \[
2 - 2 \log(2)
\]

(\%i3) integrate(cos(a)/sqrt((tan(a))^2+1),a,-%pi/2,%pi/2);
The number 1 isn't in the domain of atanh
-- an error. To debug this try: debugmode(true);

(\%i4) intanalysis:false$

(\%i5) integrate(cos(a)/sqrt((tan(a))^2+1),a,-%pi/2,%pi/2);

\[
\%pi \quad \text{---}
\]

(\%o5) \[
\frac{\%pi}{2}
\]
Function \texttt{integrate} (\textit{expr}, \textit{x})

Attempts to symbolically compute the integral of \textit{expr} with respect to \textit{x}. \texttt{integrate} (\textit{expr}, \textit{x}) is an indefinite integral, while \texttt{integrate} (\textit{expr}, \textit{x}, \textit{a}, \textit{b}) is a definite integral, with limits of integration \textit{a} and \textit{b}. The limits should not contain \textit{x}, although \texttt{integrate} does not enforce this restriction. \textit{a} need not be less than \textit{b}. If \textit{b} is equal to \textit{a}, \texttt{integrate} returns zero.

See \texttt{quad_qag} and related functions for numerical approximation of definite integrals. See \texttt{residue} for computation of residues (complex integration). See \texttt{antid} for an alternative means of computing indefinite integrals.

The integral (an expression free of \texttt{integrate}) is returned if \texttt{integrate} succeeds. Otherwise the return value is the noun form of the integral (the quoted operator \texttt{’integrate}) or an expression containing one or more noun forms. The noun form of \texttt{integrate} is displayed with an integral sign.

In some circumstances it is useful to construct a noun form by hand, by quoting \texttt{integrate} with a single quote, e.g., \texttt{’integrate} (\textit{expr}, \textit{x}). For example, the integral may depend on some parameters which are not yet computed. The noun may be applied to its arguments by \texttt{ev \{i, nouns\}} where \textit{i} is the noun form of interest.

\texttt{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 (\texttt{inf} or \texttt{minf}), trigonometric functions with limits of integration equal to zero and \texttt{\%pi} or 2 \texttt{\%pi}, rational functions, integrals related to the definitions of the \texttt{beta} and \texttt{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 \texttt{ldefint}.

Special cases of indefinite integrals include trigonometric functions, exponential and logarithmic functions, and rational functions. \texttt{integrate} may also make use of a short table of elementary integrals.

\texttt{integrate} may carry out a change of variable if the integrand has the form \textit{f(g(x))} \* \textit{diff(g(x), x)}. \texttt{integrate} attempts to find a subexpression \textit{g(x)} such that the derivative of \textit{g(x)} divides the integrand. This search may make use of derivatives defined by the \texttt{gradef} function. See also \texttt{changevar} and \texttt{antid}.

If none of the preceding heuristics find the indefinite integral, the Risch algorithm is executed. The flag \texttt{risch} may be set as an \texttt{evflag}, in a call to \texttt{ev} or on the command line, e.g., \texttt{ev (integrate (expr, x), risch)} or \texttt{integrate (expr, x), risch}. If \texttt{risch} is present, \texttt{integrate} calls the \texttt{risch} function without attempting heuristics first. See also \texttt{risch}.

\texttt{integrate} works only with functional relations represented explicitly with the \textit{f(x)} notation. \texttt{integrate} does not respect implicit dependencies established by the \texttt{depends} function.

\texttt{integrate} may need to know some property of a parameter in the integrand. \texttt{integrate} will first consult the \texttt{assume} database, and, if the variable of interest
is not there, \texttt{integrate} will ask the user. Depending on the question, suitable responses are \texttt{yes}; or \texttt{no}; or \texttt{pos}; \texttt{zero}; or \texttt{neg};.

\texttt{integrate} is not, by default, declared to be linear. See \texttt{declare} and \texttt{linear}.

\texttt{integrate} attempts integration by parts only in a few special cases.

Examples:

- Elementary indefinite and definite integrals.
  
  \begin{verbatim}
  (%i1) integrate (sin(x)^3, x);
  \end{verbatim}
  \begin{verbatim}
  3
  cos (x) ------- - cos(x)
  \end{verbatim}
  \begin{verbatim}
  (%i2) integrate (x/ sqrt (b^2 - x^2), x);
  \end{verbatim}
  \begin{verbatim}
  2 2
  - sqrt(b - x )
  \end{verbatim}
  \begin{verbatim}
  (%i3) integrate (cos(x)^2 * exp(x), x, 0, %pi);
  \end{verbatim}
  \begin{verbatim}
  %pi
  3 %e 3
  ------- - -
  5 5
  \end{verbatim}
  \begin{verbatim}
  (%i4) integrate (x^2 * exp(-x^2), x, minf, inf);
  \end{verbatim}
  \begin{verbatim}
  sqrt(%pi)
  ---------
  2
  \end{verbatim}

- Use of \texttt{assume} and interactive query.

  \begin{verbatim}
  (%i1) assume (a > 1)$
  \end{verbatim}
  \begin{verbatim}
  (%i2) integrate (x**a/(x+1)**(5/2), x, 0, inf);
  \end{verbatim}
  \begin{verbatim}
  2 a + 2
  is ------- an integer?
  5
  \end{verbatim}
  \texttt{no};

  \begin{verbatim}
  is 2 a - 3 positive, negative, or zero?
  \end{verbatim}
  \texttt{neg};

  \begin{verbatim}
  (%o2)
  beta(a + 1, - - a)
  \end{verbatim}
  \begin{verbatim}
  3
  2
  \end{verbatim}

- Change of variable. There are two changes of variable in this example: one using a derivative established by \texttt{gradef}, and one using the derivation \texttt{diff(r(x))} of an unspecified function \texttt{r(x)}.

  \begin{verbatim}
  (%i3) gradef (q(x), sin(x**2));
  \end{verbatim}
  \begin{verbatim}
  (%i3) (q(x))
  \end{verbatim}
%i4) diff (log (q (r (x))), x);
   \[ \frac{d}{dx} \left( -\frac{(r(x)) \sin(r(x))}{2} \right) \]
   \[ \frac{\text{--} (r(x)) \sin(r(x))}{q(r(x))} \]

%o4)

%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));
   \[ x^4 - 4x^3 + 2x^2 - 7x - 4 \]
%o1)

%i2) integrate (1/%, x);
   \[ \frac{\log(x - 4)}{73} - \frac{\text{--} (\int \frac{x^2 + 4x + 18}{x^3 + 2x + 1}, x) / 73}}{73} \]
%o2)

%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.

%i1) f_1 (a) := integrate (x^3, x, 1, a);
   \[ \int_1^a x^3 \, dx \]
%o1)

%i2) ev (f_1 (7), nouns);
%o2) 600

%i3) /* Note parentheses around integrate(...) here */
    f_2 (a) := '''(integrate (x^3, x, 1, a));
   \[ \frac{a^4 - 1}{4} \]
%o3)

%i4) f_2 (7);
%o4) 600

**integration_constant**

Default value: %c
When a constant of integration is introduced by indefinite integration of an equation, the name of the constant is constructed by concatenating \texttt{integration_constant} and \texttt{integration_constant_counter}.

\texttt{integration_constant} may be assigned any symbol.

Examples:

\begin{verbatim}
(%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
\end{verbatim}

\texttt{integration_constant_counter} 

\textbf{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 \texttt{integration_constant} and \texttt{integration_constant_counter}.

\texttt{integration_constant_counter} is incremented before constructing the next integration constant.

Examples:

\begin{verbatim}
(%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]
\end{verbatim}
Chapter 20: Integration

(%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:

(%i11) integrate_use_rootsof: false$
(%i12) integrate (1/(1+x+x^5), x);
/ 2
[ x - 4 x + 5
I -------------- dx
] 3 2
2 x + 1
2
5 atan(--

/ x - x + 1
log(x + x + 1)

%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:

(%i13) integrate_use_rootsof: true$
(%i14) integrate (1/(1+x+x^5), x);
===== 2
\ (%r4 - 4 %r4 + 5) log(x - %r4)
> ---------------------------------------------
/ %r4 in rootsof(x - x + 1)

(%o4) ---------------------------------------------

7

2
5 atan--

\ log(x + x + 1)

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) -
 (%i2) residue (sin(a*x)/x**4, x, 0); a
 (%o2) - --
 6

risch (expr, x) Function
Integrates expr with respect to x using the transcendental case of the Risch algo-

rithm. (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
 %pi x erf(x) + (sqrt(%pi) x + sqrt(%pi)) %e
 (%o1) ------------------------------------------
 3 %pi
Chapter 20: Integration

(%i2) diff(%o4, x), ratsimp;
2
(%o2) x erf(x)

tldefint (expr, x, a, b)

Function

Equivalent to ldefint with tlimswitch set to true.

20.3 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\(^1\), E. de Doncker\(^2\), C. Ueberhuber\(^3\), and D. Kahaner\(^4\).

The QUADPACK library included in Maxima is an automatic translation (via the program f2cl) of the Fortran source code of QUADPACK as it appears in the SLATEC Common Mathematical Library, Version 4.1\(^5\). The SLATEC library is dated July 1993, but the QUADPACK functions were written some years before. There is another version of QUADPACK at Netlib\(^6\); 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\(^7\).

20.3.1 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.

---

\(^1\) Applied Mathematics and Programming Division, K.U. Leuven
\(^2\) Applied Mathematics and Programming Division, K.U. Leuven
\(^3\) Institut für Mathematik, T.U. Wien
\(^4\) National Bureau of Standards, Washington, D.C., U.S.A
\(^5\) http://www.netlib.org/slatec
\(^6\) http://www.netlib.org/quadpack
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 \( x = c \).

20.4 Functions and Variables for QUADPACK

\quad_qag (f(x), x, a, b, key, [epsrel, epsabs, limit]) \quad Function
\quad_qag (f, x, a, b, key, [epsrel, epsabs, limit]) \quad Function
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) [0.444444444492108, 3.1700968502883E-9, 961, 0]

(%i2) integrate (x^(1/2)*log(1/x), x, 0, 1);
(%o2) 4

Function quad_qags (f(x), x, a, b, [epsrel, epsabs, limit])

Function 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 \texttt{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:
```lisp
(%i1) quad_qags (x^(1/2)*log(1/x), x, 0, 1, 'epsrel=1d-10);
(%o1) [.4444444444444448, 1.11022302462516E-15, 315, 0]
```
Note that `quad_qags` is more accurate and efficient than `quad_qag` for this integrand.

### Function quad_qagi

**quad_qagi** `(f(x), 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:

```lisp
(%i1) quad_qagi (x^2*exp(-4*x), x, 0, inf, 'epsrel=1d-8);
(%o1) [0.03125, 2.95916102995002E-11, 105, 0]
(%i2) integrate (x^2*exp(-4*x), x, 0, inf);
      1
(%o2) --
      32
```

**quad_qawc** (f(x), x, 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:

- \texttt{epsrel} \quad \text{Desired relative error of approximation. Default is 1d-8.}
- \texttt{epsabs} \quad \text{Desired absolute error of approximation. Default is 0.}
- \texttt{limit} \quad \text{Size of internal work array. \textit{limit} is the maximum number of subintervals}
to use. Default is 200.

\texttt{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:

- \( 0 \) \quad \text{no problems were encountered;}
- \( 1 \) \quad \text{too many sub-intervals were done;}
- \( 2 \) \quad \text{excessive roundoff error is detected;}
- \( 3 \) \quad \text{extremely bad integrand behavior occurs;}
- \( 6 \) \quad \text{if the input is invalid.}

Examples:

\begin{verbatim}
(%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]
\end{verbatim}
(%i2) integrate (2^(-alpha)*(((x-1)^2 + 4^(-alpha))*(x-2))^(-1), x, 0, 5);
Principal Value
alpha
alpha 9 4 9
4 log(---------------- + ----------------)
alpha alpha
64 4 + 4 64 4 + 4
(%o2) (-----------------------------------------
alpha
2 4 + 2
3 alpha
-----------
2 4 atan(4 4 ) 2 4 atan(4 ) alpha
---------- ----------
alpha/2 alpha/2
2 4 + 2 2 4 + 2
(%i3) ev (%, alpha=5, numer);
(%o3) - 3.130120337415917

quad_qawf \( f(x), x, a, \omega, \text{trig}, [\text{epsabs, limit, maxp1, limlst}] \)  Function
quad_qawf \( f, x, a, \omega, \text{trig}, [\text{epsabs, limit, maxp1, limlst}] \)  Function
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 \( \text{trig} \):
\begin{align*}
\cos & \quad w(x) = \cos(\omega x) \\
\sin & \quad w(x) = \sin(\omega x)
\end{align*}
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 \text{key=val}. The keyword arguments are:
\begin{align*}
\text{epsabs} & \quad \text{Desired absolute error of approximation. Default is 1d-10.} \\
\text{limit} & \quad \text{Size of internal work array.} \quad (\text{limit} - \text{limlst})/2 \text{ is the maximum number of subintervals to use. Default is 200.} \\
\text{maxp1} & \quad \text{Maximum number of Chebyshev moments. Must be greater than 0. Default is 100.}
\end{align*}
**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(%)ipi
- 2
(%o2) -----------------------------

(%i3) ev (%o2, numer); (%o3) .6901942235215714
```

**quad_qawo(f(x), x, a, b, omega, trig, [epsrel, epsabs, limit, maxp1, limlst])**

Function

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:
\textbf{epsrel}  
Desired relative error of approximation. Default is 1d-8.

\textbf{epsabs}  
Desired absolute error of approximation. Default is 0.

\textbf{limit}  
Size of internal work array. \textit{limit}/2 is the maximum number of subintervals to use. Default is 200.

\textbf{maxp1}  
Maximum number of Chebyshev moments. Must be greater than 0. Default is 100.

\textbf{limlst}  
Upper bound on the number of cycles. Must be greater than or equal to 3. Default is 10.

\textbf{quad_qawo} returns a list of four elements:

\begin{itemize}
  \item an approximation to the integral,
  \item the estimated absolute error of the approximation,
  \item the number integrand evaluations,
  \item an error code.
\end{itemize}

The error code (fourth element of the return value) can have the values:

\begin{itemize}
  \item 0 no problems were encountered;
  \item 1 too many sub-intervals were done;
  \item 2 excessive roundoff error is detected;
  \item 3 extremely bad integrand behavior occurs;
  \item 6 if the input is invalid.
\end{itemize}

Examples:

\begin{verbatim}
(\%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));  
\[\frac{\alpha/2 - 1/2}{\sqrt{\pi}} \frac{2 \alpha}{\sqrt{\sqrt{2 + 1} + 1}}\]
(\%o2) \frac{2 \alpha}{\sqrt{\sqrt{2 + 1} + 1}}
(\%i3) ev (%a, alpha=2, numer);  
(\%o3) 1.376043390090716
\end{verbatim}

\textbf{quad_qaws} \((f(x), x, a, b, alpha, beta, wfun, [\text{epsrel, epsabs, limit}])\)  
Function

\textbf{quad_qaws} \((f, x, a, b, alpha, beta, wfun, [\text{epsrel, epsabs, limit}])\)  
Function

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.

\textbf{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 \texttt{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 \texttt{key=val}. The keyword arguments are:

- \texttt{epsrel} Desired relative error of approximation. Default is 1d-8.
- \texttt{epsabs} Desired absolute error of approximation. Default is 0.
- \texttt{limit} Size of internal work array. \texttt{limit} is the maximum number of subintervals to use. Default is 200.

\texttt{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:

\begin{verbatim}
(%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)
alpha
--- 4 2 + 2
\end{verbatim}
(%i3) ev (% , alpha=4, numer);
(%o3) 8.750097361672829
Chapter 21: Equations

21 Equations

21.1 Functions and Variables for Equations

\%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. 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 t - 3 t + 4, y = t , z = t ]]
     2  1  2  1

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 doesn’t 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 ([expr_1, ..., expr_m], [x_1, ..., x_n])

Function

algsys ([eqn_1, ..., eqn_m], [x_1, ..., x_n])

Function

Solves the simultaneous polynomials expr_1, ..., expr_m or polynomial equations
eqn_1, ..., eqn_m for the variables x_1, ..., 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, ..., x_n which satisfy the system of equations. If algsys
cannot find a solution, an empty list [] is returned.
The symbols \%r1, \%r2, ..., 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 \texttt{solve} is called to find an exact solution.

In some cases, \texttt{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 \texttt{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 \texttt{realonly} is \texttt{true}, the function \texttt{realroots} is called to find the real-valued solutions. If \texttt{realonly} is \texttt{false}, then \texttt{allroots} is called which looks for real and complex-valued solutions.

If \texttt{algsys} produces a solution which has fewer significant digits than required, the user can change the value of \texttt{algepsilon} to a higher value.

If \texttt{algexact} is set to \texttt{true}, \texttt{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 \texttt{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: "\texttt{algsys} cannot solve - system too complicated."

Interactions with \texttt{radcan} can produce large or complicated expressions. In that case, it may be possible to isolate parts of the result with \texttt{pickapart} or \texttt{reveal}.

Occasionally, \texttt{radcan} may introduce an imaginary unit \%i into a solution which is actually real-valued.

Examples:

\begin{verbatim}
 (%i1) e1: 2*x*(1 - a1) - 2*(x - 1)*a2;
 (%o1) 2 (1 - a1) x - 2 a2 (x - 1)
 (%i2) e2: a2 - a1;
 (%o2) a2 - a1
 (%i3) e3: a1*(-y - x^2 + 1); 2
 (%o3) a1 (-y - x + 1)
 (%i4) e4: a2*(y - (x - 1)^2);
 (%o4) a2 (y - (x - 1))
\end{verbatim}
Chapter 21: Equations

(%i5) algsys ([e1, e2, e3, e4], [x, y, a1, a2]);
(%o5) [[x = 0, y = %r1, a1 = 0, a2 = 0], [x = 1, y = 0, a1 = 1, a2 = 1]]

(%i6) e1: x^2 - y^2;
    2  2
(%o6) x - y

(%i7) e2: -1 - y + 2*y^2 - x + x^2;
     2  2
(%o7) 2 y - y + x - x - 1

(%i8) algsys ([e1, e2], [x, y]);

1 1
(%o8) [[x = - -------, y = -------],
    sqrt(3) sqrt(3)
1 1 1 1
[x = -------, y = - -------], [x = - -, y = - -], [x = 1, y = 1]
    sqrt(3) sqrt(3) 3 3

allroots (expr) Function
allroots (eqn) Function

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.


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 = 1.0]
(%i13) for e in soln
do (e2: subst (e, eqn), disp (expand (lhs(e2) - rhs(e2))));
   - 3.55271368805E-15
   - 5.32907051820075E-15
    4.44089209850063E-15 %i - 4.88498130835069E-15
    - 4.44089209850063E-15 %i - 4.88498130835069E-15
    3.55271368805E-15

(%o3) done

(%i14) polyfactor: true$

(%i15) allroots (eqn);

(%o15) -13.5 (x - 1.0) (x - .8296749902129361)
  2
  (x + 1.015755543828121) (x + .8139194463848151 x
  + 1.098699797110288)

bfallroots (expr) Function
bfallroots (eqn) Function

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], page 287 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.

(%i11) eq1 : x + y + z = 6$
(%i12) eq2 : x - y + z = 2$
(%i13) eq3 : x + y - z = 0$
(%i14) backsubst : false$
(%i15) linsolve ([eq1, eq2, eq3], [x,y,z]);
(%o15) [x = z - y, y = 2, z = 3]
(%i16) backsubst : true$
(%i17) linsolve ([eq1, eq2, eq3], [x,y,z]);
(%o17) [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);
   sqrt(23) 25 1/3
(%t3)  (--------- + --)
       6 sqrt(3) 54
Solution:

sqrt(3) %i 1
---------- - -
2 2

sqrt(3) %i 1
---------- - -
2 2

--

x = (- --------- - -) %t3 + ------------------- - -
     2 2

1 1
x = %t3 + ------ - -
     9 %t3 3
(%o6) [%t4, %t5, %t6]
(%i6) breakup: false$
(%i7) solve (x^3 + x^2 - 1);
Solution:

sqrt(3) %i 1
---------- - -
2 2

sqrt(23) 25 1/3
---------- + --
2 2

sqrt(23) 25 1/3
---------- + --
2 2

9 (--------- + --)
6 sqrt(3) 54

sqrt(3) %i 1 1
(- --------- - -) - -
     2 2 3
```
\( x = \frac{\sqrt{23}}{6} + \frac{1}{2} \) (\%t8)

\[ \frac{\sqrt{3}}{2} - \frac{1}{2} + \frac{1}{3} \] (\%t9)

\( x = \frac{\sqrt{23}}{6} + \frac{1}{2} + \frac{1}{3} \) (\%o9)

**dimension** \( (eqn) \) \hfill Function

**dimension** \( (eqn_1, \ldots, eqn_n) \) \hfill Function

dimen is a package for dimensional analysis. load ("dimen") loads this package.
demo ("dimen") displays a short demonstration.

**dispflag** \hfill Option variable

Default value: \texttt{true}

If set to \texttt{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 \texttt{dispflag} to \texttt{false}.

**funcsolve** \( (eqn, g(t)) \) \hfill Function

Returns \( [g(t) = \ldots] \) or [], depending on whether or not there exists a rational function \( g(t) \) satisfying \( eqn \), which must be a first order, linear polynomial in \( g(t) \) and \( g(t+1) \)

\( eqn: (n + 1)*f(n) - (n + 3)*f(n + 1)/(n + 1) = (n - 1)/(n + 2); \)

\( f(n) = \frac{\sqrt{23} \cdot 25/3 \cdot \sqrt{3} \cdot \%i \cdot 1}{6 \cdot \sqrt{3} \cdot 54 \cdot 2 \cdot 2} \)

\( \frac{\sqrt{3} \cdot \%i \cdot 1}{2 \cdot 2 \cdot 1} + \frac{\sqrt{23} \cdot 25/3 \cdot 3}{9 \cdot \sqrt{3} \cdot 54} \)

\( f(n) = \frac{\sqrt{23} \cdot 25/3 \cdot 1}{6 \cdot \sqrt{3} \cdot 54 \cdot 1 \cdot 1} \)

\[ \frac{\sqrt{3} \cdot \%i \cdot 1}{2 \cdot 2 \cdot 1} + \frac{\sqrt{23} \cdot 25/3 \cdot 3}{9 \cdot \sqrt{3} \cdot 54} \]

\( f(n) = \frac{n + 1}{n + 2} \)

\( f(n) = \frac{n}{(n + 1)(n + 2)} \)

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

(%t2) x : --
    17

(%t3) y : --
    7

(%o3) [[%t2, %t3]]
(%i3) x;
    17
(%o3) --
    7
(%i4) y;
(%o4) y
```

```
(%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
```
Function
\textbf{ieqn} \texttt{(ie, unk, tech, n, guess)}

\texttt{ieqn} is a package for solving integral equations. \texttt{load ("inteqn")} loads this package.

\texttt{ie} is the integral equation; \texttt{unk} is the unknown function; \texttt{tech} is the technique to be tried from those given above (\texttt{tech = first} means: try the first technique which finds a solution; \texttt{tech = all} means: try all applicable techniques); \texttt{n} is the maximum number of terms to take for \texttt{taylor}, \texttt{neumann}, \texttt{firstkindseries}, or \texttt{fredseries} (it is also the maximum depth of recursion for the differentiation method); \texttt{guess} is the initial guess for \texttt{neumann} or \texttt{firstkindseries}.

Default values for the 2nd thru 5th parameters are:

\texttt{unk}: \texttt{p(x)}, where \texttt{p} is the first function encountered in an integrand which is unknown to Maxima and \texttt{x} is the variable which occurs as an argument to the first occurrence of \texttt{p} found outside of an integral in the case of \texttt{secondkind} equations, or is the only other variable besides the variable of integration in \texttt{firstkind} equations. If the attempt to search for \texttt{x} fails, the user will be asked to supply the independent variable.

\texttt{tech}: \texttt{first}

\texttt{n}: 1

\texttt{guess}: \texttt{none} which will cause \texttt{neumann} and \texttt{firstkindseries} to use \texttt{f(x)} as an initial guess.

Option variable
\textbf{ieqnprint}

Default value: \texttt{true}

\texttt{ieqnprint} governs the behavior of the result returned by the \texttt{ieqn} command. When \texttt{ieqnprint} is \texttt{false}, the lists returned by the \texttt{ieqn} function are of the form

\begin{verbatim}
[solution, technique used, nterms, flag]
\end{verbatim}

where \texttt{flag} is absent if the solution is exact.

Otherwise, it is the word \texttt{approximate} or \texttt{incomplete} corresponding to an inexact or non-closed form solution, respectively. If a series method was used, \texttt{nterms} gives the number of terms taken (which could be less than the \texttt{n} given to \texttt{ieqn} if an error prevented generation of further terms).

Function
\textbf{lhs} \texttt{(expr)}

Returns the left-hand side (that is, the first argument) of the expression \texttt{expr}, when the operator of \texttt{expr} is one of the relational operators \texttt{< <= = # equal notequal >= >}, one of the assignment operators \texttt{:= ::= : ::}, or a user-defined binary infix operator, as declared by \texttt{infix}.

When \texttt{expr} is an atom or its operator is something other than the ones listed above, \texttt{lhs} returns \texttt{expr}.

See also \texttt{rhs}.

Examples:
\begin{verbatim}
(%i1) e: aa + bb = cc;
(%o1) bb + aa = cc
(%i2) lhs (e);
(%o2) bb + aa
\end{verbatim}
%i13) rhs (e);
(%o3)  cc

(%i14) [lhs (aa < bb), lhs (aa <= bb), lhs (aa >= bb),
       lhs (aa > bb)];
(%o4) [aa, aa, aa, aa]

(%i15) [lhs (aa = bb), lhs (aa # bb), lhs (equal (aa, bb)),
       lhs (notequal (aa, bb))];
(%o5) [aa, aa, aa, aa]

(%i16) e1: '(foo(x) := 2*x);
(%o6)  foo(x) := 2 x

(%i17) e2: '(bar(y) ::= 3*y);
(%o7)  bar(y) ::= 3 y

(%i18) e3: '(x : y);
(%o8)   x : y

(%i19) e4: '(x :: y);
(%o9)  x :: y

(%i20) e5: [lhs (e1), lhs (e2), lhs (e3), lhs (e4)];
(%o10) [foo(x), bar(y), x, x]

(%i21) 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.

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 triangulararized. 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 ex-
pressed in terms of others.

When programmode is false, linsolve displays the solution with intermediate ex-
pression (%t) labels, and returns the list of labels.

(%i11) e1: x + z = y;
(%o11) z + x = y

(%i12) e2: 2*a*x - y = 2*a^2;
(%o12) 2 a x - y = 2 a

(%o13) e3: y - 2*z = 2;
(%o14) [globalsolve: false, programmode: true];
(%o14) [false, true]

(%i15) linsolve ([e1, e2, e3], [x, y, z]);
(%o5) \[x = a + 1, y = 2 a, z = a - 1\]
(%i6) \[\text{globalsolve: false, programmode: false}\];
(%o6) \[\text{false, false}\]
(%i7) \text{linsolve ([e1, e2, e3], [x, y, z]);}
\text{Solution}

\(\%t7\)
\(\text{z = a - 1}\)
\(\%t8\)
\(\text{y = 2 a}\)
\(\%t9\)
\(\text{x = a + 1}\)
\(\%t10\)
\(\text{[\%t7, \%t8, \%t9]}\)
\(\%t11\) \text{'';}
\(\%t12\)
\(\text{[z = a - 1, y = 2 a, x = a + 1]}\)
(%i8) \text{globalsolve: true, programmode: false};
(%o8) \[\text{true, false}\]
(%i9) \text{linsolve ([e1, e2, e3], [x, y, z]);}
\text{Solution}

\(\%t13\)
\(\text{z : a - 1}\)
\(\%t14\)
\(\text{y : 2 a}\)
\(\%t15\)
\(\text{x : a + 1}\)
\(\%t16\)
\(\text{[\%t11, \%t12, \%t13]}\)
\(\%t17\) \text{'';}
\(\%t18\)
\(\text{[z : a - 1, y : 2 a, x : a + 1]}\)
(%i10) \text{linsolve ([e1, e2, e3], [x, y, z]);}
\text{Solution}

\(\%t19\)
\(\text{[a + 1, 2 a, a - 1]}\)
(%i11) \text{globalsolve: true, programmode: true};
(%o11) \[\text{true, true}\]
(%i12) \text{linsolve ([e1, e2, e3], [x, y, z]);}
\text{[x : a + 1, y : 2 a, z : a - 1]}
(%i13) \text{linsolve ([e1, e2, e3], [x, y, z]);}
\text{[a + 1, 2 a, a - 1]}

\text{lin solves warn} \text{ Option variable}
\text{Default value: true}
When \text{lin solves warn} \text{ is true, linsolve prints a message "Dependent equations eliminated".}

\text{lin solves params} \text{ Option variable}
\text{Default value: true}
When \text{lin solves params} \text{ 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**  
Default value: `not_set_yet`  
multiplicities is set to a list of the multiplicities of the individual solutions returned by `solve` or `realroots`.

**nroots** \((p, low, high)\)  
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`. infinity and plus infinity.  
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** \((p, n)\)  
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 nth power. This routine is much faster than `factor` or even `sqfr`.

**polyfactor**  
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**  
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**  
Default value: `false`  
When `realonly` is `true`, `algsys` returns only those solutions which are free of `%i`.

**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 \([x = x_1, x = x_2, \ldots]\). When programmode is false, realroots creates intermediate expression labels \(%t1, %t2, \ldots\), assigns the results to them, and returns the list of labels.

Examples:

```
(%i1) realroots (-1 - x + x^5, 5e-6);
   612003
(%o1) \[x = \ldots\]
         524288
(%i2) ev (%[1], float);
   1.167303085327148
(%o2) x = 1.167303085327148
(%i3) ev (-1 - x + x^5, %%);
   - 7.396496210176905E-6
(%i1) realroots (expand ((1 - x)^5 * (2 - x)^3 * (3 - x)), 1e-20);
   [x = 1, x = 2, x = 3]
(%o1) \[x = 1, x = 2, x = 3\]
(%i2) multiplicities;
   [5, 3, 1]
```

Function rhs(expr)

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 \(<\leq\#\text{ equal notequal} \rangle\geq\rangle\), 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]
```
Chapter 21: Equations

(%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:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Value of rootsconmode</th>
<th>Result of applying rootscontract</th>
</tr>
</thead>
<tbody>
<tr>
<td>x^(1/2)*y^(3/2)</td>
<td>false</td>
<td>(x*y^3)^(1/2)</td>
</tr>
<tr>
<td>x^(1/2)*y^(1/4)</td>
<td>false</td>
<td>x^(1/2)*y^(1/4)</td>
</tr>
<tr>
<td>x^(1/2)*y^(1/4)</td>
<td>true</td>
<td>(x*y^(1/2))^(1/2)</td>
</tr>
<tr>
<td>x^(1/2)*y^(1/3)</td>
<td>true</td>
<td>x^(1/2)*y^(1/3)</td>
</tr>
<tr>
<td>x^(1/2)*y^(1/4)</td>
<td>all</td>
<td>(x^2*y)^(1/4)</td>
</tr>
<tr>
<td>x^(1/2)*y^(1/3)</td>
<td>all</td>
<td>(x^3*y^2)^(1/6)</td>
</tr>
</tbody>
</table>

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:

(%i11) rootsconmode: false$
(%i12) rootscontract (x^(1/2)*y^(3/2));
(\%o2) \sqrt{x \ y }
(\%o3) \sqrt{x} \ y
(\%i4) \text{rootsconmode: true}\$
(\%o5) \sqrt{x \ \sqrt{y}}
(\%i6) \text{rootsconmode: all}\$
(\%o6) \sqrt{x \ y}
(\%i7) \text{rootsconmode: false}\$
(\%i8) \text{rootsconmode: true}\$
(\%i10) \text{rootsconmode: false}\$
(\%i11) \sqrt{x \ \sqrt{y}}
(\%i13) \sqrt{5+\sqrt{5}} - 5^{1/4}\sqrt{1+\sqrt{5}}
(\%o13) 0

\text{rootsepsilon}

\text{Option variable}

\text{Default value: 1.0e-7}

\text{rootsepsilon} is the tolerance which establishes the confidence interval for the roots found by the \text{realroots} function.

\text{solve (expr, x)}

\text{Function}

\text{solve (expr)}

\text{Function}

\text{solve ([eqn_1, ..., eqn_n], [x_1, ..., x_n])}

\text{Function}

Solves the algebraic equation \text{expr} for the variable \text{x} and returns a list of solution equations in \text{x}. If \text{expr} is not an equation, the equation \text{expr} = 0 is assumed in its place. \text{x} may be a function (e.g. \text{f(x)}), or other non-atomic expression except a sum or product. \text{x} may be omitted if \text{expr} contains only one variable. \text{expr} may be a rational expression, and may contain trigonometric functions, exponentials, etc.

The following method is used:

Let \text{E} be the expression and \text{X} be the variable. If \text{E} is linear in \text{X} then it is trivially solved for \text{X}. Otherwise if \text{E} is of the form \text{A}*\text{X}^N + \text{B} then the result is \((-\text{B}/\text{A})^{1/N}) times the N'th roots of unity.

If \text{E} is not linear in \text{X} then the gcd of the exponents of \text{X} in \text{E} (say \text{N}) is divided into the exponents and the multiplicity of the roots is multiplied by \text{N}. Then \text{solve} is called again on the result. If \text{E} factors then \text{solve} is called on each of the factors. Finally \text{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 \( \texttt{solve} \) 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 \( \texttt{solve} \), \( \texttt{realroots} \), or \( \texttt{allroots} \). Try \( \texttt{apropos (solve)} \) for the switches which affect \( \texttt{solve} \). \( \texttt{describe} \) may then by used on the individual switch names if their purpose is not clear.

\( \texttt{solve ([eqn_1, ..., eqn_n], [x_1, ..., x_n])} \) solves a system of simultaneous (linear or non-linear) polynomial equations by calling \( \texttt{linsolve} \) or \( \texttt{algsys} \) and returns a list of the solution lists in the variables. In the case of \( \texttt{linsolve} \) 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**, \( \texttt{solve} \) 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 \((\text{asin} (\cos (3*x))*(f(x) - 1)), x)\;

solve: using arc-trig functions to get a solution.
Some solutions will be lost.
\[ x = \frac{\pi}{6}, f(x) = 1 \]

\( \%i2 \) ev (solve (5^f(x) = 125, f(x)), solveradcan);

\[ f(x) = \frac{\log(125)}{\log(5)} \]

\( \%i3 \) \([4*x^2 - y^2 = 12, x*y - x = 2]\);

\[ 2 \]

\( \%i4 \) solve \((%, [x, y])\);

\( \%i4 \) \([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 \((1 + a*x + x^3), x)\);
\[ \begin{align*}
\text{(%o5)} \quad & x = \left( - \frac{\sqrt{3} \, \%i}{2} - \frac{1}{2} \right) \left( -\frac{\sqrt{4 \, a + 27}}{6 \, \sqrt{3}} - \frac{1}{2} \right) \\
& - \frac{\sqrt{3} \, \%i}{2} - \frac{1}{2} a \\
& \frac{3}{2} \frac{\sqrt{4 \, a + 27}}{6 \, \sqrt{3}} + \frac{1}{2} \frac{1}{3} \\
& = \frac{3}{2} \frac{\sqrt{4 \, a + 27}}{6 \, \sqrt{3}} + \frac{1}{2} \frac{1}{3} \\
\text{(%i6)} \quad & \text{solve } (x^3 - 1); \\
& \frac{\sqrt{3} \, \%i}{2} - 1 \frac{\sqrt{3} \, \%i}{2} + 1 \\
& \frac{3}{2} \frac{\sqrt{4 \, a + 27}}{6 \, \sqrt{3}} + \frac{1}{2} \frac{1}{3} \\
\text{(%i7)} \quad & \text{solve } (x^6 - 1); \\
& -\frac{\sqrt{3} \, \%i}{2} + 1 -\frac{\sqrt{3} \, \%i}{2} - 1 \\
& = -\frac{\sqrt{3} \, \%i}{2} + 1 -\frac{\sqrt{3} \, \%i}{2} - 1 \\
\end{align*} \]
Chapter 21: Equations

\[
\frac{(\sqrt{3} \, i + 1)}{64} - 1
\]

\(\%o8\)

\(\%o9\)

\(\%o10\)

\(\%o11\)

\(\%o12\)

The symbols \(\%r\) are used to denote arbitrary constants in a solution.

\(\%i1\)

\(\%o1\)

See [algsys], page 285, and See [\%rnum_list], page 285, for more information.

\textbf{solvedecomposes} \hspace{2cm} \textbf{Option variable}

Default value: true

When \texttt{solvedecomposes} is true, \texttt{solve} calls \texttt{polydecomp} if asked to solve polynomials.

\textbf{solveexplicit} \hspace{2cm} \textbf{Option variable}

Default value: false

When \texttt{solveexplicit} is true, inhibits \texttt{solve} from returning implicit solutions, that is, solutions of the form \(F(x) = 0\) where \(F\) is some function.

\textbf{solvefactors} \hspace{2cm} \textbf{Option variable}

Default value: true

When \texttt{solvefactors} is false, \texttt{solve} does not try to factor the expression. The \texttt{false} setting may be desired in some cases where factoring is not necessary.

\textbf{solvenullwarn} \hspace{2cm} \textbf{Option variable}

Default value: true

When \texttt{solvenullwarn} is true, \texttt{solve} prints a warning message if called with either a null equation list or a null variable list. For example, \texttt{solve ([], [])} would print two warning messages and return [].

\textbf{solveradcan} \hspace{2cm} \textbf{Option variable}

Default value: false

When \texttt{solveradcan} is true, \texttt{solve} calls \texttt{radcan} which makes \texttt{solve} slower but will allow certain problems containing exponentials and logarithms to be solved.
\textbf{solvetrigwarn}  \hspace{1cm} \textbf{Option variable}

Default value: \texttt{true}

When \texttt{solvetrigwarn} is \texttt{true}, \texttt{solve} may print a message saying that it is using inverse trigonometric functions to solve the equation, and thereby losing solutions.
22 Differential Equations

22.1 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.

22.2 Functions and Variables for Differential Equations

**bc2** \((solution, xval1, yval1, xval2, yval2)\)

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 \(x = x_1\), and **yval1** gives the value of the dependent variable in that point, in the form \(y = y_1\). 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** \((eqn, x)\)

The function **desolve** solves systems of linear ordinary differential equations using Laplace transform. Here the **eqn**'s are differential equations in the dependent variables \(x_1, ..., x_n\). The functional dependence of \(x_1, ..., x_n\) 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:

\[
\begin{align*}
  \text{eqn}_1: & \quad \text{diff}(f(x),x,2) = \sin(x) + \text{diff}(g(x)); \\
  \text{eqn}_2: & \quad \text{diff}(f(x),x) + x^2 - f = 2*\text{diff}(g(x),x,2);
\end{align*}
\]

The correct way would be:

\[
\begin{align*}
  \text{eqn}_1: & \quad \text{diff}(f(x),x,2) = \sin(x) + \text{diff}(g(x),x); \\
  \text{eqn}_2: & \quad \text{diff}(f(x),x) + x^2 - f(x) = 2*\text{diff}(g(x),x,2);
\end{align*}
\]

The call to the function **desolve** would then be

\[
\text{desolve([eqn_1, eqn_2], [f(x),g(x)])};
\]

If initial conditions at **x**=0 are known, they can be supplied before calling **desolve** by using atvalue.

\[
\begin{align*}
  (%i1) & \quad \text{diff}(f(x),x)='\text{diff}(g(x),x)+\sin(x); \\
  \quad \text{d} & \quad \text{d} \\
  (%o1) & \quad \frac{\text{d}}{\text{dx}} (f(x)) = -\frac{\text{d}}{\text{dx}} (g(x)) + \sin(x) \\
  (%i2) & \quad \text{diff}(g(x),x,2)='\text{diff}(f(x),x)-\cos(x);
\end{align*}
\]
\[
\frac{2}{dx} \quad (g(x)) = \frac{d}{dx} \quad (f(x)) - \cos(x)
\]

\[(%i3) \quad \text{atvalue('diff(g(x),x),x=0,a);}
\]
\[(%o3) \quad a
\]
\[(%i4) \quad \text{atvalue(f(x),x=0,1);}
\]
\[(%o4) \quad 1
\]
\[(%i5) \quad \text{desolve([%o1,%o2],[f(x),g(x)]);}
\]
\[(%o5) \quad f(x) = a \, e^x - a + 1, \quad g(x) = x \cos(x) + a \, e^x - a + g(0) - 1
\]

\[(%i6) \quad [%o1,%o2],%o5,\text{diff};
\]
\[(%o6) \quad \left[ a \, e^x = a \, e^x, \quad a \, e^x - \cos(x) = a \, e^x - \cos(x) \right]
\]

If \text{desolve} cannot obtain a solution, it returns \text{false}.

\textbf{ic1 \quad (solution, xval, yval)}

Solves initial value problems for first order differential equations. Here \textit{solution} is a general solution to the equation, as found by \texttt{ode2}, \textit{xval} gives an initial value for the independent variable in the form \textit{x} = \textit{x0}, and \textit{yval} gives the initial value for the dependent variable in the form \textit{y} = \textit{y0}.

See \texttt{ode2} for an example of its usage.

\textbf{ic2 \quad (solution, xval, yval, dval)}

Solves initial value problems for second-order differential equations. Here \textit{solution} is a general solution to the equation, as found by \texttt{ode2}, \textit{xval} gives the initial value for the independent variable in the form \textit{x} = \textit{x0}, \textit{yval} gives the initial value of the dependent variable in the form \textit{y} = \textit{y0}, and \textit{dval} gives the initial value for the first derivative of the dependent variable with respect to independent variable, in the form \texttt{diff(y,x)} = \textit{dy0} (\texttt{diff} does not have to be quoted).

See \texttt{ode2} for an example of its usage.

\textbf{ode2 \quad (eqn, dvar, ivar)}

The function \texttt{ode2} solves an ordinary differential equation (ODE) of first or second order. It takes three arguments: an ODE given by \textit{eqn}, the dependent variable \textit{dvar}, and the independent variable \textit{ivar}. When successful, it returns either an explicit or implicit solution for the dependent variable. \texttt{%c} is used to represent the integration constant in the case of first-order equations, and \texttt{%k1} and \texttt{%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 \texttt{desolve}, but the independent variable must always be given as the third argument.
If \texttt{ode2} cannot obtain a solution for whatever reason, it returns \texttt{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: \texttt{method} denotes the method of solution used (e.g., \texttt{linear}), \texttt{intfactor} denotes any integrating factor used, \texttt{odeindex} denotes the index for Bernoulli's method or for the generalized homogeneous method, and \texttt{yp} denotes the particular solution for the variation of parameters technique.

In order to solve initial value problems (IVP) functions \texttt{ic1} and \texttt{ic2} are available for first and second order equations, and to solve second-order boundary value problems (BVP) the function \texttt{bc2} can be used.

Example:

\begin{verbatim}
(%i1) x^2*'diff(y,x) + 3*y*x = sin(x)/x;
               2 dy sin(x)
(%o1) x -- + 3 x y = ------
          dx x
(%i2) ode2(%o1,y,x);
 %c - cos(x)
(%o2) y = ------------
      3 x
(%i3) ic1(%o2,x=%pi,y=0);
         cos(x) + 1
(%o3) y = - -----------
         x
(%i4) 'diff(y,x,2) + y*'diff(y,x)^3 = 0;
          2
               dy 3
(%o4) --- + y (--) = 0
               dx
(%i5) ode2(%o4,y,x);
 3
(%o5) y + 6 %k1 y
     ------------ = x + %k2
       6
(%i6) ratsimp(ic2(%o5,x=0,y=0,'diff(y,x)=2));
\end{verbatim}
(\%o6) \frac{2y - 3y}{6} = x

(\%i7) bc2(\%o6, x=0, y=1, x=1, y=3);

(\%o7) \frac{3y - 10y}{6} = x - \frac{3}{2}
Chapter 23: Numerical

23 Numerical

23.1 Introduction to fast Fourier transform

The \texttt{fft} package comprises functions for the numerical (not symbolic) computation of the fast Fourier transform.

23.2 Functions and Variables for fast Fourier transform

\textbf{polartorect} \((r, t)\)  
\begin{align*}
\text{Function} & \\
\text{Translates complex values of the form } & r \ e^{\text{i} t} \text{ to the form } a + b \ \text{i}, \text{ where } r \text{ is} \\
\text{the magnitude and } t \text{ is the phase. } r \text{ and } t \text{ are } & 1\text{-dimensional arrays of the same size.} \\
\text{The array size need not be a power of } & 2. \\
\text{The original values of the input arrays are replaced by the real and imaginary parts,} & a \text{ and } b, \text{ on return. The outputs are calculated as} \\
& a = r \ \cos(t) \\
& b = r \ \sin(t) \\
\text{polartorect is the inverse function of recttopolar.} & \\
\text{load(fft) loads this function. See also fft.} & \\
\end{align*}

\textbf{recttopolar} \((a, b)\)  
\begin{align*}
\text{Function} & \\
\text{Translates complex values of the form } a + b \ \text{i} \text{ to the form } r \ e^{\text{i} t}, \text{ where } a \text{ is} \\
\text{the real part and } b \text{ is the imaginary part. } a \text{ and } b \text{ are } 1\text{-dimensional arrays of the same size.} \\
\text{The array size need not be a power of } & 2. \\
\text{The original values of the input arrays are replaced by the magnitude and angle, } r \text{ and } t, \text{ on return. The outputs are calculated as} \\
& r = \sqrt{a^2 + b^2} \\
& t = \text{atan2}(b, a) \\
\text{The computed angle is in the range } & -\pi \text{ to } \pi. \\
\text{recttopolar is the inverse function of polartorect.} & \\
\text{load(fft) loads this function. See also fft.} & \\
\end{align*}

\textbf{inverse_fft} \((y)\)  
\begin{align*}
\text{Function} & \\
\text{Computes the inverse complex fast Fourier transform. } y \text{ is a list or array (named} \\
\text{or unnamed) which contains the data to transform. The number of elements must} & \text{be a power of } 2. \\
\text{The elements must be literal numbers (integers, rationals, floats,} & \text{or bigfloats) or symbolic constants, or expressions } a + b*\text{i} \text{ where } a \text{ and } b \text{ are literal} \\
\text{numbers or symbolic constants.} & \\
\text{inverse_fft returns a new object of the same type as } y, \text{ which is not modified.} \\
\text{Results are always computed as floats or expressions } a + b*\text{i} \text{ where } a \text{ and } b \text{ are} & \text{floats.} \\
\text{The inverse discrete Fourier transform is defined as follows. Let } x \text{ be the output of} & \text{the inverse transform. Then for } j \text{ from } 0 \text{ through } n - 1, \\
\end{align*}
\[ x[j] = \sum (y[k] \exp(+2 \pi i j k / n), k, 0, n - 1) \]

load(fft) loads this function.

See also fft (forward transform), recttopolar, and polartorect.

Examples:

Real data.

\begin{verbatim}
(%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
\end{verbatim}

Complex data.

\begin{verbatim}
(%i1) load (fft) $
(%i2) fpprintprec : 4 $
(%i3) L : [1, 1 + %i, 1 - %i, -1, 1 - %i, 1 + %i, 1] $
(%i4) L1 : inverse_fft (L);
(%o4) [4.0, 2.711L-19 %i + 4.0, 1.0 %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
\end{verbatim}

fft (x)

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.

The discrete Fourier transform is defined as follows. Let \( y \) be the output of the transform. Then for \( k \) from 0 through \( n - 1 \),

\[ y[k] = (1/n) \sum (x[j] \exp(-2 \pi i j k / n), j, 0, n - 1) \]

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
a[0] = realpart (y[0])
b[0] = 0
and, for k from 1 through n/2 - 1,
a[k] = realpart (y[k] + y[n - k])
b[k] = imagpart (y[n - k] - y[k])
and
a[n/2] = realpart (y[n/2])
b[n/2] = 0

load(fft) loads this function.
See also inverse_fft (inverse 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 : fft (L);
(%o4) \[0.0, -1.811 \times 10^3, 0.0, 0.6036 - 0.3107 \times 10^1, 0.0, 
0.3107 \times 10^1 + 0.6036, 0.0, 1.811 \times 10^3 - 10^3\]
(%i5) L2 : inverse_fft (L1);
(%o5) \[1.0, 2.168 \times 10^{-19} \times 10^1 + 2.0, 7.525 \times 10^{-20} \times 10^3 + 3.0, 
4.256 \times 10^{-19} \times 10^3 + 4.0, -1.0, -2.168 \times 10^{-19} \times 10^3 - 2.0, 
-7.525 \times 10^{-20} \times 10^3 - 3.0, -4.256 \times 10^{-19} \times 10^3 - 4.0\]
(%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) \[0.5, 0.3536 \times 10^1 + 0.3536, -0.25 \times 10^1 - 0.25, 
0.5 - 6.776 \times 10^{-21} \times 10^1, 0.0, -0.3536 \times 10^1 - 0.3536, 0.25 \times 10^1 - 0.25, 
0.5 - 3.388 \times 10^{-20} \times 10^n\]
(%i5) L2 : inverse_fft (L1);
(%i5) L2 : inverse_fft (L1);
(%o5) \[1.0 - 1.947 \times 10^{-20} \times 10^1, 1.0 \times 10^1 + 1.0, 1.0 - 1.0 \times 10^1, 
-1.008 \times 10^{-19} \times 10^1 - 1.0, 4.066 \times 10^{-20} \times 10^1 - 1.0, 1.0 - 1.0 \times 10^1, 
1.0 \times 10^1 + 1.0, 1.947 \times 10^{-20} \times 10^1 + 1.0\]
(%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]

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 ^ 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 forindent causes expressions to be printed further to the right.

When forspaces 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.

Examples:

(%i11) expr: (a + b)^12$
(%i12) fortran (expr);
Chapter 23: Numerical

(b+a)**12  \[ \text{done} \]

(\%i3) fortran ('x=expr);
\[
\begin{align*}
&x = (b+a)**12 \\
&\text{done}
\end{align*}
\]

(\%o3) fortran ('x=expand (expr));
\[
\begin{align*}
&x = b**12+12*a*b**11+66*a**2*b**10+220*a**3*b**9+495*a**4*b**8+792*a**5*b**7+924*a**6*b**6+792*a**7*b**5+495*a**8*b**4+220*a**9*b**3+66*a**10*b**2+12*a**11*b+a**12 \\
&\text{done}
\end{align*}
\]

(\%i4) fortran ('x=7+5*%i);
\[
\begin{align*}
&x = (7,5) \\
&\text{done}
\end{align*}
\]

(\%o4) fortran ('x=[1,2,3,4]);
\[
\begin{align*}
&x = [1,2,3,4] \\
&\text{done}
\end{align*}
\]

(\%i5) f(x) := x^2$
\[
\begin{align*}
&f \\
&\text{done}
\end{align*}
\]

\textbf{fortspaces}  \hspace{1cm} \text{Option variable}

Default value: false

When \text{fortspaces} is true, \text{fortran} fills out each printed line with spaces to 80 columns.

\textbf{horner} (\textit{expr}, \textit{x}) \hspace{1cm} \text{Function}

\textbf{horner} (\textit{expr}) \hspace{1cm} \text{Function}

Returns a rearranged representation of \textit{expr} as in Horner’s rule, using \textit{x} as the main variable if it is specified. \textit{x} may be omitted in which case the main variable of the canonical rational expression form of \textit{expr} is used.

\textbf{horner} sometimes improves stability if \textit{expr} is to be numerically evaluated. It is also useful if Maxima is used to generate programs to be run in Fortran. See also \textbf{stringout}.

(\%i11) expr: 1e-155*x^2 - 5.5*x + 5.2e155;
\[
\begin{align*}
&1e-155 x^2 - 5.5 x + 5.2e155 \\
&\text{2}
\end{align*}
\]

(\%o1) 1.0E-155 x - 5.5 x + 5.2E+155

(\%i12) expr2: horner (%), keepfloat: true;
\[
\begin{align*}
&\text{horner } \%, \text{ keepfloat: true;} \\
&\text{2}
\end{align*}
\]

(\%o2) (1.0E-155 x - 5.5) x + 5.2E+155

(\%i13) ev (expr, x=1e155);
Maxima encountered a Lisp error:

\text{floating point overflow}

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.

(\%i14) ev (expr2, x=1e155);
\[
\begin{align*}
&7.0E+154
\end{align*}
\]

(\%o4) 7.0E+154
Function \texttt{find_root}(\texttt{expr, x, a, b, [abserr, relerr]})

Function \texttt{find_root}(\texttt{f, a, b, [abserr, relerr]})

Function \texttt{bf_find_root}(\texttt{expr, x, a, b, [abserr, relerr]})

Function \texttt{bf_find_root}(\texttt{f, a, b, [abserr, relerr]})

Option variable \texttt{find_root_error}

Option variable \texttt{find_root_abs}

Option variable \texttt{find_root_rel}

\texttt{find_root} finds a root of the expression \texttt{expr} or the function \texttt{f} over the closed interval \([a, b]\). The expression \texttt{expr} may be an equation, in which case \texttt{find_root} seeks a root of \texttt{lhs(expr) - rhs(expr)}.

Given that Maxima can evaluate \texttt{expr} or \texttt{f} over \([a, b]\) and that \texttt{expr} or \texttt{f} is continuous, \texttt{find_root} is guaranteed to find the root, or one of the roots if there is more than one.

\texttt{find_root} initially applies binary search. If the function in question appears to be smooth enough, \texttt{find_root} applies linear interpolation instead.

\texttt{bf_find_root} is a bigfloat version of \texttt{find_root}. The function is computed using bigfloat arithmetic and a bigfloat result is returned. Otherwise, \texttt{bf_find_root} is identical to \texttt{find_root}, and the following description is equally applicable to \texttt{bf_find_root}.

The accuracy of \texttt{find_root} is governed by \texttt{abserr} and \texttt{relerr}, which are optional keyword arguments to \texttt{find_root}. These keyword arguments take the form \texttt{key=val}.

The keyword arguments are

\begin{itemize}
  \item \texttt{abserr} Desired absolute error of function value at root. Default is \texttt{find_root_abs}.
  \item \texttt{relerr} Desired relative error of root. Default is \texttt{find_root_rel}.
\end{itemize}

\texttt{find_root} stops when the function in question evaluates to something less than or equal to \texttt{abserr}, or if successive approximants \texttt{x_0, x_1} differ by no more than \texttt{relerr * max(abs(x_0), abs(x_1))}. The default values of \texttt{find_root_abs} and \texttt{find_root_rel} are both zero.

\texttt{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 \texttt{find_root} is governed by \texttt{find_root_error}. When \texttt{find_root_error} is \texttt{true}, \texttt{find_root} prints an error message. Otherwise \texttt{find_root} returns the value of \texttt{find_root_error}. The default value of \texttt{find_root_error} is \texttt{true}.

If \texttt{f} evaluates to something other than a number at any step in the search algorithm, \texttt{find_root} returns a partially-evaluated \texttt{find_root} expression.

The order of \texttt{a} and \texttt{b} is ignored; the region in which a root is sought is \([\texttt{min(a, b), max(a, b)}]\).

Examples:

\begin{verbatim}
(%i1) f(x) := sin(x) - x/2;
x
(%o1) f(x) := sin(x) - ----
  2
\end{verbatim}
Chapter 23: Numerical

(%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);
   (%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) 2.30258509299404643852b0
(%i11) log(10b0);
   (%o11) 2.30258509299404643852b0

Function

newton (expr, x, x_0, eps)

Returns an approximate solution of \( expr = 0 \) by Newton’s method, considering \( expr \) to be a function of one variable, \( x \). The search begins with \( x = x_0 \) and proceeds until \( \text{abs}(expr) < eps \) (with \( expr \) evaluated at the current value of \( x \)).

newton allows undefined variables to appear in \( expr \), so long as the termination test \( \text{abs}(expr) < eps \) 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 mnewton.

Examples:

(%i1) load (newton1);
(%o1) /usr/share/maxima/5.10.0cvs/share/numeric/newton1.mac
(%i2) newton (cos (u), u, 1, 1/100);
   (%o2) 1.570675277161251
(%i3) ev (cos (u), u = %);
   (%o3) 1.210496333503352E-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
23.3 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.

23.4 Functions and Variables for Fourier series

equalp (x, y)
Returns true if equal (x, y) otherwise false (doesn’t give an error message like equal (x, y) would do in this case).

remfun (f, expr)
remfun (f, expr, x) 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.

fupn (f, expr)
fupn (f, expr, x) returns true if expr contains the function f.
fupn (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 (f, x, halfplane)
absint (f, x) 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)
Returns a list of the Fourier coefficients of f(x) defined on the interval [-p, p].

foursimp (l)
Simplifies sin (n %pi) to 0 if sinnpiflag is true and cos (n %pi) to (-1)^n if cosnppiflag is true.

sinppiflag
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 \(\text{fourexpand (foursimp (fourier (f, x, p)), x, p, 'inf)}\).

**fourint** \((f, x)\)

Function

Constructs and returns a list of the Fourier integral coefficients of \(f(x)\) defined on \([\text{minf, inf}]\).

**fourintcos** \((f, x)\)

Function

Returns the Fourier cosine integral coefficients for \(f(x)\) on \([0, \text{inf}]\).

**fourintsin** \((f, x)\)

Function

Returns the Fourier sine integral coefficients for \(f(x)\) on \([0, \text{inf}]\).
24 Arrays

24.1 Functions and Variables for Arrays

array \( (\text{name}, \text{dim}_1, \ldots, \text{dim}_n) \)  
Function

array \( (\text{name}, \text{type}, \text{dim}_1, \ldots, \text{dim}_n) \)  
Function

array \( ([\text{name}_1, \ldots, \text{name}_m], \text{dim}_1, \ldots, \text{dim}_n) \)  
Function

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 \( \text{dim}_i \).

array \( (\text{name}, \text{dim}_1, \ldots, \text{dim}_n) \) creates a general array.

array \( (\text{name}, \text{type}, \text{dim}_1, \ldots, \text{dim}_n) \) creates an array, with elements of a specified type. \text{type} can be \text{fixnum} for integers of limited size or \text{flonum} for floating-point numbers.

array \( ([\text{name}_1, \ldots, \text{name}_m], \text{dim}_1, \ldots, \text{dim}_n) \) creates \( m \) arrays, all of the same dimensions.

If the user assigns to a subscripted variable before declaring the corresponding array, an undeclared array is created. Undeclared arrays, otherwise known as hashed arrays (because hash coding is done on the subscripts), are more general than declared arrays. The user does not declare their maximum size, and they grow dynamically by hashing as more elements are assigned values. The subscripts of undeclared arrays need not even be numbers. However, unless an array is rather sparse, it is probably more efficient to declare it when possible than to leave it undeclared. The array function can be used to transform an undeclared array into a declared array.

arrayapply \( (A, [i_1, \ldots, i_n]) \)  
Function

Evaluates \( A [i_1, \ldots, i_n] \), where \( A \) is an array and \( i_1, \ldots, i_n \) 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, an undeclared (hashed) array, an array function, or a subscripted function.

For declared arrays, arrayinfo returns a list comprising the atom \text{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 \text{hashed}, the number of subscripts, and the subscripts of every element which has a value. The values are returned by listarray.

For array functions, arrayinfo returns a list comprising the atom \text{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 \text{hashed}, the number of subscripts, and any subscript values for which there are lambda expressions. The lambda expressions are returned by listarray.
Examples:

arrayinfo and listarray applied to a declared array.

\[
\begin{align*}
(\%i1) \ & \text{array (aa, 2, 3);} \\
(\%o1) \ & \text{aa} \\
(\%i2) \ & \text{aa [2, 3] : %pi;} \\
(\%o2) \ & \text{%pi} \\
(\%i3) \ & \text{aa [1, 2] : %e;} \\
(\%o3) \ & \text{%e} \\
(\%i4) \ & \text{arrayinfo (aa);} \\
(\%o4) \ & \text{[declared, 2, [2, 3]]} \\
(\%i5) \ & \text{listarray (aa);} \\
(\%o5) \ & \text{[%%%%, %%%%%, %%%%%, %%%%%, %%%%%, %%%%%, %e, %%%%%,}
\text{ %%%%%, %%%%%, %%%%%, %pi]}
\end{align*}
\]

arrayinfo and listarray applied to an undeclared (hashed) array.

\[
\begin{align*}
(\%i1) \ & \text{bb [FOO] : (a + b)^2;} \\
(\%o1) \ & \text{(b + a)} \\
(\%i2) \ & \text{bb [BAR] : (c - d)^3;} \\
(\%o2) \ & \text{(c - d)} \\
(\%i3) \ & \text{arrayinfo (bb);} \\
(\%o3) \ & \text{[hashed, 1, [BAR], [FOO]]} \\
(\%i4) \ & \text{listarray (bb);} \\
(\%o4) \ & \text{[(c - d), (b + a)]}
\end{align*}
\]

arrayinfo and listarray applied to an array function.

\[
\begin{align*}
(\%i1) \ & \text{cc [x, y] := y / x;} \\
(\%o1) \ & \text{cc := \(- y \)} \\
(\%o2) \ & \text{\(x, y \)} \\
(\%i2) \ & \text{cc [u, v];} \\
(\%o2) \ & \text{v} \\
(\%i3) \ & \text{cc [4, z];} \\
(\%o3) \ & \text{z} \\
(\%i4) \ & \text{arrayinfo (cc);} \\
(\%o4) \ & \text{[hashed, 2, [4, z], [u, v]]} \\
(\%i5) \ & \text{listarray (cc);} \\
(\%o5) \ & \text{[z, v [\(-, \)]} \\
(\%o5) \ & \text{4 \ u]}
\end{align*}
\]

arrayinfo and listarray applied to a subscripted function.
Chapter 24: Arrays 319

\[(%i1) \text{dd \[x\] \(y\) := } y \wedge x; \]
\[(%o1) \text{dd \(y\) := } y \wedge x \]

\[(%i2) \text{dd \[a + b\]; } b + a \]
\[(%o2) \lambda(y); \]

\[(%i3) \text{dd \[v - u\]; } v - u \]
\[(%o3) \lambda(y); \]

\[(%i4) \text{arrayinfo (dd);} \]
\[(%o4) \text{[hashed, 1, \[b + a\], \[v - u\]]} \]

\[(%i5) \text{listarray (dd);} \]
\[(%o5) \lambda(y), \lambda(y) \]

\[\text{arraymake}\ (A, [i_1, \ldots, i_n]) \]
\[\text{Function} \]
\[\text{Returns the expression } A[i_1, \ldots, i_n]. \text{ The result is an unevaluated array reference.} \]

\[\text{arraymake} \text{ is reminiscent of } \text{funmake}, \text{ except the return value is an unevaluated array reference instead of an unevaluated function call.} \]

\[\text{Examples:} \]
\[(%i1) \text{arraymake (A, \[1\]); } A \]
\[(%o1) A \]

\[(%i2) \text{arraymake (A, \[k\]); } A \]
\[(%o2) A \]

\[(%i3) \text{arraymake (A, \[i, j, 3\]); } A \]
\[(%o3) A \]

\[(%i4) \text{array (A, fixnum, 10);} \]
\[(%o4) A \]

\[(%i5) \text{fillarray (A, makelist (i\wedge2, i, 1, 11)); } A \]
\[(%o5) A \]

\[(%i6) \text{arraymake (A, \[5\]); } A \]
\[(%o6) A \]

\[(%i7) \]"%; \]
\[(%o7) 36 \]

\[(%i8) L : \[a, b, c, d, e\]; \]
\[(%o8) \[a, b, c, d, e\] \]

\[(%i9) \text{arraymake ('L, \[n\]); } L \]
\[(%o9) L \]

\[(%i10) '"%,' n = 3; \]
\[(%o10) c \]
(\%i11) A2 : make_array (fixnum, 10);
(\%o11) {Array: #(0 0 0 0 0 0 0 0 0 0)}
(\%i12) fillarray (A2, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
(\%o12) {Array: #(1 2 3 4 5 6 7 8 9 10)}
(\%i13) arraymake ('A2, [8]);
(\%o13) A2
(\%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 constructed by implicit definition (assigning something to an array element), and array 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 [FOO] : (a + b)^2;
(\%o2) (b + a)
(\%i3) cc [x] := x/100;
(\%o3) cc := \frac{x}{100}
(\%i4) dd : make_array ('any, 7);
(\%o4) {Array: #(NIL NIL NIL NIL NIL NIL NIL)}
(\%i5) arrays;
(\%o5) [aa, bb, cc]

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.

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.

```lisp
(%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.

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

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

```lisp
(%i1) a3 : make_array (fixnum, 2, 5);  
(%o1) {Array: #2A((0 0 0 0 0) (0 0 0 0 0))}
(%i2) fillarray (a3, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
(%o2) {Array: #2A((1 2 3 4 5) (6 7 8 9 10))}
(%i3) a4 : make_array (fixnum, 5, 2);  
(%o3) {Array: #2A((0 0) (0 0) (0 0) (0 0))}
(%i4) fillarray (a4, a3);
(%o4) {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 a declared array, an undeclared (hashed) array, an array 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, array functions, and subscripted functions, the elements correspond to the index values returned by `arrayinfo`.

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

Examples:

listarray and arrayinfo applied to a declared array.

```lisp
(%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) [#####, #####, #####, #####, #####, %pi, #####, #####, #####, %pi]
(%i5) arrayinfo (aa);
(%o5) [declared, 2, [2, 3]]
```

listarray and arrayinfo applied to an undeclared (hashed) array.

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

listarray and arrayinfo applied to an array function.

```lisp
(%i1) cc [x, y] := y / x;
(%o1) cc := -
x, y x
(%i2) cc [u, v];
(%o2) -
v
(%i3) cc [4, z];
(%o3) -
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.
Chapter 24: Arrays

Function

make_array (type, dim_1, ..., dim_n)

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) {Array: #(0 0 0 0 0 0 0 0 0 0)}
(%o2) 1729
(%i3) A1;
(%o3) {Array: #(0 0 0 0 0 0 0 0 1729 0)}
(%i4) A2 : make_array (flonum, 10);
(%o4) {Array: #(0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0)}
(%o5) 2.718281828
(%i6) A2;
(%o6) {Array: #(0.0 0.0 2.718281828 0.0 0.0 0.0 0.0 0.0 0.0 0.0)}
(%i7) A3 : make_array (any, 10);
(%o7) {Array: #(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)}
(%i8) A3 [4] : x - y - z;
(%o8) - z - y + x
(%i9) A3;
(%o9) {Array: #(NIL NIL NIL NIL ((MPLUS SIMP) $X ((MTIMES SIMP) -1 $Y) ((MTIMES SIMP) -1 $Z)) NIL NIL NIL NIL NIL)}
(%i10) A4 : make_array (fixnum, 2, 3, 5);
(%o10) \{Array: #3A(((0 0 0 0 0) (0 0 0 0 0) (0 0 0 0 0)) ((0 0 \n0 0 0) (0 0 0 0 0) (0 0 0 0 0))))\}
(%i11) fillarray (A4, makelist (i, i, 1, 2*3*5));
(%o11) \{Array: #3A(((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15))
((16 17 18 19 20) (21 22 23 24 25) (26 27 28 29 30))))\}
(%i12) A4 [0, 2, 1];
(%o12) 12

rearray (A, dim_1, ..., dim_n)

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

Removes arrays and array associated functions and frees the storage occupied. The
arguments may be declared arrays, undeclared (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 redefine the values in a
hashed array.

remarray returns the list of arrays removed.

remarray quotes its arguments.

subvar (x, i)

Evaluates the subscripted expression x[i].

Examples:

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

- if true then only two types of arrays are recognized.

1) The art-q array (t in Common Lisp) which may have several dimensions indexed by integers, and may hold any Lisp or Maxima object as an entry. To construct such an array, enter `a:make_array(any,3,4);` then `a` will have as value, an array with twelve slots, and the indexing is zero based.

2) The Hash table array which is the default type of array created if one does `b[x+1]:y^2` (and `b` is not already an array, a list, or a matrix – if it were one of these an error would be caused since `x+1` would not be a valid subscript for an art-q array, a list or a matrix). Its indices (also known as keys) may be any object. It only takes one key at a time (`b[x+1,u]:y` would ignore the `u`). Referencing is done by `b[x+1] => y^2`. Of course the key may be a list, e.g. `b[[x+1,u]]:y` would be valid. This is incompatible with the old Maxima hash arrays, but saves consing.

An advantage of storing the arrays as values of the symbol is that the usual conventions about local variables of a function apply to arrays as well. The Hash table type also uses less consing and is more efficient than the old type of Maxima hashar. To obtain consistent behaviour in translated and compiled code set `translate_fast_arrays` to be true.
25 Matrices and Linear Algebra

25.1 Introduction to Matrices and Linear Algebra

25.1.1 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 \( \text{sum}(a[i] \cdot b[i], i, 1, \text{length}(a)) \). 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 \( \text{conjugate}(a) . b \) when \( a \) and \( b \) are complex; \text{innerproduct} in the \text{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 \text{dot}, \text{dot0nscsimp}, \text{dot0simp}, \text{dot1simp}, \text{dotassoc}, \text{dotconstrules}, \text{dotdistrib}, \text{dotexptsimp}, \text{dotident}, and \text{dotscrules}.

25.1.2 Vectors

\text{vect} is a package of functions for vector analysis. \text{load} ("vect") loads this package, and \text{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 \text{vect} package contains these functions: \text{vectorsimp}, \text{scalefactors}, \text{express}, \text{potential}, and \text{vectorpotential}.

By default the \text{vect} package does not declare the dot operator to be a commutative operator. To get a commutative dot operator \( . \), the command \text{declare}(".","\text{commutative}\) must be executed.

25.1.3 \text{eigen}

The package \text{eigen} contains several functions devoted to the symbolic computation of eigenvalues and eigenvectors. Maxima loads the package automatically if one of the functions \text{eigenvalues} or \text{eigenvectors} is invoked. The package may be loaded explicitly as \text{load} ("\text{eigen}"").
The functions in the \texttt{eigen} package are \texttt{innerproduct}, \texttt{unitvector}, \texttt{columnvector}, \texttt{gramschmidt}, \texttt{eigenvalues}, \texttt{eigenvectors}, \texttt{uniteigenvectors}, and \texttt{similaritytransform}.

\section*{25.2 Functions and Variables for Matrices and Linear Algebra}

\begin{itemize}
  \item \texttt{addcol} \((M, \text{list}_1, \ldots, \text{list}_n)\) \hspace{1cm} \text{Function}\]
  \begin{itemize}
    \item Appends the column(s) given by the one or more lists (or matrices) onto the matrix \(M\).
  \end{itemize}

  \item \texttt{addrow} \((M, \text{list}_1, \ldots, \text{list}_n)\) \hspace{1cm} \text{Function}\]
  \begin{itemize}
    \item Appends the row(s) given by the one or more lists (or matrices) onto the matrix \(M\).
  \end{itemize}

  \item \texttt{adjoint} \((M)\) \hspace{1cm} \text{Function}\]
  \begin{itemize}
    \item Returns the adjoint of the matrix \(M\). The adjoint matrix is the transpose of the matrix of cofactors of \(M\).
  \end{itemize}

  \item \texttt{augcoefmatrix} \(([\text{eqn}_1, \ldots, \text{eqn}_m], [\text{x}_1, \ldots, \text{x}_n])\) \hspace{1cm} \text{Function}\]
  \begin{itemize}
    \item Returns the augmented coefficient matrix for the variables \(x_1, \ldots, x_n\) of the system of linear equations \(\text{eqn}_1, \ldots, \text{eqn}_m\). This is the coefficient matrix with a column adjoined for the constant terms in each equation (i.e., those terms not dependent upon \(x_1, \ldots, x_n\)).
  \end{itemize}

\end{itemize}

\begin{verbatim}
(%i1) m: [2*x - (a - 1)*y = 5*b, c + b*y + a*x = 0]
(%i2) augcoefmatrix (m, [x, y]);
[ 2 1 - a - 5 b ]
[ a b c ]
\end{verbatim}

\begin{itemize}
  \item \texttt{cauchy_matrix} \(([x_1,x_2, \ldots, x_m], [y_1,y_2, \ldots, y_n])\) \hspace{1cm} \text{Function}\]
  \begin{itemize}
    \item Returns a \(n \times m\) Cauchy matrix with the elements \(a[i,j] = 1/(x_i+y_j)\). The second argument of \texttt{cauchy_matrix} is optional. For this case the elements of the Cauchy matrix are \(a[i,j] = 1/(x_i+y_j)\).
  \end{itemize}

  \item \texttt{cauchy_matrix} \(([x_1,x_2, \ldots, x_n])\) \hspace{1cm} \text{Function}\]
  \begin{itemize}
    \item Returns a \(n \times n\) Cauchy matrix with the elements \(a[i,j] = 1/(x_i+y_j)\). The second argument of \texttt{cauchy_matrix} is optional. For this case the elements of the Cauchy matrix are \(a[i,j] = 1/(x_i+y_j)\).
  \end{itemize}

\end{itemize}

\begin{verbatim}
(%i1) cauchy_matrix([x1,x2],[y1,y2]);
\end{verbatim}
```plaintext
(%i1) cauchy_matrix([x1,x2]);
[ 1 1 ]
[ ------- ------- ]
[ y1 + x1 y2 + x1 ]
(%o1)

(%i2) cauchy_matrix([x1,x2]);
[ 1 1 ]
[ ------- ------- ]
[ y1 + x2 y2 + x2 ]
(%o2)

charpoly (M, x)
Returns the characteristic polynomial for the matrix M with respect to variable x.
That is, determinant (M - diagmatrix (length (M), x)).

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

(%i2) expand (charpoly (a, lambda));
2
(%o2) lambda - 7 lambda + 10

(%i3) (programmode: true, solve (%));
(%o3) [lambda = 5, lambda = 2]

(%i4) matrix ([x1], [x2]);
(%o4)

(%i5) ev (a . % - lambda*%, %th(2)[1]);
(%o5)

(%i6) %[1, 1] = 0;
(%o6)

(%i7) x2^2 + x1^2 = 1;
(%i8) solve ([%th(2), %], [x1, x2]);
(%o8) [[x1 = - -------, x2 = - -------],
     sqrt(5)  sqrt(5)]

(%o9) [x1 = -------, x2 = -------]
     sqrt(5)  sqrt(5)
```

```
[ 1 1 ]
[ ------- ------- ]
[ y1 + x1 y2 + x1 ]

([%i1]
[ 1 1 ]
[ ------- ------- ]
[ y1 + x2 y2 + x2 ]

(%i2) cauchy_matrix([x1,x2]);

(%i2) cauchy_matrix([x1,x2]);

charpoly (M, x)

Returns the characteristic polynomial for the matrix M with respect to variable x.

That is, determinant (M - diagmatrix (length (M), x)).

(%i1) a: matrix ([3, 1], [2, 4]);

(%o1)

(%i2) expand (charpoly (a, lambda));

2

(%o2) lambda - 7 lambda + 10

(%i3) (programmode: true, solve (%));

(%o3) [lambda = 5, lambda = 2]

(%i4) matrix ([x1], [x2]);

(%o4)

(%i5) ev (a . % - lambda*%, %th(2)[1]);

(%o5)

(%i6) %[1, 1] = 0;

(%o6)

(%i7) x2^2 + x1^2 = 1;

(%i8) solve ([%th(2), %], [x1, x2]);

(%o8) [[x1 = - -------, x2 = - -------],
     sqrt(5)  sqrt(5)]

(%o9) [x1 = -------, x2 = -------]
     sqrt(5)  sqrt(5)
```
Function

**coefmatrix** ([eqn_1, ..., eqn_m], [x_1, ..., x_n])

Returns the coefficient matrix for the variables x_1, ..., x_n of the system of linear equations eqn_1, ..., eqn_m.

\[
\begin{bmatrix}
2 & 1 - a \\
\end{bmatrix}
\]

Function

**col** (M, i)

Returns the i'th column of the matrix M. The return value is a matrix.

Function

**columnvector** (L)

Function

**covect** (L)

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 ]
[ ]
[ cc ]
[ ]
[ dd ]
```

Function

**conjugate** (x)

Returns the complex conjugate of x.

```
(%i1) declare ([aa, bb], real, cc, complex, ii, imaginary); done
(%i2) conjugate (aa + bb*%i);
%o2)
(%i3) conjugate (cc);
(%i4) conjugate (ii);
(%o4)
```

- ii
(%i5) conjugate (xx + yy);  
(%o5)     
    conjugate(yy) + conjugate(xx)

**copymatrix** \((M)\)

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 `setelmx (x, i, j, m2)` also modifies \(m1 [i,j]\). Creating a copy with `copymatrix` and then using assignment creates a separate, modified copy.

**determinant** \((M)\)

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:

```lisp
(%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** \((n, x)\)

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 \( \exp (\text{ev}(M)) \).

domxexpt affects all expressions of the form \( \text{base}^\text{power} \) where \( \text{base} \) is an expression assumed scalar or constant, and power is a list or matrix.

Example:

\[
\begin{align*}
(\%i1) \ m & : \text{matrix} ([1, \%i], [a+b, \%pi]); \\
(\%o1) \ & \begin{bmatrix} 1 & \%i \\ b+a & \%pi \end{bmatrix} \\
(\%i2) \ \text{domxexpt} : \text{false}$
(\%i3) \ (1 - c)^m; \\
(\%o3) \ & \begin{bmatrix} 1 & \%i \\ b+a & \%pi \end{bmatrix} \\
(\%i4) \ \text{domxexpt} : \text{true}$
(\%i5) \ (1 - c)^m;
(\%o5) \ & \begin{bmatrix} \%i \\ 1 - c & (1 - c) \\ b+a & \%pi \\ (1 - c) & (1 - c) \end{bmatrix}
\end{align*}
\]

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 \texttt{dontfactor} list.

\textbf{doscmxops} \\
Option variable \hspace{1cm} Default value: \texttt{false} \hspace{1cm} When \texttt{doscmxops} is \texttt{true}, scalar-matrix operations are carried out.

\textbf{doscmxplus} \\
Option variable \hspace{1cm} Default value: \texttt{false} \hspace{1cm} When \texttt{doscmxplus} is \texttt{true}, scalar-matrix operations yield a matrix result. This switch is not subsumed under \texttt{doallmxops}.

\textbf{dot0nscsimp} \\
Option variable \hspace{1cm} Default value: \texttt{true} \hspace{1cm} When \texttt{dot0nscsimp} is \texttt{true}, a non-commutative product of zero and a nonscalar term is simplified to a commutative product.

\textbf{dot0simp} \\
Option variable \hspace{1cm} Default value: \texttt{true} \hspace{1cm} When \texttt{dot0simp} is \texttt{true}, a non-commutative product of zero and a scalar term is simplified to a commutative product.

\textbf{dot1simp} \\
Option variable \hspace{1cm} Default value: \texttt{true} \hspace{1cm} When \texttt{dot1simp} is \texttt{true}, a non-commutative product of one and another term is simplified to a commutative product.

\textbf{dotassoc} \\
Option variable \hspace{1cm} Default value: \texttt{true} \hspace{1cm} When \texttt{dotassoc} is \texttt{true}, an expression \((A.B).C\) simplifies to \(A.(B.C)\).

\textbf{dotconstrules} \\
Option variable \hspace{1cm} Default value: \texttt{true} \hspace{1cm} When \texttt{dotconstrules} is \texttt{true}, a non-commutative product of a constant and another term is simplified to a commutative product. Turning on this flag effectively turns on \texttt{dot0simp}, \texttt{dot0nscsimp}, and \texttt{dot1simp} as well.

\textbf{dotdistrib} \\
Option variable \hspace{1cm} Default value: \texttt{false} \hspace{1cm} When \texttt{dotdistrib} is \texttt{true}, an expression \(A.(B + C)\) simplifies to \(A.B + A.C\).

\textbf{dotexptsimp} \\
Option variable \hspace{1cm} Default value: \texttt{true} \hspace{1cm} When \texttt{dotexptsimp} is \texttt{true}, an expression \(A.A\) simplifies to \(A^{\wedge}2\).
Option variable \textit{dotident}

Default value: 1

\textit{dotident} is the value returned by $X^{\times 0}$.

Option variable \textit{dotscrules}

Default value: \texttt{false}

When \textit{dotscrules} is \texttt{true}, an expression $A.SC$ or $SC.A$ simplifies to $SC\times A$ and $A.(SC\times B)$ simplifies to $SC\times (A.B)$.

Function \texttt{echelon}\hspace{1em}(\textit{M})

Returns the echelon form of the matrix \textit{M}, as produced by Gaussian elimination. The echelon form is computed from \textit{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.

\texttt{triangularize}\ also carries out Gaussian elimination, but it does not normalize the leading non-zero element in each row.

\texttt{lu_factor} and \texttt{cholesky} are other functions which yield triangularized matrices.

\begin{verbatim}
(%i1) M: matrix ([3, 7, aa, bb], [-1, 8, 5, 2], [9, 2, 11, 4]);

(%o1) [ 3 7 aa bb ]
    [ -1 8 5 2 ]
    [ 9 2 11 4 ]

(%i2) echelon (M);

(%o2) [ 1 - 8 - 5 - 2 ]
    [ ]
    [ 28 11 ]
    [ 0 1 -- -- ]
    [ ]
    [ 37 37 ]
    [ ]
    [ 37 bb - 119 ]
    [ 0 0 1 ---- ]
    [ ]
    [ 37 aa - 313 ]
\end{verbatim}

Function \texttt{eigenvalues}\hspace{1em}(\textit{M})

\texttt{eivals} \hspace{1em}(\textit{M})

Returns a list of two lists containing the eigenvalues of the matrix \textit{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.

\texttt{eivals} is a synonym for \texttt{eigenvalues}.

\texttt{eigenvalues} calls the function \texttt{solve} to find the roots of the characteristic polynomial of the matrix. Sometimes \texttt{solve} may not be able to find the roots of the polynomial; in that case some other functions in this package (except \texttt{innerproduct, unitvector, columnvector} and \texttt{gramschmidt}) will not work.

In some cases the eigenvalues found by \texttt{solve} may be complicated expressions. (This may happen when \texttt{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*)

**eivects** (*M*)

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.

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 of *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.

```lisp
(%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)
```

The *algsys* function 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.
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]);
(%o1) 

(%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]]
Function \texttt{ematrix}(m, n, x, i, j)\quad\text{Returns an } m \times n \text{ matrix, all elements of which are zero except for the } [i, j] \text{ element which is } x.

Function \texttt{entermatrix}(m, n)\quad\text{Returns an } m \times n \text{ 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. \texttt{entermatrix} evaluates its arguments.

\begin{verbatim}
(%i1) n: 3$
(%i2) m: entermatrix (n, n)$

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;

\begin{bmatrix}
3 \\
(b+a) & 0 & 0 \\

\end{bmatrix}
\end{verbatim}

\begin{verbatim}
(%o3)
\begin{bmatrix}
4 \\
0 & (b+a) & 0 \\
0 & 0 & (b+a)
\end{bmatrix}
\end{verbatim}

Function \texttt{genmatrix}(a, i1, j1, i2, j2)\quad\text{Returns a matrix generated from } a, \text{ taking element } a[i_1,j_1] \text{ as the upper-left element and } a[i_2,j_2] \text{ as the lower-right element of the matrix. Here } a \text{ is a declared array (created by \texttt{array} but not by \texttt{make_array}) or an undeclared array, or an array function, or a lambda expression of two arguments. (An array function is created like}

\begin{verbatim}
function \texttt{a(i, j)} { 
return (i + j); 
}
\end{verbatim}
other functions with := or define, but arguments are enclosed in square brackets instead of parentheses.)

If \( j \) is omitted, it is assumed equal to \( i \). If both \( j \) and \( i \) 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:

\[
\begin{align*}
(\%i1) & \ h \ [i, j] := 1 / (i + j - 1); \\
(\%o1) & \ h \ := \frac{1}{i + j - 1} \\
(\%i2) & \ \text{genmatrix} (h, 3, 3); \\
(\%o2) & \ \begin{bmatrix}
1 & 1 \\
1 & -1 \\
2 & 3 \\
\end{bmatrix} \\
(\%i3) & \ \text{array} (a, \text{fixnum}, 2, 2); \\
(\%o3) & \ a \\
(\%i4) & \ a \ [1, 1] : %e; \\
(\%o4) & \ %e \\
(\%i5) & \ a \ [2, 2] : %pi; \\
(\%o5) & \ %pi \\
(\%i6) & \ \text{genmatrix} (a, 2, 2); \\
(\%o6) & \ \begin{bmatrix}
%e & 0 \\
0 & %pi \\
\end{bmatrix} \\
(\%i7) & \ \text{genmatrix} (\lambda ([i, j], j - i), 3, 3); \\
(\%o7) & \ \begin{bmatrix}
0 & 1 & 2 \\
-1 & 0 & 1 \\
-2 & -1 & 0 \\
\end{bmatrix} \\
(\%i8) & \ \text{genmatrix} (B, 2, 2); \\
(\%o8) & \ \begin{bmatrix}
B & B \\
1, 1 & 1, 2 \\
\end{bmatrix}
\end{align*}
\]
gramschmidt \( (x) \)

gramschmidt \( (x, F) \)

Function

Carries out the Gram-Schmidt orthogonalization algorithm on \( x \), which is either a matrix or a list of lists. \( x \) is not modified by \texttt{gramschmidt}. The inner product employed by \texttt{gramschmidt} is \( F \), if present, otherwise the inner product is the function \texttt{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.

\texttt{factor} is called at each stage of the algorithm to simplify intermediate results. As a consequence, the return value may contain factored integers.

\texttt{load(eigen)} loads this function.

Example:

Gram-Schmidt algorithm using default inner product function.

\begin{verbatim}
(%i1) load (eigen)$
(%i2) x: matrix ([1, 2, 3], [9, 18, 30], [12, 48, 60]);
    [ 1   2   3 ]
    [         ]
    [ 9  18  30 ]
    [         ]
    [12 48 60 ]
(%o2) 
(%i3) y: gramschmidt (x);
    2   2   4   3
2 --- - ---, ---
7    7   5  5
(%o3) 
(%i4) map (innerproduct, [y[1], y[2], y[3]], [y[2], y[3], y[1]]);
(%o4) [0, 0, 0]
\end{verbatim}

Gram-Schmidt algorithm using a specified inner product function.

\begin{verbatim}
(%i1) load (eigen)$
(%i2) ip (f, g) := integrate (f * g, u, a, b);
    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
2 --- - ---
7    7
(%o3) 
(%i4) map (ip, [y[1], y[2], y[3]], [y[2], y[3], y[1]]),
a=-%pi/2, b=%pi/2;
    3 3 5 2 3 2 3
2 7 7 5 5
(%o4) [0, 0, 0]
\end{verbatim}

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 \texttt{conjugate} \( (x) \cdot y \), where \( \cdot \) is the noncommutative multiplication operator.

\texttt{load}\("\text{eigen}\") loads this function.

inprod is a synonym for \texttt{innerproduct}.

\textbf{invert} \( (M) \)

Function

Returns the inverse of the matrix \( M \). The inverse is computed by the adjoint method.

This allows a user to compute the inverse of a matrix with bfloat entries or polynomials with floating pt. coefficients without converting to \texttt{cre}-form.

Cofactors are computed by the \texttt{det} function, so if \texttt{ratmx} is false the inverse is computed without changing the representation of the elements.

The current implementation is inefficient for matrices of high order.

When \texttt{detout} is true, the determinant is factored out of the inverse.

The elements of the inverse are not automatically expanded. If \( M \) has polynomial elements, better appearing output can be generated by \texttt{expand\ (invert (m)), detout}.

If it is desirable to then divide through by the determinant this can be accomplished by \texttt{xthru (%) or alternatively from scratch by}

\[
\texttt{expand (adjoint (m)) / expand (determinant (m))}
\]

\[
\texttt{invert (m) := adjoint (m) / determinant (m)}
\]

See ^\^ (noncommutative exponent) for another method of inverting a matrix.

\textbf{list\_matrix\_entries} \( (M) \)

Function

Returns a list containing the elements of the matrix \( M \).

Example:

\[
(\%i1) \text{list\_matrix\_entries(matrix([a,b],[c,d]));}
\]

\[
(\%o1) \quad [a, b, c, d]
\]

\textbf{lmxchar} \hspace{1em} \text{Option variable}

Default value: [

\texttt{lmxchar} is the character displayed as the left delimiter of a matrix. See also \texttt{rmxchar}.

Example:

\[
(\%i1) \text{lmxchar: "|"}$
\]

\[
(\%i2) \text{matrix ([a, b, c], [d, e, f], [g, h, i]);}
\]

\[
\begin{array}{ccc}
| & a & b & c \\
| & d & e & f \\
| & g & h & i \\
\end{array}
\]

\[
(\%o2)
\]
**Function**

`matrix(row_1, ..., row_n)`

Returns a rectangular matrix which has the rows `row_1, ..., 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.

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.**
  ```lisp
  (%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.**
  ```lisp
  (%i3) x + y;
  (%o3) [ %pi + 17  %e + 3 ]
       [   a - 8   b + 11 ]
  ```

- **Subtraction, element by element.**
  ```lisp
  (%i4) x - y;
  (%o4) [ 17 - %pi  3 - %e ]
       [   - a - 8  11 - b ]
  ```

- **Multiplication, element by element.**
  ```lisp
  (%i5) x * y;
  (%o5) [ 17 %pi  3 %e ]
       [   - 8 a   11 b ]
  ```
• Division, element by element.

\[(%i6) \ x / \ y; \]
\[
\begin{bmatrix}
17 & -1 \\
--- & 3 \%e \\
\%pi & \\
\end{bmatrix}
\]

\[(%o6) \]
\[
\begin{bmatrix}
8 & 11 \\
- & -- \\
a & b \\
\end{bmatrix}
\]

• Matrix to a scalar exponent, element by element.

\[(%i7) \ x ^ 3; \]
\[
\begin{bmatrix}
4913 & 27 \\
\cdot & \\
-512 & 1331 \\
\end{bmatrix}
\]

\[(%o7) \]

• Scalar base to a matrix exponent, element by element.

\[(%i8) \ \text{exp}(y); \]
\[
\begin{bmatrix}
\%pi & \%e \\
\%e & \%e \\
\end{bmatrix}
\]

\[(%o8) \]
\[
\begin{bmatrix}
a & b \\
\%e & \%e \\
\end{bmatrix}
\]

• Matrix base to a matrix exponent. This is not carried out element by element.

\[(%i9) \ x ^ y; \]
\[
\begin{bmatrix}
\%pi & \%e \\
\end{bmatrix}
\]

\[(%o9) \]
\[
\begin{bmatrix}
17 & 3 \\
- & 8 \\
11 & 11 \\
\end{bmatrix}
\]

• Noncommutative matrix multiplication.

\[(%i10) \ x . \ y; \]
\[
\begin{bmatrix}
3 \ a + 17 \%pi & 3 \ b + 17 \%e \\
11 \ a - 8 \%pi & 11 \ b - 8 \%e \\
\end{bmatrix}
\]

\[(%o10) \]
\[
\begin{bmatrix}
17 \%pi - 8 \%e & 3 \%pi + 11 \%e \\
17 \ a - 8 \ b & 11 \ b + 3 \ a \\
\end{bmatrix}
\]

• Noncommutative matrix exponentiation. A scalar base \(b\) to a matrix power \(M\) is carried out element by element and so \(b^\%\) is the same as \(b^\%m\).

\[(%i12) \ x ^ ^ 3; \]
\[
\begin{bmatrix}
3833 & 1719 \\
\cdot & \\
-4584 & 395 \\
\end{bmatrix}
\]

\[(%o12) \]
\[
\begin{bmatrix}
\%e ^ ^ y; \]
\]
A matrix raised to a -1 exponent with noncommutative exponentiation is the matrix inverse, if it exists.

\[
\begin{bmatrix}
%\pi & %e \\
%e & %e \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
\%e & %e \\
\end{bmatrix}
\]

- Function **matrixmap** \((f, M)\)
  - Returns a matrix with element \(i,j\) equal to \(f(M[i,j])\).
  - See also map, fullmap, fullmapl, and apply.

- Function **matrixp** \((expr)\)
  - Returns true if expr is a matrix, otherwise false.

- Option variable **matrix_element_add**
  - 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:

\[
\begin{bmatrix}
11 & 3 \\
211 & 211 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
8 & 17 \\
211 & 211 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 \\
0 & 1 \\
\end{bmatrix}
\]
(%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

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];  
    2  2  2  
    sqrt((c - z) + (b - y) + (a - x))
(%i4) aa: matrix ([a, b, c], [d, e, f]);  
(%o4)  [ a  b  c ]  
    [ d  e  f ]  
(%i5) bb: matrix ([u, v, w], [x, y, z]);  
(%o5)  [ u  v  w ]  
    [ 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

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:

```lisp
(%i1) declare (a, nonscalar)$
(%i2) transpose ([a, b]);
      [ transpose(a) ]
     [             ]
      [     b     ]
(%o2) [        ]                   
     [  ]
(%i3) matrix_element_transpose: nonscalars$
(%i4) transpose ([a, b]);
      [ transpose(a) ]
     [             ]
      [     b     ]
(%o4) [        ]                   
     [  ]
(%i5) matrix_element_transpose: transpose$
(%i6) transpose ([a, b]);
      [ transpose(a) ]
     [             ]
      [     transpose(b)     ]
(%o6) [        ]                     
     [  ]
(%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 ]
    [                  ]
    [ 7 %i 11 ]
(%o8) [                  ]            
    [  ]
(%i9) transpose (m);
    [ 1 - 5 %i - 7 %i ]
    [                  ]
    [ 2 %i + 3 11 ]
```

Function

`mattrace (M)`

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.

Function

`minor (M, i, j)`

Returns the `i, j` minor of the matrix `M`. That is, `M` with row `i` and column `j` removed.
Function \texttt{ncexpt} \((a, b)\)

If a non-commutative exponential expression is too wide to be displayed as \(a^\wedge b\) it appears as \texttt{ncexpt} \((a, b)\).

\texttt{ncexpt} is not the name of a function or operator; the name only appears in output, and is not recognized in input.

Function \texttt{ncharpoly} \((M, x)\)

Returns the characteristic polynomial of the matrix \(M\) with respect to \(x\). This is an alternative to Maxima’s \texttt{charpoly}.

\texttt{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. \texttt{charpoly} works by forming the determinant of \(x \cdot \text{ident}[n] - a\). Thus \texttt{ncharpoly} wins, for example, in the case of large dense matrices filled with integers, since it avoids polynomial arithmetic altogether.

\texttt{load ("nchrpl")} loads this file.

Function \texttt{newdet} \((M)\)

Computes the determinant of the matrix \(M\) by the Johnson-Gentleman tree minor algorithm. \texttt{newdet} returns the result in CRE form.

Declaration \texttt{nonscalar}

Makes atoms behave as does a list or matrix with respect to the dot operator.

Function \texttt{nonscalarp} \((expr)\)

Returns \texttt{true} if \(expr\) is a non-scalar, i.e., it contains atoms declared as non-scalars, lists, or matrices.

Function \texttt{permanent} \((M)\)

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. \texttt{permanent} returns the result in CRE form.

See also \texttt{newdet}.

Function \texttt{rank} \((M)\)

Computes the rank of the matrix \(M\). That is, the order of the largest non-singular subdeterminant of \(M\).

\texttt{rank} may return the wrong answer if it cannot determine that a matrix element that is equivalent to zero is indeed so.

Option variable \texttt{ratmx}

Default value: \texttt{false}

When \texttt{ratmx} is \texttt{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 \(\text{row}(M, i)\) Returns the \(i\)'th row of the matrix \(M\). The return value is a matrix.

**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** \((\text{coordinatetransform})\)  
Function \(\text{scalefactors}(\text{coordinatetransform})\)  
Here `coordinatetransform` evaluates to the form `[[expression1, expression2, ...], indeterminate1, indeterminate2, ...]`, where `indeterminate1`, `indeterminate2`, etc. are the curvilinear coordinate variables and where a set of rectangular Cartesian components is given in terms of the curvilinear coordinates by `[[expression1, expression2, ...]]`.  
`coordinates` is set to the vector `[[indeterminate1, indeterminate2, ...]]`, and `dimension` is set to the length of this vector. `SF[1]`, `SF[2]`, ..., `SF[dimension]` are set to the coordinate scale factors, and `sfprod` is set to the product of these scale factors. Initially, `coordinates` is `[X, Y, Z]`, `dimension` is 3, and `SF[1]=SF[2]=SF[3]=SFPROD=1`, corresponding to 3-dimensional rectangular Cartesian coordinates. To expand an expression into physical components in the current coordinate system, there is a function with usage of the form

**setelmx** \((x, i, j, M)\)  
Function \(\text{setelmx}(x, i, j, M)\) 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 \(\text{similaritytransform}(M)\) \(\text{similaritytransform}(M)\) 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 (i_1, ..., i_m, M, j_1, ..., j_n)

Function

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 ]
[ ]
[ -1 8 5 2 ]
[ ]
[ 9 2 11 4 ]

(%o1)

(%i2) triangularize (M);

[ -1 8 5 2 ]
[ ]
[ 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 `knowneigvects` 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 `knowneigvects` 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`.

```lisp
unitvector (x)
```

Function

```lisp
uvect (x)
```

Function

Returns \( x / \text{norm}(x) \): this is a unit vector in the same direction as \( x \).

`load` ("eigen") loads this function.

`uvect` is a synonym for `unitvector`.

```lisp
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.

```lisp
expandcrosscross
```

Simplifies \( p \cdot (q \times r) \) to \((p \cdot r) \times q - (p \cdot q) \times r\).

```lisp
expandcurlcurl
```

Simplifies \( \nabla \times \nabla \times p \) to \( \nabla \cdot \nabla \cdot p + \nabla ^2 p \).

```lisp
expandlaplaciantodivgrad
```

Simplifies \( \Delta p \) to \( \nabla \cdot \nabla p \).

```lisp
expandcross
```

Enables `expandcrossplus` and `expandcrosscross`.

```lisp
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.

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

[ ] also enclose the subscripts of a list, array, hash array, or array function.

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);
1
(%o8) g := -------
    k 2  
    k + 1
(%i9) g[10];
1
(%o9) ---
    101
26 Affine

26.1 Introduction to Affine

affine is a package to work with groups of polynomials.

26.2 Functions and Variables for Affine

**fast_linsolve** ([expr₁, ..., exprₘ], [x₁, ..., xₙ])

Solves the simultaneous linear equations expr₁, ..., exprₘ for the variables x₁, ..., xₙ. Each exprᵢ may be an equation or a general expression; if given as a general expression, it is treated as an equation of the form exprᵢ = 0.

The return value is a list of equations of the form [x₁ = a₁, ..., xₙ = aₙ] where a₁, ..., aₙ are all free of x₁, ..., xₙ.

fast_linsolve is faster than linsolve for system of equations which are sparse.

load(affine) loads this function.

**grobner** basis ([expr₁, ..., exprₘ])

Returns a Groebner basis for the equations expr₁, ..., exprₘ. The function polysimp can then be used to simplify other functions relative to the equations.

`grobner_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 expr₁, ..., exprₘ, that is, if and only if f is a polynomial combination of the elements of expr₁, ..., exprₘ.

load(affine) loads this 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.
**Function**

**declare_weights** \((x_1, w_1, \ldots, x_n, w_n)\)

Assigns weights \(w_1, \ldots, w_n\) to \(x_1, \ldots, x_n\), respectively. These are the weights used in computing **nc_degree**.

`load(affine)` loads this function.

**nc_degree** \((p)\)

Returns the degree of a noncommutative polynomial \(p\). See **declare_weights**.

`load(affine)` loads this function.

**dotsimp** \((f)\)

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** \(([x_1, \ldots, x_n], n)\)

If \(\text{set_up_dot_simplifications}\) has been previously done, finds the central polynomials in the variables \(x_1, \ldots, x_n\) 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** \((n, add_to_simps)\)

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 \(\text{reset, false}\) says don’t bother to query about resetting things.

`load(affine)` loads this function.

**mono** \(([x_1, \ldots, x_n], n)\)

Returns the list of independent monomials relative to the current dot simplifications of degree \(n\) in the variables \(x_1, \ldots, x_n\).

`load(affine)` loads this function.

**monomial_dimensions** \((n)\)

Compute the Hilbert series through degree \(n\) for the current algebra.

`load(affine)` loads this function.

**extract_linear_equations** \(([p_1, \ldots, p_n], [m_1, \ldots, m_n])\)

Makes a list of the coefficients of the noncommutative polynomials \(p_1, \ldots, p_n\) of the noncommutative monomials \(m_1, \ldots, m_n\). The coefficients should be scalars. Use **list_nc_monomials** to build the list of monomials.

`load(affine)` loads this function.
list\_nc\_monomials \((p_1, ... , p_n)\)

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

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.
Chapter 27: itensor

27 itensor

27.1 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, \texttt{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.

### 27.1.1 New tensor notation

Earlier versions of the \texttt{itensor} package in Maxima used a notation that sometimes led to incorrect index ordering. Consider the following, for instance:

\begin{verbatim}
(%i2) imetric(g); 
(%o2) done 
(%i3) ishow(g([],[j,k])*g([],[i,l])*a([i,j],[[]])); 
   i  l  j  k  
   g  g  a  
   i  j
(%t3) g g a  
   i  j
(%i4) ishow(contract(%));  
   k  l  
   a
(%t4)
\end{verbatim}

This result is incorrect unless \texttt{a} happens to be a symmetric tensor. The reason why this happens is that although \texttt{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 \texttt{contract} and \texttt{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:

\begin{verbatim}
(%i5) ishow(g([-j,-k],[[]])*g([-i,-l],[[]])*a([i,j],[[]])); 
   i  l  j  k  
   g  a  g  
   i  j
(%t5) g a g  
   i  j
(%i6) ishow(contract(%));  
   i  j  
   a
(%t6)
\end{verbatim}

Presently, the only code that makes use of this notation is the \texttt{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!

### 27.1.2 Indicial tensor manipulation

The indicial tensor manipulation package may be loaded by \texttt{load(itensor)}. Demos are also available: try \texttt{demo(tensor)}.
In *itensor* a tensor is represented as an "indexed object". This is a function of 3 groups of indices which represent the covariant, contravariant and derivative indices. The covariant indices are specified by a list as the first argument to the indexed object, and the contravariant indices by a list as the second argument. If the indexed object lacks either of these groups of indices then the empty list [] is given as the corresponding argument. Thus, \( g([a,b],[c]) \) represents an indexed object called \( g \) which has two covariant indices \((a,b)\), one contravariant index \((c)\) and no derivative indices.

The derivative indices, if they are present, are appended as additional arguments to the symbolic function representing the tensor. They can be explicitly specified by the user or be created in the process of differentiation with respect to some coordinate variable. Since ordinary differentiation is commutative, the derivative indices are sorted alphanumerically, unless *iframe_flag* is set to true, indicating that a frame metric is being used. This canonical ordering makes it possible for Maxima to recognize that, for example, \( t([a],[b],i,j) \) is the same as \( t([a],[b],j,i) \). Differentiation of an indexed object with respect to some coordinate whose index does not appear as an argument to the indexed object would normally yield zero. This is because Maxima would not know that the tensor represented by the indexed object might depend implicitly on the corresponding coordinate. By modifying the existing Maxima function *diff* in *itensor*, Maxima now assumes that all indexed objects depend on any variable of differentiation unless otherwise stated. This makes it possible for the summation convention to be extended to derivative indices. It should be noted that *itensor* does not possess the capabilities of raising derivative indices, and so they are always treated as covariant.

The following functions are available in the tensor package for manipulating indexed objects. At present, with respect to the simplification routines, it is assumed that indexed objects do not by default possess symmetry properties. This can be overridden by setting the variable *allsym*[false] to true, which will result in treating all indexed objects completely symmetric in their lists of covariant indices and symmetric in their lists of contravariant indices.

The *itensor* package generally treats tensors as opaque objects. Tensorial equations are manipulated based on algebraic rules, specifically symmetry and contraction rules. In addition, the *itensor* package understands covariant differentiation, curvature, and torsion. Calculations can be performed relative to a metric of moving frame, depending on the setting of the *iframe_flag* variable.

A sample session below demonstrates how to load the *itensor* package, specify the name of the metric, and perform some simple calculations.

```
(%i1) load(itensor);
(%o1) /share/tensor/itensor.lisp
(%i2) imetric(g);
(%o2) done
(%i3) components(g([i,j],[ ]),p([i,j],[ ])*e([],[ ]))$
(%i4) ishow(g([k,l],[ ]))$
(%t4) e p
(%i5) ishow(diff(v([i],[ ]),t))$
(%t5) 0
(%i6) depends(v,t);
```
(%o6) \[ v(t) \]
(%i7) ishow(diff(v([i],[]),t))$
\begin{align*}
&\frac{d}{dt} v_i \\
\end{align*}

(%t7) -- (v )
dt i

(%i8) ishow(idiff(v([i],[]),j))$
\begin{align*}
&v_{i,j} \\
\end{align*}

(%t8) v_{i,j}

(%i9) ishow(extdiff(v([i],[]),j))$
\begin{align*}
&v_{j,i} - v_{i,j} \\
&\frac{1}{2} \\
\end{align*}

(%t9) \frac{1}{2} v_{j,i} - v_{i,j}

(%i10) ishow(liediff(v,w([i],[i])))$
\begin{align*}
&v_{i,j} + v_{j,i} \\
&\frac{1}{2} \\
\end{align*}

(%t10) \frac{1}{2} v_{i,j} + v_{j,i}

(%i11) ishow(covdiff(v([i],[]),j))$
\begin{align*}
&v_{i,j} - v_{j,i} \\
&\frac{1}{2} \\
\end{align*}

(%t11) \frac{1}{2} v_{i,j} - v_{j,i}

(%i12) ishow(ev(%,ichr2))$
\begin{align*}
&v_{i,j} - (g^{ij} v_{i,p} + e^i p - e^p - e^p_{j,i} + e^p_{i,j} + e^p_{i,j})/2 \\
&\frac{1}{2} \\
\end{align*}

(%t12) \frac{1}{2} v_{i,j} - (g^{ij} v_{i,p} + e^i p - e^p - e^p_{j,i} + e^p_{i,j} + e^p_{i,j})/2

(%i13) iframe_flag:true;
true

(%i14) ishow(covdiff(v([i],[]),j))$
\begin{align*}
&v_{i,j} - v_{j,i} \\
&\frac{1}{2} \\
\end{align*}

(%t13) \frac{1}{2} v_{i,j} - v_{j,i}

(%i15) ishow(ev(%,icc2))$
\begin{align*}
&v_{i,j} - v_{j,i} \\
&\frac{1}{2} \\
\end{align*}

(%t14) \frac{1}{2} v_{i,j} - v_{j,i}

(%i16) ishow(radcan(ev(%,ifc2,ifc1)))$
\begin{align*}
&- (ifg v_{i,j} + ifg v_{j,i} - 2 v_{j,i} - v_{i,j})/2 \\
&\frac{1}{2} \\
\end{align*}

(%t15) \frac{1}{2} - (ifg v_{i,j} + ifg v_{j,i} - 2 v_{j,i} - v_{i,j})/2

(%i17) ishow(canform(s([i,j],[i]) - s([j,i])))$
\begin{align*}
&s - s \\
&\frac{1}{2} \\
\end{align*}

(%t16) \frac{1}{2} s - s

(%i18) ishow(canform(s([i,j],[i]) - s([j,i])))$
\begin{align*}
&s - s \\
&\frac{1}{2} \\
\end{align*}

(%t17) \frac{1}{2} s - s
27.2 Functions and Variables for itensor

27.2.1 Managing indexed objects

entertensor \( (\text{name}) \) \hspace{1cm} \text{Function}

is a function which, by prompting, allows one to create an indexed object called \text{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 \text{covdiff}).

changenamel \( (\text{old}, \text{new}, \text{expr}) \) \hspace{1cm} \text{Function}

will change the name of all indexed objects called \text{old} to \text{new} in \text{expr}. \text{old} may be either a symbol or a list of the form \([\text{name}, m, n]\) in which case only those indexed objects called \text{name} with \( m \) covariant and \( n \) contravariant indices will be renamed to \text{new}.

listoftens \hspace{1cm} \text{Function}

Lists all tensors in a tensorial expression, complete with their indices. E.g.,

\[
(\%16) \text{ishow(a([i,j],[k])*b([u],[v])*c([x,y][i])*d([i],[e]))}\$ k \\
(\%t6) \hspace*{1cm} \text{d e c + a b x y i j u,v} \\
(\%17) \text{ishow(listoftens(\%))}\$ k \\
(\%t7) \hspace*{1cm} \text{[a , b , c , d]} i j u,v x y
\]

ishow \( (\text{expr}) \) \hspace{1cm} \text{Function}

displays \text{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)

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))$
   k l j m p
   a b i j, m n k o, q r
(%t2)
(%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 (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 alphabetically (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(%)$
flipflag

Option variable

Default: 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 (tensor_1)

defcon (tensor_1, tensor_2, tensor_3)

Function

Function
gives tensor_1 the property that the contraction of a product of tensor_1 and tensor_2 results in tensor_3 with the appropriate indices. If only one argument, tensor_1, is given, 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**
  
  ```lisp
  remcon (tensor_1, ..., tensor_n)
  remcon (all)
  ```

  Function

  Removes all the contraction properties from the `tensor_1, ..., tensor_n`. `remcon(all)` removes all contraction properties from all indexed objects.

- **contract**

  ```lisp
  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**

  ```lisp
  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**

  ```lisp
  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([...],[...])` 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:

  ```lisp
  (%i2) components(g([],[i,j]),e([],[i])*p([],[j]))$
  (%i3) ishow(g([],[i,j]))$
  (%t3) e\[i\] j\[p\]
  ```

  2) As a matrix:

  ```lisp
  (%i5) lg:-ident(4)$lg[1,1]:1$lg;
  ```
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 \( \delta_{ij} \) to \( h \) if \( h \) has the same number of covariant and contravariant indices and no derivative indices, and \( g \) otherwise:

\[
\begin{align*}
(\%i4) \quad & h([1,2],[13]) := \text{if length}(11) = \text{length}(12) \text{ and length}(13) = 0 \text{ then } \delta_{ij} \text{ else } \text{apply}(g, \text{append}([11,12], 13)) \end{align*}
\]

\[
(\%i5) \quad \text{ishow}(h([i],[j]))$
\]

\[
(\%t5) \quad \delta_{ij}
\]

\[
(\%i6) \quad \text{ishow}(h([i],[j],[k]))$
\]

\[
(\%t6) \quad g_{ij,k}
\]

4) Using Maxima’s pattern matching capabilities, specifically the \texttt{defrule} and \texttt{applyb1} commands:

\[
(\%i11) \quad \text{load(itensor)};
\]

\[
(\%i12) \quad \text{matchdeclare}(11, \text{listp});
\]

\[
(\%i13) \quad \text{defrule}(r1, m([1], []), (i1: \text{idummy}(), g([11[1],11[2]], [])*q([i1], [])*e([], [i1])))$
\]

\[
(\%i14) \quad \text{defrule}(r2, m([], 11), (i1: \text{idummy}(), w([], [11[1], 11[2]])*e([i1], [])*q([], [i1])))$
\]

\[
(\%i15) \quad \text{ishow}(m([i], [])*m([], [i,m]))$
\]

\[
(\%t5) \quad m_i m_i n
\]
remcomps (tensor)  
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:

```
(%i11) load(ctensor);
(%o11) /share/tensor/ctensor.mac
(%i12) load(itensor);
(%o12) /share/tensor/itensor.lisp
(%i13) 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)]);

(%o13) [                  [  r   ]
     [ sqrt(-------) 0 0 0 ]
     [ r - 2 m          ]
     [ ]
     [  0   r 0 0 ]
     [ ]
     [  0 0 r sin(theta) 0 ]
     [ ]
     [ r - 2 m          ]
     [  0 0 0 sqrt(-------) ]
     [ r ]

(%i14) components(g([i,j],[[]]),lg);
(%o14) done
(%i15) showcomps(g([i,j],[[]]));

(%t5) g = [                  [  r   ]
     [ sqrt(-------) 0 0 0 ]
     [ r - 2 m          ]
     [ ]
     [  0   r 0 0 ]
     [ ]
     [  0 0 r sin(theta) 0 ]
     [ ]
     [ r - 2 m          ]
     [  0 0 0 sqrt(-------) ]
     [ r ]

(%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([1,j],[])` is a valid tensorial object.

**kdels** \((L1, L2)\)

Function

Symmetrized Kronecker delta, used in some calculations. For instance:

```lisp
(%i1) load(itensor);  
(%o1) /share/tensor/itensor.lisp 
(%i2) kdelta([1,2],[2,1]);  
(%o2) - 1  
(%i3) kdels([1,2],[2,1]);  
(%o3) 1  
(%i4) ishow(kdelta([a,b],[c,d]));  
  c   d      d   c     
  kdelta kdelta - kdelta kdelta  
  a   b      a   b  
(%i4) ishow(kdels([a,b],[c,d]));  
  c   d      d   c     
  kdelta kdelta + kdelta kdelta  
  a   b      a   b
```
Function **levi_civita** \((L)\)

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.

Function **lc2kdt** \((expr)\)

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**.

```maxima
(%i1) load(itensor);
(%o1) /share/tensor/itensor.lisp
(%i2) expr:ishow('levi_civita([],[i,j])*'levi_civita([k,l],[])'*a([j],[k]))$
   i j k
(%t2) levi_civita a levi_civita
   j k l
(%i3) ishow(ev(expr,levi_civita))$
   i j k 1 2
(%t3) kdelta a kdelta
   1 2 j k l
(%i4) ishow(ev(%kdelta))$
   i j j i k
(%t4) (kdelta kdelta - kdelta kdelta ) a
   1 2 1 2 j
   1 2 2 1
   (kdelta kdelta - kdelta kdelta )
   k l k l
(%i5) ishow(lc2kdt(expr))$
   k i j k j i
(%t5) a kdelta kdelta - a kdelta kdelta
   j k l j k l
(%i6) ishow(contract(expand(%)))$
   i 1
(%t6) a - a kdelta
   l 1
(%i7) expr:ishow('levi_civita([],[i,j])*'levi_civita([],[k,l])*a([j],[k],[]))$
```

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.
i j k l
(%t7) levi_civita levi_civita a

(%i8) ishow(lc2kdt(expr))$
Maxima encountered a Lisp error:

Error in $IMETRIC [or a callee]:
$IMETRIC [or a callee] requires less than two arguments.

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
(%i9) imetric(g);
(%o9) done
(%i10) ishow(lc2kdt(expr))$
    %3 i k %4 j l %3 i l %4 j
(%t10) (g kdelta g kdelta - g kdelta g
    %3 %4 %3 %4 k
    kdelta ) a
    %4 j k
(%i11) ishow(contract(expand(%)))$
    l i l i j
(%t11)
    a - g a

lc_l 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) el1:ishow('levi_civita([i,j,k],[[]])*a([],[])*a([],[]))$
    i j
(%t2) a a levi_civita
(%i3) el2:ishow('levi_civita([],[])*a([[]])*a([[]]))$
    i j k
(%t3) levi_civita a a
(%i4) canform(contract(expand(applyb1(el1,lc_l,lc_u))));
(%t4) 0
(%i5) canform(contract(expand(applyb1(el2,lc_l,lc_u))));
(%t5) 0
lc_u

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_l.

canten (expr)

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.

cconcen (expr)

Similar to canten but also performs index contraction.

27.2.2 Tensor symmetries

allsym

Option variable

Default: 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, ...])

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);
(%i2) /share/tensor/itensor.lisp
(%i3) expr:contract(expand(a([i1, j1, k1], []))*kdels([i, j, k], [i1, j1, k1]))$
(%i4) ishow(expr)$
remsym (tensor, m, n)
Removes all symmetry properties from tensor which has m covariant indices and n contravariant indices.

canform (expr)
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.

27.2.3 Indicial tensor calculus
diff (expr, v_1, [n_1, [v_2, n_2] ...])
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 $\text{dim}$. This will cause the differentiation to be carried out with respect to the $v_i$th member of the list $\text{vect\_coords}$. If $\text{vect\_coords}$ 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 $\text{diff}$ 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.

$idiff\ (expr, v_1, [n_1, [v_2, n_2] ...])$

Indicial differentiation. Unlike $\text{diff}$, 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.

$\text{liediff}\ (v, ten)$

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],[[]])*b([[],[k],1])))$
   k     k     k
   %2    %2    %2
   b ( v a + v a + v a )
   ,1    ,i %2 ,j %2 ,i %2 j
   + (v b - b v + v b ) a
   ,%1 ,1 %1 ,1 %1 i j
```

$\text{rediff}\ (ten)$

Evaluates all occurrences of the $idiff$ command in the tensorial expression $ten$.

$\text{undiff}\ (expr)$

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 $\text{ev}(expr, idiff)$.
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))$
\%(t3) icurvature i j k,m
\%(i4) ishow(evundiff(%))$
\%(t4) - ichr2 - ichr2 ichr2 - ichr2 ichr2
i k,j m i k,m i k
\l i i i i
\l l l l
\l + ichr2 + ichr2 ichr2 + ichr2 ichr2
i j,k m i j,k m i j,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))$
\%(t6) -----------------------------------------
\l k l k l
\l g (g - g + g )
j %3,i l i j,%3 l i %3,j l
\l + -----------------------------------
\l k l k l
\l g (g - g + g )
j %3,i l i j,%3 l i %3,j l
\l -----------------------------------------
\l k l k l

flush (expr, tensor_1, tensor_2, ...)

Function

Set to zero, in expr, all occurrences of the tensor_i that have no derivative indices.
flushd \((expr, tensor_1, tensor_2, \ldots)\) 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.

\[
\begin{align*}
  \%t1 & \text{ load(itensor);} \\
  \%t2 & \text{ ishow(a([i],[J,r],k,r)*a([i],[j,r,s],k,r,s));} \\
  \%t3 & \text{ ishow(flushnd(%a,3));}
\end{align*}
\]

coord \((tensor_1, tensor_2, \ldots)\) 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 \(idiff(x([],[]),j)\) gives \(\text{kdelta}([],[]),j)\). coord is a list of all indexed objects having this property.

remcoord \((tensor_1, tensor_2, \ldots)\) Function
remcoord \((all)\) Function
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 \(g([],a,j)*p([m],[n],i,j)\).

connmetderiv \((expr, tensor)\) Function
Simplifies expressions containing ordinary derivatives of both covariant and contravariant forms of the metric tensor (the current restriction). For example, connmetderiv can relate the derivative of the contravariant metric tensor with the Christoffel symbols as seen from the following:

\[
\begin{align*}
  \%t1 & \text{ load(itensor);} \\
  \%t2 & \text{ ishow(g([],[],[a,b],c));}
\end{align*}
\]
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 `simpmetderiv` 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))$
     a b b c
  (%t3) g g g g
       a b,d b c,e
  ishow(canform(%))$
errexp1 has improper indices
```
-- an error. Quitting. To debug this try debugmode(true);
(%i5) ishow(simpmetderiv(%))$
   \begin{array}{ccc}
   a & b & c \\
   g & g & g \\
   a, b, d & b & c, e \\
   \end{array}$
(%i6) flipflag:not flipflag;
(%o6) true
(%i7) ishow(simpmetderiv(%th(2)))$
   \begin{array}{ccc}
   a & b & c \\
   g & g & g \\
   a, b & d & b, c \\
   \end{array}$
(%i8) flipflag:not flipflag;
(%o8) false
(%i9) ishow(simpmetderiv(%th(2),stop))$
   \begin{array}{ccc}
   a & b & c \\
   g & g & g \\
   , d & , e & a, b & b, c \\
   \end{array}$
(%i10) ishow(contract(%))$
   \begin{array}{cc}
   b & c \\
   - g & g \\
   , e & c, b, d \\
   \end{array}$

See also weyl.dem for an example that uses simpmetderiv 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.

27.2.4 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
\[
(\frac{g_{ik, j} + g_{j, ik} - g_{i, jk}}{2})/2 .
\]
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])
Yields the Christoffel symbol of the second kind defined by the relation
\[
ichr2([i,j],[k]) = g \frac{(g_{is,j} + g_{js,i} - g_{ij,s})}{2}
\]

icurvature ([i, j, k], [h])
Yields the Riemann curvature tensor in terms of the Christoffel symbols of the second kind (ichr2). The following notation is used:
\[
\text{icurvature} = - ichr2 - ichr2 ichr2 + ichr2
\]

\[
\text{h} \text{h} \text{h} \text{h} \text{h}
\]

\[
\text{icurvature} = - ichr2 - ichr2 ichr2 + ichr2
\]

\[
\text{i} \text{j} \text{k} \text{i} \text{k,j} \text{i} \text{k} \text{i} \text{j,k}
\]

\[
\text{h} \text{h} \text{h} \text{h}
\]

\[
\text{icurvature} = - ichr2 - ichr2 ichr2 + ichr2
\]

\[
\text{i} \text{j} \text{k} \text{i} \text{j,s} \text{i} \text{j,s} \text{i} \text{j,s}
\]

covdiff (expr, v_{1}, v_{2}, ...)
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) enter_tensor();$
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) a

(%i3) ishow(covdiff(%,s))$

(%t3) - a ichr2 - a ichr2 + a

(%i4) imetric:g;
(%o4) g
(%i5) ishow(ev(%th(2),ichr2))$

(%t5) - ------------------------------------------

2
\[
\begin{align*}
%1 %3 k \\
g a (g - g + g ) \\
%1 j s %3 i i s, %3 i %3, s \\
- \frac{2}{2}
\end{align*}
\]

\[
\begin{align*}
%2 %1 \\
g a (g - g + g ) \\
i j s %2, %1 %1 s, %2 %1 %2, s \quad k \\
+ \frac{2}{2} \quad i j s
\end{align*}
\]

(\%i6)

**lorentz_gauge** (\textit{expr})

Imposes the Lorentz condition by substituting 0 for all indexed objects in \textit{expr} that have a derivative index identical to a contravariant index.

**igeodesic_coords** (\textit{expr, name})

Causes undifferentiated Christoffel symbols and first derivatives of the metric tensor vanish in \textit{expr}. The name in the \textit{igeodesic_coords} function refers to the metric \textit{name} (if it appears in \textit{expr}) while the connection coefficients must be called with the names \textit{ichr1} and/or \textit{ichr2}. The following example demonstrates the verification of the cyclic identity satisfied by the Riemann curvature tensor using the \textit{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 \\
+ ichr2 ichr2 \\
t r s t %1 t \\
(\%i3) ishow(igeodesic_coords(%, ichr2))$
(\%t3) ichr2 - ichr2 \\
r s , t r t, s \\
(\%i4) ishow(igeodesic_coords(icurvature([r,s,t],[u]), ichr2)+ igeodesic_coords(icurvature([s,t,r],[u]), ichr2)+ igeodesic_coords(icurvature([t,r,s],[u]), ichr2))$
(\%t4) - ichr2 + ichr2 + ichr2 - ichr2 \\
t s , r t r, s s t, r s r, t \\
+ ichr2 + ichr2 \\
r t, s r s, t
27.2.5 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 \texttt{iframe\_flag} to \texttt{true}. This causes the Christoffel-symbols, \texttt{ichr1} and \texttt{ichr2}, to be replaced by the more general frame connection coefficients \texttt{icc1} and \texttt{icc2} in calculations. Specifically, the behavior of \texttt{covdiff} and \texttt{icurvature} is changed.

The frame is defined by two tensors: the inverse frame field (\texttt{ifri}, the dual basis tetrad), and the frame metric \texttt{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 \texttt{iframe\_flag} is true, many \texttt{itensor} expressions use the frame metric \texttt{ifg} instead of the metric defined by \texttt{imetric} for raising and lowerind indices.

IMPORTANT: Setting the variable \texttt{iframe\_flag} to \texttt{true} does NOT undefine the contraction properties of a metric defined by a call to \texttt{defcon} or \texttt{imetric}. If a frame field is used, it is best to define the metric by assigning its name to the variable \texttt{imetric} and NOT invoke the \texttt{imetric} function.

Maxima uses these two tensors to define the frame coefficients \texttt{(ifc1} and \texttt{ifc2}) which form part of the connection coefficients \texttt{(icc1} and \texttt{icc2}), as the following example demonstrates:

```
(%i1) load(itensor);  
(%o1) /share/tensor/itensor.lisp  
(%i2) iframe_flag:true;  
(%o2) true  
(%i3) ishow(covdiff(v\([\],\[i\]),\[j\]));  
   (\(\%1\) \(i\)) \(i\) \(\%1\)  
   \(\%t3\) \(v\) \(icc2\) \(v\)  
   \(j\) \(\%1\) \(j\)  
(%i4) ishow(ev(\(\%1\),icc2));  
   \(\%1\) \(i\) \(i\)  
   \(\%t4\) \(v\) \(ifc2\) \(+\) \(v\)  
   \(\%1\) \(j\) \(j\)  
(%i5) ishow(ev(\(\%1\),ifc2));  
   \(\%1\) \(i\) \(\%2\) \(i\)  
   \(\%t5\) \(v\) \(ifg\) \(ifc1\) \(+\) \(v\)  
   \(\%1\) \(j\) \(\%2\) \(j\)  
(%i6) ishow(ev(\(\%1\),ifc1));
```
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])))$
```

The frame bracket. The contribution of the frame metric to the connection coefficients
is expressed using the frame bracket:

\[
-ifb + ifb + ifb
------------------------- + v
2 \\
\]

The frame bracket itself is defined in terms of the frame field and frame metric.
Two alternate methods of computation are used depending on the value of `frame_bracket_form`. If true (the default) or if the `itorsion_flag` is true:

\[
ifb = ifr ifr (ifr ifr - ifr ifr - ifr itr)
\]

Otherwise:

\[
ifb = (ifr ifr - ifr ifr ) ifri
\]


**icc1**

Connection coefficients of the first kind. In *itensor*, defined as

\[
icc_1 = ichr_1 - ikt_1 - inmc_1
\]

\[
\begin{array}{cccc}
abc & abc & abc & abc \\
\end{array}
\]

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**

Connection coefficients of the second kind. In *itensor*, defined as

\[
icc_2 = ichr_2 - ikt_2 - inmc_2
\]

\[
\begin{array}{cccc}
ab & ab & ab & ab \\
\end{array}
\]

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**

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:

\[
ifc_1 = \frac{-ifb + ifb + ifb}{abc}
\]

\[
\begin{array}{cccc}
c & a & b & b & c & a & a & b & c \\
\end{array}
\]

\[
\begin{array}{cccc}
abc & 2 \\
\end{array}
\]

**ifc2**

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:

\[
ifc_2 = ifg ifc_1
\]

\[
\begin{array}{cccc}
c & cd \\
\end{array}
\]

\[
\begin{array}{cccc}
ab & abd \\
\end{array}
\]
**ifr**

The frame field. Contracts with the inverse frame field (**ifri**) to form the frame metric (**ifg**).

**ifri**

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**

The frame metric. Defaults to **kdelta**, but can be changed using **components**.

**ifgi**

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.

### 27.2.6 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**

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**:

\[
g_{ij;k} = \frac{g_{ij} \text{inm}_{ijk}}{abc^2}
\]

(Substitute **ifg** in place of **g** if a frame metric is used.)

**inmc1**

Covariant permutation of the nonmetricity vector components. Defined as

\[
inmc1 = \frac{g \text{inm} - \text{inm} g - g \text{inm}}{abc^2}
\]

**inmc2**

Contravariant permutation of the nonmetricity vector components. Used in the connection coefficients if **inonmet_flag** is **true**. Defined as:
\[ -\text{inm} \cdot \delta_{a}^{c} - \text{inm} \cdot \delta_{b}^{c} \inm_{c}^{a} + g \inm_{c}^{a} \inm_{c}^{b} \quad \text{inmc2} = \frac{-\text{inm} \cdot \delta_{a}^{c} - \text{inm} \cdot \delta_{b}^{c} \inm_{c}^{a} + g \inm_{c}^{a} \inm_{c}^{b}}{ab} \]

(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:

\[ -g \itr^{a} - g \itr^{b} - g \itr^{c} \quad \text{ikt1} = \frac{-g \itr^{a} - g \itr^{b} - g \itr^{c}}{abc} \]

(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:

\[ g \quad \text{ikt1} \quad \text{ikt2} = \frac{g \quad \text{ikt1}}{ab \quad abd} \]

(Substitute \( ifg \) in place of \( g \) if a frame metric is used.)

**itr** Variable

The torsion tensor. For a metric with torsion, repeated covariant differentiation on a scalar function will not commute, as demonstrated by the following example:

```lisp
(%i1) load(itensor);
(%o1) /share/tensor/itensor.lisp
(%i2) imetric:g;
(%o2) g
(%i3) covdiff( covdiff( f( [], []), i), j) - covdiff( covdiff( f( [], []), j), i)
(%i4) ishow(%)$
%4 %2
f ichr2 - f ichr2
,%4 j i ,%2 i j
(%i5) canform(%);
(%o5) 0
(%i6) itorsion_flag:true;
```
The \texttt{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

\begin{equation}
\mathbf{a} \wedge \mathbf{a} = \frac{- \mathbf{a} \cdot \mathbf{a}}{2}
\end{equation}

27.2.7 Exterior algebra
More generally, the product of a p-form and a q-form would be defined as
\[
A \wedge B = \frac{1}{D} \� A \raisebox{0pt}[0pt][0pt]{\mathbf{D}} B
\]
where D stands for the Kronecker-delta.

Other authors, however, prefer a “geometric” definition that corresponds with the notion of the volume element:
\[
a \wedge a = a a - a a
\]
and, in the general case
\[
A \wedge B = \frac{1}{D} \� A \raisebox{0pt}[0pt][0pt]{\mathbf{D}} B
\]
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 \( \sim \). 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);
(%i2) ishow(a([i])~b([j]))$
   a b - b a
   i j i j
   2

(%i3) decsym(a,2,0,[anti(all),[]]);
(%i4) ishow(a([i,j])~b([k]))$
   a b + b a - a b
   i j k i j k i k j
   3

(%i5) igeowedge_flag:true;
(%i6) ishow(a([i])~b([j]))$
   a b - b a
   i j i j

(%i7) ishow(a([i,j])~b([k]))$
```
The vertical bar \( \mid \) 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 \texttt{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 \( \mid \) operator. For instance:

```lisp
(%i1) load(itensor);
(%o1) /share/tensor/itensor.lisp
(%i2) decsym(a,2,0,[anti(all)],[]);
(%o2) done
(%i3) ishow(a([i,j],[])|v)$
%1
(%t3) v a
%1 j
(%i4) ishow(a([j,i],[])|v)$
%1
(%t4) - v a
%1 j
```

Note that it is essential that the tensors used with the \( \mid \) operator be declared totally antisymmetric in their covariant indices. Otherwise, the results will be incorrect.

\textbf{extdiff} \( (\text{expr}, i) \)

Computes the exterior derivative of \text{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 \texttt{igeowedge_flag}. For instance:

```lisp
(%i1) load(itensor);
(%o1) /share/tensor/itensor.lisp
(%i2) ishow(extdiff(v([i]),j))$
%1
(%t2) v - v
%1 j,i i,j
---
2
(%i3) decsym(a,2,0,[anti(all)],[]);
(%o3) done
(%i4) ishow(extdiff(a([i,j]),k))$
%1
(%t4) a - a + a
%1 j,k,i i,k,j i,j,k
---
3
(%i5) igeowedge_flag:true;
(%o5) true
```
\begin{verbatim}
(%i6) ishow(extdiff(v([i]),j))$
(%t6) v - v
    j,i   i,j
(%i7) ishow(extdiff(a([i,j]),k))$
(%t7) - (a - a + a )
    k j,i   k i,j   j i,k

hodge (expr) Function
Compute the Hodge-dual of expr. For instance:

(%i11) load(itensor);
(%o11) /share/tensor/itensor.lisp
(%i12) imetric(g);
(%o12) done
(%i13) idim(4);
(%o13) done
(%i14) icounter:100;
(%o14) 100
(%i15) decsym(A,3,0,[anti(all)],[])$
(%i16) ishow(A([i,j,k],[]))$
(%t16) A
     i j k
(%i17) ishow(canform(hodge(%)))$
(%t17) -----------------------------------------

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
\end{verbatim}
totally antisymmetric covariant tensor field. When set to \texttt{true}, differential forms will agree with the notion of the volume element.

### 27.2.8 Exporting TeX expressions

The \texttt{itensor} package provides limited support for exporting tensor expressions to TeX. Since \texttt{itensor} expressions appear as function calls, the regular Maxima \texttt{tex} command will not produce the expected output. You can try instead the \texttt{tentex} command, which attempts to translate tensor expressions into appropriately indexed TeX objects.

#### \texttt{tentex (expr)}

To use the \texttt{tentex} function, you must first load \texttt{tentex}, as in the following example:

```maxima
(%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]));
$$m1
t1
t1
$$
(%i5) tentex(%$\\textbackslash Gamma\{j\},\textbackslash Gamma\{k\}\textbackslash k1\textbackslash m1\textbackslash i\textbackslash l\textbackslash i\textbackslash j1\textbackslash k1\textbackslash k\textbackslash i\textbackslash j1\textbackslash k1\textbackslash i\textbackslash j$\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbackslash k1\textbackslash l\textbackslash k1\textbackslash i$\textbackslash j1\textbacklace

Note the use of the \texttt{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 \texttt{tentex} function is somewhat experimental.

### 27.2.9 Interfacing with ctensor

The \texttt{itensor} package has the ability to generate Maxima code that can then be executed in the context of the \texttt{ctensor} package. The function that performs this task is \texttt{ic\_convert}.

#### \texttt{ic\_convert (eqn)}

Converts the \texttt{itensor} equation \texttt{eqn} to a \texttt{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 \texttt{diff} taken with respect to \texttt{ct\_coords} subscripted by the derivative index. The Christoffel symbols \texttt{ichr1} and \texttt{ichr2} will be translated to \texttt{lcs} and \texttt{mcs}, respectively and if \texttt{metricconvert} is \texttt{true} then all occurrences of the metric with two covariant (contravariant) indices will be renamed to \texttt{lg} (\texttt{ug}). In addition, \texttt{do} loops
will be introduced summing over all free indices so that the transformed assignment statement can be evaluated by just doing \texttt{ev}. The following examples demonstrate the features of this function.

\begin{verbatim}
(%i1) load(itensor);
(%o1) /share/tensor/itensor.lisp
(%i2) eqn:ishow(t([i,j],[k])=f([],[])*g([l,m],[[]]*a([],[],[m],j)*b([l],[l,k]))$ 

k m l k
(%t2) t = f a b g
i j, j i l m

(%i3) ic_convert(eqn);
(%o3) for i thru dim do (for j thru dim do (for k thru dim do
    t : f sum(sum(diff(a , ct_coords ) b
    i, j, k m  j i, l, k

    g , l, 1, dim), m, 1, dim)) )

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
\end{verbatim}

### 27.2.10 Reserved words

The following Maxima words are used by the \texttt{itensor} package internally and should not be redefined:

\begin{tabular}{ll}
\textbf{Keyword} & \textbf{Comments} \\
\hline
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 & \\
lcpred & \\
_extlc & \\
\end{tabular}
28 ctensor

28.1 Introduction to ctensor

ctensor is a component tensor manipulation package. To use the ctensor package, type \texttt{load(ctensor)}. To begin an interactive session with ctensor, type \texttt{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 \texttt{lg}. Finally, the metric inverse is computed and stored in the matrix \texttt{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.

\begin{verbatim}
(%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;

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.
\end{verbatim}
Enter functional dependencies with the DEPENDS function or 'N' if none
depends([a,d],x);
Do you wish to see the metric?
y;

\[
\begin{bmatrix}
  a & 0 & 0 & 0 \\
  2 &   &   &  \\
  0 & x & 0 & 0 \\
  2 & 2 &   &  \\
  0 & 0 & x \sin(y) & 0 \\
  0 & 0 & 0 & -d
\end{bmatrix}
\]

(%o2) done

(%i3) christof(mcs);

\[
\begin{align*}
\text{mcs} &= \frac{a}{2} \\
\text{mcs} &= \frac{-x}{2} \\
\text{mcs} &= \frac{-x}{2} \\
\text{mcs} &= \frac{-x \cos(y)}{2} \\
\text{mcs} &= \frac{-x \sin(y)}{2} \\
\text{mcs} &= -\frac{\cos(y) \sin(y)}{3}
\end{align*}
\]
28.2 Functions and Variables for ctensor

28.2.1 Initialization and setup

csetup ()

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 (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.

c.t_coordsys (coordinate_system, extra_arg)

c.t_coordsys (coordinate_system)

Sets up a predefined coordinate system and metric. The argument coordinate_system can be one of the following symbols:

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>Dim</th>
<th>Coordinates</th>
<th>Description/comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>cartesian2d</td>
<td>2</td>
<td>[x,y]</td>
<td>Cartesian 2D coordinate system</td>
</tr>
<tr>
<td>polar</td>
<td>2</td>
<td>[r,phi]</td>
<td>Polar coordinate system</td>
</tr>
<tr>
<td>elliptic</td>
<td>2</td>
<td>[u,v]</td>
<td>Elliptic coord. system</td>
</tr>
<tr>
<td>confocalelliptic</td>
<td>2</td>
<td>[u,v]</td>
<td>Confocal elliptic coordinates</td>
</tr>
<tr>
<td>bipolar</td>
<td>2</td>
<td>[u,v]</td>
<td>Bipolar coord. system</td>
</tr>
<tr>
<td>parabolic</td>
<td>2</td>
<td>[u,v]</td>
<td>Parabolic coord. system</td>
</tr>
<tr>
<td>cartesian3d</td>
<td>3</td>
<td>[x,y,z]</td>
<td>Cartesian 3D coordinate system</td>
</tr>
</tbody>
</table>
polarcylindrical 3 [r, theta, z] Polar 2D with cylindrical z
ellipticcylindrical 3 [u, v, z] Elliptic 2D with cylindrical z
confocalellipsoidal 3 [u, v, w] Confocal ellipsoidal
datapolarcylindrical 3 [u, v, z] Bipolar 2D with cylindrical z
paraboliccylindrical 3 [u, v, z] Parabolic 2D with cylindrical z
paraboloidal 3 [u, v, phi] Paraboloidal coords.
conical 3 [u, v, w] Conical coordinates
toroidal 3 [u, v, phi] Toroidal coordinates
spherical 3 [r, theta, phi] Spherical coord. system
oblatespheroidal 3 [u, v, phi] Oblate spheroidal coordinates
oblatespheroidalsqrt 3 [u, v, phi] Oblate spheroidal
prolatespheroidal 3 [u, v, phi] Prolate spheroidal coordinates
prolatespheroidalsqrt 3 [u, v, phi] Prolate spheroidal
ellipsoidal 3 [r, theta, phi] Ellipsoidal coordinates
cartesian4d 4 [x, y, z, t] Cartesian 4D coordinate system
spherical4d 4 [r, theta, eta, phi] Spherical 4D coordinate system
exteriorschwarzschild 4 [t, r, theta, phi] Schwarzschild metric
interiorschwarzschild 4 [t, z, u, v] Interior Schwarzschild metric
kerr_newman 4 [t, r, theta, phi] Charged axially symmetric metric

coordinate_system can also be a list of transformation functions, followed by a list containing the coordinate variables. For instance, you can specify a spherical metric as follows:

(%i1) load(ctensor);
(%o1) /share/tensor/ctensor.mac
(%i2) ct_coordsys([r*cos(theta)*cos(phi), r*cos(theta)*sin(phi), 
          r*sin(theta), [r, theta, phi]]);
(%o2) done
(%i3) lg:trigsimp(lg); 
  [ 1 0 0 ]
  [ ]
  [ 2 ]
(%o3) [ 0 r 0 ]
  [ ]
  [ 2 2 ]
  [ 0 0 r cos(theta) ]
(%i4) ct_coords;
(%o4) [r, theta, phi]
(%i5) dim;
Transformation functions can also be used when \( \text{cframe\_flag} \) is true:

\begin{verbatim}
(%i1) load(ctensor);
(%o1) /share/tensor/ctensor.mac
(%i2) cframe_flag: true;
(%o2) true
(%i3) ct_coordsys([r*cos(theta)*cos(phi), r*cos(theta)*sin(phi),
                  r*sin(theta), [r, theta, phi]]);
(%o3) done
(%i4) fri;
(%o4) {
    [cos(phi)*cos(theta) - cos(phi) r sin(theta) - sin(phi) r cos(theta)]
    [ ]
    [sin(phi)*cos(theta) - sin(phi) r sin(theta) cos(phi) r cos(theta)]
    [ ]
    [ sin(theta) r cos(theta) 0 ]

(%i5) cmetric();
(%o5) false
(%i6) lg: trigsimp(lg);
   [ 1 0 0 ]
   [ ]
   [ 2 ]
   [ 0 r 0 ]
   [ ]
   [ 2 2 ]
   [ 0 0 r cos(theta) ]
\end{verbatim}

The optional argument \texttt{extra\_arg} can be any one of the following:

- \texttt{cylindrical} tells \texttt{ct\_coordsys} to attach an additional cylindrical coordinate.
- \texttt{minkowski} tells \texttt{ct\_coordsys} to attach an additional coordinate with negative metric signature.
- \texttt{all} tells \texttt{ct\_coordsys} to call \texttt{cmetric} and \texttt{christof(false)} after setting up the metric.

If the global variable \texttt{verbose} is set to \texttt{true}, \texttt{ct\_coordsys} displays the values of \texttt{dim}, \texttt{ct\_coords}, and either \texttt{lg} or \texttt{lfg} and \texttt{fri}, depending on the value of \texttt{cframe\_flag}.

\textbf{Function} \texttt{init\_ctensor ()}

Initializes the \texttt{ctensor} package.

The \texttt{init\_ctensor} function reinitializes the \texttt{ctensor} package. It removes all arrays and matrices used by \texttt{ctensor}, resets all flags, resets \texttt{dim} to 4, and resets the frame metric to the Lorentz-frame.
28.2.2 The tensors of curved space

The main purpose of the ctensor package is to compute the tensors of curved space(time), most notably the tensors used in general relativity.

When a metric base is used, ctensor can compute the following tensors:

\[
\begin{align*}
\text{lg} & \quad \text{ug} \\
\text{lcs} & \quad \text{mcs} \quad \text{ric} \quad \text{uric} \\
\text{tracer} & \quad \text{ein} \quad \text{lein} \\
\text{riem} & \quad \text{riem} \quad \text{weyl} \\
\text{uriem}
\end{align*}
\]

ctensor can also work using moving frames. When cframe_flag is set to true, the following tensors can be calculated:

\[
\begin{align*}
\text{lfg} & \quad \text{ufg} \\
\text{fri} & \quad \text{fr} \quad \text{lcs} \quad \text{mcs} \quad \text{riem} \quad \text{ric} \quad \text{uric} \\
\text{lg} & \quad \text{ug} \quad \text{weyl} \quad \text{tracer} \quad \text{ein} \quad \text{lein} \\
\text{riem} & \\
\text{uriem}
\end{align*}
\]

\textbf{christof (dis)}

A function in the ctensor (component tensor) package. It computes the Christoffel symbols of both kinds. The argument dis determines which results are to be immediately displayed. The Christoffel symbols of the first and second kinds are stored in the arrays lcs[i,j,k] and mcs[i,j,k] respectively and defined to be symmetric in the first two indices. If the argument to christof is lcs or mcs then the unique non-zero values of lcs[i,j,k] or mcs[i,j,k], respectively, will be displayed. If the argument is all then the unique non-zero values of lcs[i,j,k] and mcs[i,j,k] will be displayed. If the argument is false then the display of the elements will not occur. The array elements mcs[i,j,k] are defined in such a manner that the final index is contravariant.

\textbf{ricci (dis)}

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) 
This function first computes the covariant components \( \text{ric}[i,j] \) of the Ricci tensor. Then the mixed Ricci tensor is computed using the contravariant metric tensor. If the value of the argument \( \text{dis} \) is true, then these mixed components, \( \text{uric}[i,j] \) (the index \( i \) is covariant and the index \( j \) is contravariant), will be displayed directly. Otherwise, \( \text{ricci}(\text{false}) \) will simply compute the entries of the array \( \text{uric}[i,j] \) without displaying the results.

scurvature ()
Returns the scalar curvature (obtained by contracting the Ricci tensor) of the Riemannian manifold with the given metric.

einstein (dis) 
A function in the \texttt{ctensor} (component tensor) package. \texttt{einstein} computes the mixed Einstein tensor after the Christoffel symbols and Ricci tensor have been obtained (with the functions \texttt{christof} and \texttt{ricci}). If the argument \( \text{dis} \) is true, then the non-zero values of the mixed Einstein tensor \( \text{ein}[i,j] \) will be displayed where \( j \) is the contravariant index. The variable \texttt{rateinstein} will cause the rational simplification on these components. If \texttt{ratfac} is true then the components will also be factored.

leinstein (dis)
Covariant Einstein-tensor. \texttt{leinstein} stores the values of the covariant Einstein tensor in the array \texttt{lein}. The covariant Einstein-tensor is computed from the mixed Einstein tensor \texttt{ein} by multiplying it with the metric tensor. If the argument \( \text{dis} \) is true, then the non-zero values of the covariant Einstein tensor are displayed.

riemann (dis) 
A function in the \texttt{ctensor} (component tensor) package. \texttt{riemann} computes the Riemann curvature tensor from the given metric and the corresponding Christoffel symbols. The following index conventions are used:

\[
R[i,j,k,l] = R = |^{1}_{ijk} - |^{1}_{ij,k} + |^{1}_{ik,j} - |^{1}_{mk} + |^{1}_{ij} - |^{1}_{mj} - |^{1}_{ik} 
\]

This notation is consistent with the notation used by the \texttt{itensor} package and its \texttt{icurvature} function. If the optional argument \( \text{dis} \) is true, the unique non-zero components \( \text{riem}[i,j,k,l] \) 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 \texttt{ratriemann} is true, then rational simplification will be done. If \texttt{ratfac} is true then each of the components will also be factored.

If the variable \texttt{cframe_flag} is false, the Riemann tensor is computed directly from the Christoffel-symbols. If \texttt{cframe_flag} is true, the covariant Riemann-tensor is computed first from the frame field coefficients.

lriemann (dis)
Covariant Riemann-tensor (\texttt{lriem[]}).
Computes the covariant Riemann-tensor as the array \texttt{lriem}. If the argument \texttt{dis} is true, unique non-zero values are displayed.

If the variable \texttt{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 \texttt{riemann}.

\textbf{uriemann (dis)}

Function

Computes the contravariant components of the Riemann curvature tensor as array elements \texttt{uriem[i,j,k,l]}. These are displayed if \texttt{dis} is true.

\textbf{rinvariant ()}

Function

Forms the Kretchmann-invariant (\texttt{kinvariant}) obtained by contracting the tensors \texttt{lriem[i,j,k,l]*uriem[i,j,k,l]}.

This object is not automatically simplified since it can be very large.

\textbf{weyl (dis)}

Function

Computes the Weyl conformal tensor. If the argument \texttt{dis} is true, the non-zero components \texttt{weyl[i,j,k,l]} will be displayed to the user. Otherwise, these components will simply be computed and stored. If the switch \texttt{ratweyl} is set to true, then the components will be rationally simplified; if \texttt{ratfac} is true then the results will be factored as well.

\subsection*{28.2.3 Taylor series expansion}

The \texttt{ctensor} package has the ability to truncate results by assuming that they are Taylor-series approximations. This behavior is controlled by the \texttt{ctayswitch} variable; when set to true, \texttt{ctensor} makes use internally of the function \texttt{ctaylor} when simplifying results.

The \texttt{ctaylor} function is invoked by the following \texttt{ctensor} functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>christof()</td>
<td>For mcs only</td>
</tr>
<tr>
<td>ricci()</td>
<td></td>
</tr>
<tr>
<td>uricci()</td>
<td></td>
</tr>
<tr>
<td>einstein()</td>
<td></td>
</tr>
<tr>
<td>riemann()</td>
<td></td>
</tr>
<tr>
<td>weyl()</td>
<td></td>
</tr>
<tr>
<td>checkdiv()</td>
<td></td>
</tr>
</tbody>
</table>

\textbf{ctaylor ()}

Function

The \texttt{ctaylor} function truncates its argument by converting it to a Taylor-series using \texttt{taylor}, and then calling \texttt{ratdisrep}. This has the combined effect of dropping terms higher order in the expansion variable \texttt{ctayvar}. The order of terms that should be dropped is defined by \texttt{ctaypov}; the point around which the series expansion is carried out is specified in \texttt{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]);
(%o5)
[ - 1 0 0 0 ]
[ ]
[ 0 1 0 0 ]
[ ]
[ 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]);
(%o6)
[ h11 0 0 0 ]
[ ]
[ 0 h22 0 0 ]
[ ]
[ 0 0 h33 0 ]
[ ]
[ 0 0 0 h44 ]
(%i7) depends(l,r);
(%o7) [l(r)]
(%i8) lg:lg+l*h;
(%o8)
[ h11 l - 1 0 0 0 ]
[ ]
[ 0 h22 l + 1 0 0 ]
[ ]
[ 0 0 r + h33 l 0 ]
[ ]
[ 2 2 ]
[ 0 0 0 r sin(theta) + h44 l ]
(%i9) cmetric(false);
(%o9) done
(%i10) einstein(false);
(%o10) done
(%i11) ntermst(ein);
[[[1, 1], 62]
However, if we recompute this example as an approximation that is linear in the variable \( l \), we get much simpler expressions:

```plaintext
(%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]
```
This capability can be useful, for instance, when working in the weak field limit far from a gravitational source.

28.2.4 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)

The frame bracket (fb[]).
Computes the frame bracket according to the following definition:

\[
ifb = \begin{pmatrix} ifr & ifr \\ d,e & e,d \end{pmatrix}
\]

28.2.5 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 ()

Computes a Newman-Penrose null tetrad (np) and its raised-index counterpart (npi).
See petrov for an example.
The null tetrad is constructed on the assumption that a four-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:

\[
np = \frac{(fri + fri)}{\sqrt{2}}
\]

\[
np = \frac{(fri - fri)}{\sqrt{2}}
\]
\[
\text{np} = \frac{(\text{fr}_i + \text{i} \text{fr}_i)}{\sqrt{2}} \\
\text{np} = \frac{(\text{fr}_i - \text{i} \text{fr}_i)}{\sqrt{2}}
\]

\text{psi}(\text{dis}) \quad \text{Function}

Computes the five Newman-Penrose coefficients \(\text{psi}[0]...\text{psi}[4]\). If \text{dis} is set to \text{true}, the coefficients are displayed. See \text{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 \(\text{lg}\) (computed automatically by \text{cmetric} and then \text{ug}). At this point, you can switch back to a coordinate base by setting \text{cframe_flag} to \text{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.

\text{petrov}() \quad \text{Function}

Computes the Petrov classification of the metric characterized by \(\text{psi}[0]...\text{psi}[4]\).

For example, the following demonstrates how to obtain the Petrov-classification of the Kerr metric:

\begin{verbatim}
(%i1) load(ctensor);
(%o1) /share/tensor/ctensor.mac
(%i2) (cframe_flag:true,gcd:spmod,ctrgsimp:true,ratfac:true);
(%o2) true
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3) done
(%i4) ug:invert(lg)$
(%i5) weyl(false);
(%o5) done
(%i6) nptetrad(true);
(%t6) np =
\end{verbatim}
The Petrov classification function is based on the algorithm published in "Classifying geometries in general relativity: III Classification in practice" by Pollney, Skea, and
28.2.6 Torsion and nonmetricity

c tensor 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:

\[
kt = \frac{-g^{im} tr^{jk} - g^{jm} tr^{ki} - tr^{g}}{ij km}
\]

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:

\[
nmc = \frac{-nm D^{ij} - D^{ij} nm + g^{nm g}}{ij m}
\]

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.
28.2.7 Miscellaneous features

\textbf{cttransform} \((M)\)

Function

A function in the \texttt{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 \texttt{transform}.)

\textbf{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. \texttt{deindex} is a global list containing the indices of \(A\) corresponding to these unique differential equations. For the Einstein tensor \((\texttt{ein})\), which is a two dimensional array, if computed for the metric in the example below, \texttt{findde} gives the following independent differential equations:

\begin{verbatim}
(%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]);
(%o4) [ a 0 0 0 ]
[ ]
[ ]
[2]
[ 0 x 0 0 ]
(%o4)
(%i5) depends([a,d],x);
(%o5) [a(x), d(x)]
(%i6) ct_coords:[x,y,z,t];
(%o6) [x, y, z, t]
(%i7) cmetric();
(%o7) done
(%i8) einstein(false);
(%o8) done
(%i9) findde(ein,2);
2
(%o9) [d x - a d + d, 2 a d d x - a (d ) x - a d d x
 x x x x x x
 x x + 2 a d d - 2 a d , a x + a - a]
 x x x
(%i10) deindex;
\end{verbatim}
(%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:

```maxima
(%i1) load(ctensor);
(%o1) /share/tensor/ctensor.mac
(%i2) derivabbrev:true;
(%o2) true
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3) done
(%i4) depends(f,r);
(%o4) [f(r)]
(%i5) cograd(f,g1);
(%o5) done
(%i6) listarray(g1);
(%o6) [0, f, 0, 0]
   r
(%i7) contragrad(f,g2);
(%o7) done
(%i8) listarray(g2);
   f  r - 2 f  m
   r r r
(%o8) [0, ---------------------, 0, 0]
         r
```

**dscalar** () Function

computes the tensor d’Alembertian of the scalar function once dependencies have been declared upon the function. For example:

```maxima
(%i1) load(ctensor);
(%o1) /share/tensor/ctensor.mac
(%i2) derivabbrev:true;
(%o2) true
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3) done
(%i4) depends(p,r);
(%o4) [p(r)]
(%i5) factor(dscalar(p));
   2          2                      2
(%o5) p  r - 2 m p  r + 2 p  r - 2 m p
      r r r r              r r
```
Function \texttt{checkdiv} ()

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 \texttt{div}.

Function \texttt{cgeodesic} (\texttt{dis})

A function in the \texttt{ctensor} (component tensor) package. \texttt{cgeodesic} computes the
geodesic equations of motion for a given metric. They are stored in the array \texttt{geod[i]}. If the argument \texttt{dis} is \texttt{true} then these equations are displayed.

Function \texttt{bdvac} (\texttt{f})

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., \texttt{’p(x)’}. The components of the second rank covariant field tensor are represented by the array \texttt{bd}.

Function \texttt{invariant1} ()

generates the mixed Euler- Lagrange tensor (field equations) for the invariant density
of \( \text{R}^2 \). The field equations are the components of an array named \texttt{inv1}.

Function \texttt{invariant2} ()

*** NOT YET IMPLEMENTED ***
generates the mixed Euler- Lagrange tensor (field equations) for the invariant density
of \texttt{ric[i,j]*uriem[i,j]}. The field equations are the components of an array named \texttt{inv2}.

Function \texttt{bimetric} ()

*** NOT YET IMPLEMENTED ***
generates the field equations of Rosen’s bimetric theory. The field equations are the
components of an array named \texttt{rosen}.

28.2.8 Utility functions

Function \texttt{diagmatrixp} (\texttt{M})

Returns \texttt{true} if \( M \) is a diagonal matrix or (2D) array.

Function \texttt{symmetricp} (\texttt{M})

Returns \texttt{true} if \( M \) is a symmetric matrix or (2D) array.

Function \texttt{ntermst} (\texttt{f})

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.
Function

cdisplay (ten)

displays all the elements of the tensor ten, as represented by a multidimensional array. Tensors of rank 0 and 1, as well as other types of variables, are displayed as with ldisplay. Tensors of rank 2 are displayed as 2-dimensional matrices, while tensors of higher rank are displayed as a list of 2-dimensional matrices. For instance, the Riemann-tensor of the Schwarzschild metric can be viewed as:

```
(%i1) load(ctensor);
  %o1) /share/tensor/ctensor.mac

(%i2) ratfac:true;
   (%o2) true

(%i3) ct_coordsys(exteriorschwarzschild,all);
   (%o3) done

(%i4) riemann(false);
   (%o4) done

(%i5) cdisplay(riem);

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
2 & & & \\
3 m (r - 2 m) & m & 2 m & \\
0 & - \frac{m (r - 2 m)}{4} + \frac{m}{3} - \frac{m}{4} & 0 & 0 \\
4 & 3 & 4 & \\
0 & r & r & r \\
& & & \\
& & & \\
& & & \\
\end{bmatrix}
\]

riem = \[
1, 1 \\
0 & 0 & \frac{m (r - 2 m)}{4} & 0 \\
\]

riem = \[
1, 2 \\
0 & 0 & 0 & 0 \\
\]

riem = \[
2 m (r - 2 m) & \frac{m (r - 2 m)}{4} \\
0 & 0 & 0 & 0 \\
\]

riem = \[
2 m (r - 2 m) & \frac{m (r - 2 m)}{4} \\
0 & 0 & 0 & 0 \\
\]
```

riem = \[
2 m (r - 2 m) & \frac{m (r - 2 m)}{4} \\
0 & 0 & 0 & 0 \\
\]

riem = \[
2 m (r - 2 m) & \frac{m (r - 2 m)}{4} \\
0 & 0 & 0 & 0 \\
\]
1, 3
[ 0 0 0 0 ]
[ ]
[ 0 0 0 0 ]
[ ]
[ 0 0 0 0 ]

\[
\begin{bmatrix}
m(r - 2m) \\
0 & 0 & 0 - \frac{m}{4} \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\]
riem = [ ]
1, 4
[ 0 0 0 0 ]
[ ]
[ 0 0 0 0 ]
[ ]
[ 0 0 0 0 ]

\[
\begin{bmatrix}
r (r - 2m) \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
riem = [ ]
2, 1
[ 0 0 0 0 ]
[ ]
[ 0 0 0 0 ]
[ ]
[ 0 0 0 0 ]

\[
\begin{bmatrix}
2m \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
riem = [ ]
2, 2
[ 0 0 0 - \frac{m}{r (r - 2m)} ]
[ ]
[ 0 0 0 - \frac{m}{r (r - 2m)} ]
[ ]
[ 0 0 0 - \frac{m}{r (r - 2m)} ]

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
\[
\text{riem} = \begin{bmatrix}
2 \\
2, 3 \\
r (r - 2 \, m) \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

\[
\text{riem} = \begin{bmatrix}
2 \\
2, 4 \\
r (r - 2 \, m) \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

\[
\text{riem} = \begin{bmatrix}
m \\
3, 1 \\
-0 & 0 & 0 & 0 \\
r \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

\[
\text{riem} = \begin{bmatrix}
m \\
3, 2 \\
0 & -0 & 0 & 0 \\
r \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

\[
\text{riem} = \begin{bmatrix}
m \\
3, 3 \\
-0 & 0 & 0 & 0 \\
r \\
-0 & 0 & 0 & 0 \\
2 \, m - \, r \\
\end{bmatrix}
\]
\[
\text{riem} = \begin{bmatrix}
2m \\
0 \\
0 \\
0 \\
0 \\
\end{bmatrix}
\]

\[
\begin{align*}
\text{riem} &= \begin{bmatrix}
2 \sin(\theta) \\
0 \\
0 \\
0 \\
0 \\
\end{bmatrix} \\
\text{riem} &= \begin{bmatrix}
2 \sin(\theta) \\
0 \\
0 \\
0 \\
0 \\
\end{bmatrix} \\
\text{riem} &= \begin{bmatrix}
2m \sin(\theta) \\
0 \\
0 \\
0 \\
0 \\
\end{bmatrix}
\end{align*}
\]
\[
\text{riem} = \begin{bmatrix}
\frac{2 \, m \sin(\theta)}{r} & 0 \\
0 & -\frac{m \sin(\theta)}{r} \\
0 & 0 \\
0 & 0 & \frac{2}{r}
\end{bmatrix}
\]

(%o5) done

\textbf{deleten} \ ((L, n))

Function

Returns a new list consisting of \( L \) with the \( n \)’th element deleted.

\textbf{28.2.9 Variables used by ctensor}

\textbf{dim}

Option variable

Default value: 4

An option in the \texttt{ctensor} (component tensor) package. \texttt{dim} is the dimension of the manifold with the default 4. The command \texttt{dim: n} will reset the dimension to any other value \( n \).

\textbf{diagmetric}

Option variable

Default value: \texttt{false}

An option in the \texttt{ctensor} (component tensor) package. If \texttt{diagmetric} is \texttt{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 \texttt{csetup} if a diagonal metric is specified.

\textbf{ctrgsimp}

Option variable

Causes trigonometric simplifications to be used when tensors are computed. Presently, \texttt{ctrgsimp} affects only computations involving a moving frame.

\textbf{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 \texttt{fri} and the frame metric \texttt{lfg}. For computations using a Cartesian frame, \texttt{lfg} should be the unit matrix of the appropriate dimension; for computations in a Lorentz frame, \texttt{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
The covariant frame metric. By default, it is initialized to the 4-dimensional Lorentz frame with signature (+,+,+,-). Used when cframe_flag is true.

ufg
The inverse frame metric. Computed from lfg when cmetric is called while cframe_flag is set to true.

riem
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
The covariant Riemann tensor. Computed by lriemann.

uriem
The contravariant Riemann tensor. Computed by uriemann.

ric
The mixed Ricci-tensor. Computed by ricci.

uric
The contravariant Ricci-tensor. Computed by uricci.

lg
The metric tensor. This tensor must be specified (as a dim by dim matrix) before other computations can be performed.

ug
The inverse of the metric tensor. Computed by cmetric.

weyl
The Weyl tensor. Computed by weyl.

fb
Frame bracket coefficients, as computed by frame_bracket.

kinvariant
The Kretchmann invariant. Computed by rinvariant.

np
A Newman-Penrose null tetrad. Computed by nptetrad.
npi
The raised-index Newman-Penrose null tetrad. Computed by \texttt{nptetrad}. Defined as \texttt{ug.np}. The product \texttt{np.transpose(npi)} is constant:

\begin{verbatim}
(\%i39) trigsimp(np.transpose(npi));
[ [ 0  -1  0  0 ]
 [ [ -1  0  0  0 ]
 [ [ 0  0  1  0 ]
(\%o39)
\end{verbatim}

\texttt{tr}
User-supplied rank-3 tensor representing torsion. Used by \texttt{contortion}.

\texttt{kt}
The contortion tensor, computed from \texttt{tr} by \texttt{contortion}.

\texttt{nm}
User-supplied nonmetricity vector. Used by \texttt{nonmetricity}.

\texttt{nmc}
The nonmetricity coefficients, computed from \texttt{nm} by \texttt{nonmetricity}.

\texttt{tensorkill}
System variable
Variable indicating if the tensor package has been initialized. Set and used by \texttt{csetup}, reset by \texttt{init_ctensor}.

\texttt{ct_coords}
Option variable
Default value: []
An option in the \texttt{ctensor} (component tensor) package. \texttt{ct_coords} contains a list of coordinates. While normally defined when the function \texttt{csetup} is called, one may redefine the coordinates with the assignment \texttt{ct_coords: [j1, j2, ..., jn]} where the j's are the new coordinate names. See also \texttt{csetup}.

28.2.10 Reserved names
The following names are used internally by the \texttt{ctensor} package and should not be redefined:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_lg()</td>
<td>Evaluates to lfg if frame metric used, lg otherwise</td>
</tr>
<tr>
<td>_ug()</td>
<td>Evaluates to ufg if frame metric used, ug otherwise</td>
</tr>
<tr>
<td>cleanup()</td>
<td>Removes items from the deindex list</td>
</tr>
<tr>
<td>contract4()</td>
<td>Used by psi()</td>
</tr>
<tr>
<td>filemet()</td>
<td>Used by csetup() when reading the metric from a file</td>
</tr>
<tr>
<td>findde1()</td>
<td>Used by findde()</td>
</tr>
</tbody>
</table>
findde2() Used by findde()
findde3() Used by findde()
kdelta() Kronecker-delta (not generalized)
newmet() Used by csetup() for setting up a metric interactively
setflags() Used by init_ctensor()
readvalue() 
reimp() 
sermet() Used by csetup() for entering a metric as Taylor-series
txyzsum() 
tmetric() Frame metric, used by cmetric() when cframe_flag: true
triemann() Riemann-tensor in frame base, used when cframe_flag: true
tricci() Ricci-tensor in frame base, used when cframe_flag: true
ttrc() Ricci rotation coefficients, used by christof()
yesp()

28.2.11 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.

<table>
<thead>
<tr>
<th>New Name</th>
<th>Old Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctaylor()</td>
<td>DLGTAYLOR()</td>
<td>Taylor-series expansion of an expression</td>
</tr>
<tr>
<td>lgeod[]</td>
<td>EM</td>
<td>Geodesic equations</td>
</tr>
<tr>
<td>ein[]</td>
<td>G[]</td>
<td>Mixed Einstein-tensor</td>
</tr>
<tr>
<td>ric[]</td>
<td>LR[]</td>
<td>Mixed Ricci-tensor</td>
</tr>
<tr>
<td>rici()</td>
<td>LRICCICOM()</td>
<td>Compute the mixed Ricci-tensor</td>
</tr>
<tr>
<td>ctaypov</td>
<td>MINP</td>
<td>Maximum power in Taylor-series expansion</td>
</tr>
<tr>
<td>cgeodesic()</td>
<td>MOTION</td>
<td>Compute geodesic equations</td>
</tr>
<tr>
<td>ct_coords</td>
<td>OMEGA</td>
<td>Metric coordinates</td>
</tr>
<tr>
<td>ctayvar</td>
<td>PARAM</td>
<td>Taylor-series expansion variable</td>
</tr>
<tr>
<td>lriem[]</td>
<td>R[]</td>
<td>Covariant Riemann-tensor</td>
</tr>
<tr>
<td>uriemann()</td>
<td>RAISERIEMANN()</td>
<td>Compute the contravariant Riemann-tensor</td>
</tr>
<tr>
<td>ratriemann</td>
<td>RATRIEMAN</td>
<td>Rational simplif. of the Riemann-tensor</td>
</tr>
<tr>
<td>uric[]</td>
<td>RICCI[]</td>
<td>Contravariant Ricci-tensor</td>
</tr>
<tr>
<td>uricci()</td>
<td>RICCICOM()</td>
<td>Compute the contravariant Ricci-tensor</td>
</tr>
<tr>
<td>cmetric()</td>
<td>SETMETRIC()</td>
<td>Set up the metric</td>
</tr>
<tr>
<td>ctaypt</td>
<td>TAYPT</td>
<td>Point for Taylor-series expansion</td>
</tr>
<tr>
<td>ctayswitch</td>
<td>TAYSWITCH</td>
<td>Taylor-series setting switch</td>
</tr>
<tr>
<td>csetup()</td>
<td>TSETUP()</td>
<td>Start interactive setup session</td>
</tr>
<tr>
<td>ctransform()</td>
<td>TTRANSFORM()</td>
<td>Interactive coordinate transformation</td>
</tr>
<tr>
<td>uriem[]</td>
<td>UR[]</td>
<td>Contravariant Riemann-tensor</td>
</tr>
<tr>
<td>weyl[]</td>
<td>W[]</td>
<td>(3,1) Weyl-tensor</td>
</tr>
</tbody>
</table>
29 atensor

29.1 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.:

\[
i = v_1 \quad j = v_2 \quad k = v_1 \cdot v_2
\]

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);

(%i6) q[1,1]:1;
(%o6) 1
(%i7) for i thru adim do q[i+1,1]:q[i+1,1]*v[i];
(%o7) done
(%i8) q[1,4]:q[4,1]*v[1];
(%o8) v_1 \cdot v_2
(%i9) for i from 2 thru 4 do for j from 2 thru 4 do q[i,j]:atensimp(q[i,1]*q[i,j]);
(%o9) done
(%i10) q;
```
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:
dotscrules: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.

29.2 Functions and Variables for atensor

init_atensor (alg_type, opt_dims) Function
init_atensor (alg_type) Function

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, degene-
rate, 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 sym-
plectic algebra, opt_dims can be up to two nonnegative integers, representing the
nondegenerate and degenerate dimensions, respectively. If any \( \text{opt_dims} \) values are
supplied, \( \text{atensor} \) will configure the values of \( \text{adim} \) and \( \text{aform} \) appropriately. Other-
wise, \( \text{adim} \) will default to 0 and \( \text{aform} \) will not be defined.

\( \text{lie_envelop} \): The algebra of the Lie envelope is defined by the commutation relation
\[ u.v - v.u = 2\cdot \text{av}(u,v) \] where \( \text{av} \) is an antisymmetric function.

The \( \text{init_atensor} \) function also recognizes several predefined algebra types:

- \( \text{complex} \) implements the algebra of complex numbers as the Clifford algebra \( \text{Cl}(0,1) \).
The call \( \text{init_atensor(complex)} \) is equivalent to \( \text{init_atensor(clifford,0,0,1)} \).

- \( \text{quaternion} \) implements the algebra of quaternions. The call \( \text{init_atensor(quaternion)} \) is equivalent to \( \text{init_atensor(clifford,0,0,2)} \).

- \( \text{pauli} \) implements the algebra of Pauli-spinors as the Clifford-algebra \( \text{Cl}(3,0) \). A call
to \( \text{init_atensor(pauli)} \) is equivalent to \( \text{init_atensor(clifford,3)} \).

- \( \text{dirac} \) implements the algebra of Dirac-spinors as the Clifford-algebra \( \text{Cl}(3,1) \). A call
to \( \text{init_atensor(dirac)} \) is equivalent to \( \text{init_atensor(clifford,3,0,1)} \).

\( \text{atensimp(e)} \) Function
Simplifies an algebraic tensor expression \( e \) according to the rules configured by a
call to \( \text{init_atensor} \). Simplification includes recursive application of commutation
relations and resolving calls to \( \text{sf} \), \( \text{af} \), and \( \text{av} \) where applicable. A safeguard is used
to ensure that the function always terminates, even for complex expressions.

\( \text{alg_type} \) Function
The algebra type. Valid values are \( \text{universal} \), \( \text{grassmann} \), \( \text{clifford} \), \( \text{symmetric} \),
\( \text{symplectic} \) and \( \text{lie_envelop} \).

\( \text{adim} \) Variable
Default value: 0
The dimensionality of the algebra. \( \text{atensor} \) uses the value of \( \text{adim} \) to determine if an
indexed object is a valid base vector. See \( \text{abasep} \).

\( \text{aform} \) Variable
Default value: \( \text{ident}(3) \)
Default values for the bilinear forms \( \text{sf} \), \( \text{af} \), and \( \text{av} \). The default is the identity matrix
\( \text{ident}(3) \).

\( \text{asymbol} \) Variable
Default value: \( v \)
The symbol for base vectors.

\( \text{sf}\) \((u, v)\) Function
A symmetric scalar function that is used in commutation relations. The default
implementation checks if both arguments are base vectors using \( \text{abasep} \) and if that
is the case, substitutes the corresponding value from the matrix \( \text{aform} \).
af \( (u, v) \)  
An antisymmetric scalar function that is used in commutation relations. The default implementation checks if both arguments are base vectors using \texttt{abasep} and if that is the case, substitutes the corresponding value from the matrix \texttt{aform}.

av \( (u, v) \)  
An antisymmetric function that is used in commutation relations. The default implementation checks if both arguments are base vectors using \texttt{abasep} and if that is the case, substitutes the corresponding value from the matrix \texttt{aform}.

For instance:

```
(%i1) load(atensor);
(%o1) /share/tensor/atensor.mac
(%i2) adim:3;
(%o2) 3
(%i3) aform:matrix([0,3,-2],[-3,0,1],[2,-1,0]);
     [ 0  3  -2 ]
    [         ]
(%o3) [ -3  0  1 ]
    [         ]
     [  2 -1  0 ]
(%i4) asymbol:x;
(%o4) x
(%i5) av(x[1],x[2]);
(%o5) x
     3
```

abasep \( (v) \)  
Checks if its argument is an \texttt{atensor} base vector. That is, if it is an indexed symbol, with the symbol being the same as the value of \texttt{asymbol}, and the index having a numeric value between 1 and \texttt{adim}.
30 Series

30.1 Introduction to Series

Maxima contains functions \texttt{taylor} and \texttt{powerseries} for finding the series of differentiable functions. It also has tools such as \texttt{nusum} 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.

30.2 Functions and Variables for Series

\texttt{cauchysum} \\
Option variable \hspace{1cm} Default value: \texttt{false}

When multiplying together sums with inf as their upper limit, if \texttt{sumexpand} is \texttt{true} and \texttt{cauchysum} is \texttt{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:

\begin{verbatim}
(%i1) sumexpand: false$
(%i2) cauchysum: false$
(%i3) s: sum (f(i), i, 0, inf) * sum (g(j), j, 0, inf);
inf    inf
     \ /  \
    ( > f(i)) > g(j)
     /    /  \
        i = 0 j = 0

(%i4) sumexpand: true$
(%i5) cauchysum: true$
(%i6) ''s;
inf    inf
     \ /  \
    > > g(i1 - i2) f(i2)
     /    /  \
        i1 = 0 i2 = 0
\end{verbatim}

\texttt{deftaylor} \hspace{1cm} Function

\texttt{deftaylor} \hspace{1cm} \texttt{(f_1(x_1), expr_1, ..., f_n(x_n), expr_n)}

For each function \texttt{f_i} of one variable \texttt{x_i}, \texttt{deftaylor} defines \texttt{expr_i} as the Taylor series about zero. \texttt{expr_j} is typically a polynomial in \texttt{x_j} or a summation; more general expressions are accepted by \texttt{deftaylor} without complaint.

\texttt{powerseries} \hspace{1cm} \texttt{(f_i(x_i), x_i, 0)} returns the series defined by \texttt{deftaylor}. 

deftaylor returns a list of the functions \( f_1, \ldots, f_n \). deftaylor evaluates its arguments.

Example:

\[
(\%i1) \text{deftaylor (} f(x), x^2 + \text{sum}(x^i/(2^i*i!^2), i, 4, \text{inf}))\; ;
(\%o1) \quad [f]
\]

\[
(\%i2) \text{powerseries (} f(x), x, 0)\; ;
\inf
\downarrow \quad x^2 + 2
\]

\[
(\%o2) \quad \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i3) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i3) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]

\[
(\%i4) \text{powerseries (} f(x), x, 0)\; ;
\]
Chapter 30: Series 421

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) \text{niceindicespref: \[p, q, r, s, t, u\]}$
\]

\[
(\%i2) \text{product (sum (f (foo + i*j*bar), foo, 1, inf), bar, 1, inf);}$
\]

\[
(\%o2) \text{f(bar i j + foo)}$
\]

\[
(\%i3) \text{niceindices (\%);}$
\]

\[
(\%o3) \text{f(i j q + p)}$
\]

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:

\[
(\%i1) \text{nusum (n*n!, n, 0, n);}$
\]
Dependent equations eliminated: (1)

(%o1) (n + 1)! - 1

(%i2) nusum (n^4*4^n/binomial(2*n,n), n, 0, n); 4 3 2 n
2 (n + 1) (63 n + 112 n + 18 n - 22 n + 3) 4 2
(%o2) ------------------------------------------------------------------ - ------
693 binomial(2 n, n) 3 11 7

(%i3) unsum (%%, n);

4 n
n 4
(%o3) ----------------
binomial(2 n, n)

(%i4) unsum (prod (i^2, i, 1, n), n); n - 1
2
(%o4) ( ! ! i ) (n - 1) (n + 1)
! !
i = 1

(%i5) nusum (%%, n, 1, n);

Dependent equations eliminated: (2 3)

n
2
(%o5)

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 numerator_degree_bound + denominator_degree_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
1 + x + x + x + ...
(%o1)/T/

(%i2) pade (%%, 1, 1);

[ - -----

1
x - 1

]
Chapter 30: Series

(%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);

(%o3)
\[
\frac{\frac{1}{7}x^7 - \frac{1}{6}x^6 + \frac{1}{32}x^3}{2}
\]

(%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) [- \frac{520256329 x^5 - 96719020632 x^4 - 489651410240 x^3}{47041365435 x + 381702613848 x + 1360678489152 x^2 + 2856700692480 x + 3370143559680 x + 2386516803584}]

\textbf{powerdisp}

Option variable

Default value: false

When \textbf{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.

\textbf{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):
\[
\begin{align*}
\text{inf} \\
\Rightarrow \quad \sum_{n=0}^{\infty} \frac{b (x - a)^n}{n!} \\
\text{n = 0}
\end{align*}
\]

If \texttt{powerseries} is unable to expand \texttt{expr}, \texttt{taylor} may give the first several terms of the series.

When \texttt{verbose} is \texttt{true}, \texttt{powerseries} prints progress messages.

\begin{verbatim}
(%i1) verbose: true$
(%i2) powerseries (log(sin(x)/x), x, 0);
can't expand

\text{log(sin(x))}

so we'll try again after applying the rule:

\[
\frac{d}{dx} \log(sin(x)) = \frac{i \cot(x) dx}{\sin(x)} - \log(x)
\]

in the first simplification we have returned:

\[
\left[ i \cot(x) dx - \log(x) \right]
\]

\[
\begin{align*}
\text{inf} \\
\Rightarrow \quad \sum_{i=1}^{\infty} \frac{(-1)^{i} 2 i \text{bern}(2 i) x}{i! (2 i)!} \\
\text{i1 = 1}
\end{align*}
\]

\end{verbatim}

\text{psexpand}

Option variable

Default value: \texttt{false}

When \texttt{psexpand} is \texttt{true}, an extended rational function expression is displayed fully expanded. The switch \texttt{ratexpand} has the same effect.

When \texttt{psexpand} is \texttt{false}, a multivariate expression is displayed just as in the rational function package.

When \texttt{psexpand} is \texttt{multi}, then terms with the same total degree in the variables are grouped together.
Function \texttt{revert}(\textit{expr}, \textit{x})

Function \texttt{revert2}(\textit{expr}, \textit{x}, \textit{n})

These functions return the reversion of \textit{expr}, a Taylor series about zero in the variable \textit{x}. \texttt{revert} returns a polynomial of degree equal to the highest power in \textit{expr}. \texttt{revert2} returns a polynomial of degree \textit{n}, which may be greater than, equal to, or less than the degree of \textit{expr}.

\texttt{load("revert")} loads these functions.

Examples:

```markdown
(%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
5 4 3 2
(%i5) taylor(log(x+1), x, 0, 6);
2 3 4 5 6
x x x x x
(%o5)/T/ x - -- + -- - -- + -- - -- + . . .
2 3 4 5 6
(%i6) ratsimp(revert(t, x) - taylor(log(x+1), x, 0, 6));
(%o6)
(%i7) revert2(t, x, 4);
4 3 2
x x x
(%o7)
```

Function \texttt{taylor}(\textit{expr}, \textit{x}, \textit{a}, \textit{n})

Function \texttt{taylor}(\textit{expr}, [\textit{x}, \textit{a}], \textit{n})

Function \texttt{taylor}(\textit{expr}, [\textit{x}, \textit{a}], 'asympt')

Function \texttt{taylor}(\textit{expr}, [\textit{x}, \textit{a}], \textit{n}, \textit{a1}, \textit{a2}, \ldots)

Function \texttt{taylor}(\textit{expr}, [\textit{x}, \textit{a}], \textit{n}, \textit{a1}, \textit{a2}, \ldots)

\texttt{taylor}(\textit{expr}, \textit{x}, \textit{a}, \textit{n}) expands the expression \textit{expr} in a truncated Taylor or Laurent series in the variable \textit{x} around the point \textit{a}, containing terms through \((x - a)^n\).

If \textit{expr} is of the form \(f(x)/g(x)\) and \(g(x)\) has no terms up to degree \textit{n} then \texttt{taylor} attempts to expand \(g(x)\) up to degree \(2n\). If there are still no nonzero terms, \texttt{taylor} doubles the degree of the expansion of \(g(x)\) so long as the degree of the expansion is less than or equal to \(n \times \text{taylordepth}\).
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, 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 maxtaylor is true, then during algebraic manipulation of (truncated) Taylor series, taylor tries to retain as many terms as are known to be correct.

When pseexpand is true, an extended rational function expression is displayed fully expanded. The switch ratexpand has the same effect. When pseexpand is false, a multivariate expression is displayed just as in the rational function package. When pseexpand 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);
\(%o1\)/T/ 1 + \frac{(a + 1)x}{2} - \frac{(a + 2a + 1)x^2}{8} + \frac{(3a + 9a + 9a - 1)x^3}{48} + \ldots

(\%i2) \%^2;
\(%o2\)/T/ 1 + (a + 1)x - \frac{x^3}{6} + \ldots

(\%i3) taylor(sqrt(x + 1), x, 0, 5);
\(%o3\)/T/ 1 + \frac{x}{2} - \frac{x^3}{8} + \frac{x^5}{16} - \frac{x^7}{128} + \ldots

(\%i4) \%^2;
\(%o4\)/T/ 1 + x + \ldots

(\%i5) product((1 + x^-1)^2.5, i, 1, inf)/(1 + x^-2);
Chapter 30: Series 427

\[
\inf \\
/\equiv\\
\begin{align*}
\text{! ! } i & \quad 2.5 \\
\text{! ! } (x + 1) & \\
\text{! ! } i = 1 \\
\end{align*}
\]

\[(%o5) \quad \frac{2}{x + 1} \]

\[(%i6) \text{ ev } (\text{taylor}(%, x, 0, 3), \text{keepfloat});\]

\[(%o6)/T/ \quad 1 + 2.5 x + 3.375 x^2 + 6.5625 x^3 + \ldots\]

\[(%i7) \text{ taylor } (1/\log (x + 1), x, 0, 3);\]

\[(%o7)/T/ \quad 1 + x + x^2 + x^3 + x^4 + \ldots\]

\[(%i8) \text{ taylor } (\cos(x) - \sec(x), x, 0, 5);\]

\[(%o8)/T/ \quad 2 x - x^2 + \ldots\]

\[(%i9) \text{ taylor } ((\cos(x) - \sec(x))^3, x, 0, 5);\]

\[(%o9)/T/ \quad 0 + \ldots\]

\[(%i10) \text{ taylor } (1/(\cos(x) - \sec(x))^3, x, 0, 5);\]

\[(%o10)/T/ \quad \frac{1}{6} + \frac{1}{4} x + \frac{11}{24} x^2 + \frac{347}{15120} x^3 + \frac{6767}{604800} x^4 + \frac{15377}{7983360} x^5 + \ldots\]

\[(%i11) \text{ taylor } (\sqrt{1 - k^2 \sin(x)^2}, x, 0, 6);\]

\[(%o11)/T/ \quad 1 - \frac{1}{2} x + \ldots\]

\[(%i12) \text{ taylor } ((x + 1)^n, x, 0, 4);\]
\[
\frac{(n - n) x}{n - 3 n + 2 n} \times \frac{(n - 3 n + 2 n) x}{2} + \frac{4}{2} \times \frac{(n - 6 n + 11 n - 6 n) x}{6} + \frac{4}{24} \times \ldots
\]

\[
\text{%i13) taylor (sin (y + x), x, 0, 3, y, 0, 3);}
\]

\[
y - \frac{y \ y}{6} + \frac{y \ y}{2} + \ldots + \frac{1}{12} \times (\ldots) x + \frac{1}{12} \times (\ldots) x + \ldots
\]

\[
\text{%i14) taylor (sin (y + x), [x, y], 0, 3);}
\]

\[
y + x - \frac{1}{6} x + \frac{1}{2} y + \frac{1}{3} y + \frac{1}{4} y + \ldots
\]

\[
\text{%i16) taylor (1/sin (y + x), [x, y], 0, 3);}
\]

\[
\frac{1}{x + y} + \frac{1}{y} + \frac{1}{3} \times (\ldots) x + \frac{1}{3} \times (\ldots) x + \ldots
\]

**taylorder**
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 \( n \times 2^{\text{taylorder}}. \)

**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]);

(\%o1)/T/ - (y - a) - 2 a (y - a) + (1 - a )
  2
+ (1 - a - 2 a (y - a) - (y - a) ) x
  2
+ (1 - a - 2 a (y - a) - (y - a) ) x
  2
+ (1 - a - 2 a (y - a) - (y - a) ) x + . . .

(%i2) taylorinfo(%);

(\%o2) \[y, a, inf\], \[x, 0, 3\]

Function

taylorp (expr)

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.
Function \texttt{taytorat}(\texttt{expr})

Converts \texttt{expr} from \texttt{taylor} form to canonical rational expression (CRE) form. The effect is the same as \texttt{rat (ratdisrep (expr))}, but faster.

Function \texttt{trunc}(\texttt{expr})

Annotates the internal representation of the general expression \texttt{expr} so that it is displayed as if its sums were truncated Taylor series. \texttt{expr} is not otherwise modified.

Example:

\begin{verbatim}
(%i1) expr: x^2 + x + 1;
   2
(%o1) x + x + 1
(%i2) trunc (expr);
   2
(%o2) 1 + x + x + . . .
(%i3) is (expr = trunc (expr));
   true
\end{verbatim}

Function \texttt{unsum}(\texttt{f, n})

Returns the first backward difference \(f(n) - f(n - 1)\). Thus \texttt{unsum} in a sense is the inverse of \texttt{sum}.

See also \texttt{nusum}.

Examples:

\begin{verbatim}
(%i1) g(p) := p*4^n/binomial(2*n,n);noun
   n
(%o1) g(p) := ----------------
       binomial(2 n, n)
(%i2) g(n^4);
   4 n
   4
(%o2) ----------------
       binomial(2 n, n)
(%i3) nusum (%i, n, 0, n);
   4 n
   4
    3 2  n 2
(%o3) ----------------------------- - ------
    693 binomial(2 n, n) 3 11 7
(%i4) unsum (%i, n);
   4 n
   4
(%o4) ----------------
       binomial(2 n, n)
\end{verbatim}

Option variable \texttt{verbose}

Default value: \texttt{false}

When \texttt{verbose} is \texttt{true}, \texttt{powerseries} prints progress messages.
30.3 Poisson series

**intopois** (**a**)  
Converting **a** into a Poisson encoding.

**outofpois** (**a**)  
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.

**poisdif** (**a, b**)  
Differentiates **a** with respect to **b**. **b** must occur only in the trig arguments or only in the coefficients.

**poisexpt** (**a, b**)  
Functionally identical to **intopois** (**a**^**b**). **b** must be a positive integer.

**poisint** (**a, b**)  
Integrates in a similarly restricted sense (to **poisdif**). Non-periodic terms in **b** are dropped if **b** is in the trig arguments.

**poislim**  
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(5-1)+1, 2^5(5-1)], or [-15,16], but it can be set to [-2^**(n-1)+1, 2^**(n-1)]

**poismap** (**series, sinfn, cosfn**)  
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**)  
Is functionally identical to **intopois** (**a + b**).

**poissimp** (**a**)  
Converts **a** into a Poisson series for **a** in general representation.

**poisson**  
The symbol /P/ follows the line label of Poisson series expressions.
Function \texttt{poissubst} \((a, b, c)\)

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., \(6u + 4v\)).

2. Where \(b\) is other than those variables, then \(a\) must also be free of those variables, and furthermore, free of sines or cosines.

\texttt{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, \(\texttt{poissubst}(u, v, \cos(v), \%e, 3)\) yields \(\cos(u) * (1 - \%e^2/2) - \sin(u) * (\%e - \%e^3/6)\).

Function \texttt{poistimes} \((a, b)\)

Is functionally identical to \texttt{intopois}(a*b).

Function \texttt{poistrim}()

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 \(u, v, \ldots, z\) in a term. Terms for which \texttt{poistrim} is \texttt{true} (for the coefficients of that term) are eliminated during multiplication.

Function \texttt{printpois} \((a)\)

Prints a Poisson series in a readable format. In common with \texttt{outofpois}, it will convert \(a\) into a Poisson encoding first, if necessary.
31 Number Theory

31.1 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 \(\text{zerobern}\) is \(\text{false}\).
See also \(\text{burn}\).

(%i1) \text{zerobern: true}\$
(%i2) \text{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) \text{zerobern: false}\$
(%i4) \text{map (bern, [0, 1, 2, 3, 4, 5, 6, 7, 8]);}
1 1 1 5 691 7 3617 43867
(%o4) [1, - -, - -, - --, --, - ----, -, - ----, ----]
2 6 30 66 2730 6 510 798

bernpoly \((x, n)\) 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 \((\text{bfloat})\); \(n\) is the number of digits in the return value.

bhzeta \((s, h, n)\) Function
Returns the Hurwitz zeta function for the arguments \(s\) and \(h\). The return value is a big float \((\text{bfloat})\); \(n\) is the number of digits in the return value.
The Hurwitz zeta function is defined as

\[
\zeta (s, h) = \sum_{k=0}^{\infty} \frac{1}{(k + h)^s}
\]

\(\text{load ("bffac")}\) loads this function.

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:
Function \texttt{burn} \((n)\)

Returns a rational number, which is an approximation of the \(n\)‘th Bernoulli number for integer \(n\). \texttt{burn} exploits the observation that (rational) Bernoulli numbers can be approximated by (transcendental) zetas with tolerable efficiency:

\[
B(2n) = \frac{n - 1}{2n} \frac{1 - 2n}{\zeta(2n) (2n)!} \frac{2}{\pi n}
\]

\texttt{burn} may be more efficient than \texttt{bern} for large, isolated \(n\) as \texttt{bern} computes all the Bernoulli numbers up to index \(n\) before returning. \texttt{burn} invokes the approximation for even integers \(n > 255\). For odd integers and \(n \leq 255\) the function \texttt{bern} is called.

\texttt{load(“bffac“)} loads this function. See also \texttt{bern}.

Function \texttt{cf} \((\text{expr})\)

Converts \text{expr} into a continued fraction. \text{expr} is an expression comprising continued fractions and square roots of integers. Operands in the expression may be combined with arithmetic operators. Aside from continued fractions and square roots, factors in the expression must be integer or rational numbers. Maxima does not know about operations on continued fractions outside of \text{cf}.

\text{cf} evaluates its arguments after binding \texttt{listarith} to \texttt{false}. \text{cf} returns a continued fraction, represented as a list.

A continued fraction \(a + 1/(b + 1/(c + \ldots))\) is represented by the list \([a, b, c, \ldots]\). The list elements \(a, b, c, \ldots\) must evaluate to integers. \text{expr} may also contain \texttt{sqrt} \((n)\) where \(n\) is an integer. In this case \text{cf} will give as many terms of the continued fraction as the value of the variable \texttt{cflength} times the period.

A continued fraction can be evaluated to a number by evaluating the arithmetic representation returned by \text{cfdisrep}. See also \text{cfexpand} for another way to evaluate a continued fraction.

See also \text{cfdisrep}, \text{cfexpand}, and \text{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, 27]
  (%i2) cf ((3/17)*[1, -2, 5]/sqrt(11) + (8/13));
  (%o2) [0, 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)

Constructs and returns an ordinary arithmetic expression of the form \( a + 1/(b + 1/(c + \ldots)) \) from the list representation of a continued fraction \([a, b, c, \ldots] \).

  (%i1) cf ([1, 2, -3] + [1, -2, 1]);
  (%o1) [1, 1, 2]
  (%i2) cfdisrep (%);
  (%o2) \[
    \frac{1}{1 + \frac{1}{1 + \frac{1}{2}}}
  \]

**cfexpand** (x)

Returns a matrix of the numerators and denominators of the last (column 1) and next-to-last (column 2) convergents of the continued fraction \( x \).

  (%i1) cf (rat (ev (%pi, numer)));
'rat' replaced 3.141592653589793 by 103993/33102 = 3.141592653011902

(%o1) [3, 7, 15, 1, 292]

(%i2) cfexpand (%);
    [ 103993 355 ]
    [ ]
    [ 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**) 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**

Function

Returns the **n**'th Euler number for nonnegative integer **n**.

For the Euler-Mascheroni constant, see %gamma.

(%i1) map (euler, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
(%o1) [1, 0, -1, 0, 5, 0, -61, 0, 1385, 0, -50521]

**%gamma**

Constant

The Euler-Mascheroni constant, 0.5772156649015329 ....
Chapter 31: Number Theory

factorial \((x)\)

Function

Represents the factorial function. Maxima treats `factorial (x)` the same as `x!`. See `!`.

factorial_expand

Default value: false

The option variable `factorial_expand` controls the simplification of expressions like \((n+1)!\), where \(n\) is an integer. See `!` for an example.

fib \((n)\)

Function

Returns the \(n\)'th Fibonacci number. `fib(0)` equal to 0 and `fib(1)` equal to 1, and `fib (-n)` equal to `(-1)^(n + 1) * fib(n)`.

After calling `fib`, `prevfib` is equal to `fib (x - 1)`, the Fibonacci number preceding the last one computed.

\[
(\text{fib}(0), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = \[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55\]
\]

fibtophi \((expr)\)

Function

Expresses Fibonacci numbers in `expr` in terms of the constant \%phi, which is \((1 + \sqrt{5})/2\), approximately 1.61803399.

Examples:

\[
(\text{fibtophi}(\text{fib}(n))); \\
\frac{\%phi^n - (1 - \%phi)^n}{2 \%phi - 1}
\]

\[
(\text{fibtophi}(\text{fib}(n+1))); \\
\frac{\%phi^n - (1 - \%phi)^{n+1}}{2 \%phi - 1}
\]

\[
(\text{fibtophi}(\text{fib}(n-1))); \\
\frac{\%phi^\text{n-1} - (1 - \%phi)^n}{2 \%phi - 1}
\]

\[
(\text{fibtophi}(\text{fib}(n))); \\
\frac{\%phi^{n+1} - (1 - \%phi)^{n+1}}{2 \%phi - 1}
\]

ifactors \((n)\)

Function

For a positive integer \(n\) returns the factorization of \(n\). If \(n=p_1^{e_1} \cdot p_k^{e_k}\) is the decomposition of \(n\) into prime factors, ifactors returns `[[p_1, e_1], ..., [p_k, e_k]]`.

Factorization methods used are trial divisions by primes up to 9973, Pollard’s rho method and elliptic curve method.
(%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

inrt (x, n) Function
Returns the integer n’th root of the absolute value of x.
(%i1) l: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$
(%i2) map (lambda ([a], inrt (10^a, 3)), l);
(%o2) [2, 4, 10, 21, 46, 100, 215, 464, 1000, 2154, 4641, 10000]

inv_mod (n, m) 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

jacobi (p, q) Function
Returns the Jacobi symbol of p and q.
(%i1) l: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$
(%i2) map (lambda ([a], jacobi (a, 9)), l);
(%o2) [1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0]

lcm (expr_1, ..., expr_n) Function
Returns the least common multiple of its arguments. The arguments may be general expressions as well as integers.
load ("functs") loads this function.

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 + 2)!
(%i2) minfactorial (%);
(%o2) 1
   ---------------
          (n + 1) (n + 2)

next_prime (n) Function
Returns the smallest prime bigger than n.
(%i1) next_prime(27);
(%o1) 29
partfrac \((expr, var)\)

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.

\[
1/(1+x)^2 - 2/(1+x) + 2/(2+x);
\]

\[
\frac{2}{x+2} - \frac{2}{x+1} + \frac{1}{x+1};
\]

\[
\text{(x + 1)}
\]

\[
\text{(i1)}\ 1/(1+x)^2 - 2/(1+x) + 2/(2+x);
\]

\[
\frac{2}{x+2} - \frac{2}{x+1} + \frac{1}{x+1};
\]

\[
\text{(x + 1)}
\]

\[
\text{(i2)}\ \text{ratsimp (i1)};
\]

\[
- \frac{x}{3 + 2x + 5x + 2};
\]

\[
\text{(i3)}\ \text{partfrac (i2, x)};
\]

\[
\frac{2}{x+2} - \frac{2}{x+1} + \frac{1}{x+1};
\]

\[
\text{(x + 1)}
\]

power_mod \((a, n, m)\)

Uses a modular algorithm to compute \(a^n \mod m\) 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.

\[
\text{(i1)}\ \text{power_mod(3, 15, 5)};
\]

\[
2
\]

\[
\text{(i2)}\ \text{mod(3^15,5)};
\]

\[
2
\]

\[
\text{(i3)}\ \text{power_mod(2, -1, 5)};
\]

\[
3
\]

\[
\text{(i4)}\ \text{inv_mod(2,5)};
\]

\[
3
\]

primep \((n)\)

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 \(n\) 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 that 10^-15.

primep_number_of_tests

Option variable

Default value: 25

Number of Miller-Rabin’s tests used in primep.
prev_prime \( (n) \)

Returns the greatest prime smaller than \( n \).

\[
\text{(%i1)} \quad \text{prev_prime}(27); \\
\text{(%o1)} \quad 23
\]

qunit \( (n) \)

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 \( a^2 - nb^2 = 1 \).

\[
\text{(%i1)} \quad \text{qunit}(17); \\
\text{(%o1)} \quad \sqrt{17} + 4 \\
\text{(%i2)} \quad \text{expand}(\% \times (\sqrt{17} - 4)); \\
\text{(%o2)} \quad 1
\]

totient \( (n) \)

Returns the number of integers less than or equal to \( n \) which are relatively prime to \( n \).

zerobern

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 \( (n) \)

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 \( \text{limit}(zeta(x), x, 1) \) from above and below.

The Riemann zeta function distributes over lists, matrices, and equations.

See also bfzeta and zeta%pi.

Examples:

\[
\text{(%i1)} \quad \text{zeta}([-2, -1, 0, 0.5, 2, 3, 1+%i]); \\
\text{(%o1)} \quad \begin{array}{c} 0, -\infty, -\infty, -1.460354508809586, ----, zeta(3), \\
\end{array} \\
\quad \begin{array}{c} 12 \quad 2 \quad 6 \\
\end{array} \\
\quad \begin{array}{c} \text{zeta}(\%i + 1) \end{array}
\]

\[
\text{(%i2)} \quad \text{limit}(\text{zeta}(x), x, 1, \text{plus}); \\
\text{(%o2)} \quad \text{inf} \\
\text{(%i3)} \quad \text{limit}(\text{zeta}(x), x, 1, \text{minus}); \\
\text{(%o3)} \quad \text{minf}
\]
**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
 (%o2) %pi  ----
     90

(%i3) zeta%pi: false$
(%i4) zeta (4);  
(%o4) zeta(4)
```
32 Symmetries

32.1 Introduction to Symmetries

sym is a package for working with symmetric groups of polynomials.

It was written for Macsyma-Symbolics by Annick Valibouze\(^1\). The algorithms are described in the following papers:


32.2 Functions and Variables for Symmetries

32.2.1 Changing bases

\texttt{comp2pui} \((n, L)\)

\begin{verbatim}
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 \(h1, h2, \text{ etc.}\). 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 \((3, [4, g])\);
\(\%o1\) \([4, g, 2 h2 - g , 3 h3 - g h2 + g (g - 2 h2)]\)
\end{verbatim}

\texttt{ele2pui} \((m, L)\)

\begin{verbatim}
Function

\end{verbatim}

\begin{verbatim}
\text{goes from the elementary symmetric functions to the complete functions. Similar to comp2ele and comp2pui.}
\end{verbatim}

Other functions for changing bases: \texttt{comp2ele}.

---

1 \(\text{www-calfor.lip6.fr/~avb}\)
2 \(\text{www.stix.polytechnique.fr/publications/1984-1994.html}\)
3 \(\text{www-calfor.lip6.fr/~avb/DonneesTelechargeables/MesArticles/issac89ACWValibouze.pdf}\)
4 \(\text{www.stix.polytechnique.fr/publications/1984-1994.html}\)
ele2comp \((m, L)\)

Goes from the elementary symmetric functions to the complete functions. Similar to comp2ele and comp2pui.

Other functions for changing bases: comp2ele.

elem \((ele, sym, lvar)\)

 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:

\[
\begin{align*}
(%i1) & \text{elem ([3, 7], } x^4 - 2*x*y, [x, y]); \\
(%o1) & 7 (e3 - 7 e2 + 7 (49 - e2)) + 21 e3 \\
& \quad + (- 2 (49 - e2) - 2) e2 \\
(%i2) & \text{ratsimp (%);} \\
(%o2) & 28 e3 + 2 e2 - 198 e2 + 2401
\end{align*}
\]

Other functions for changing bases: comp2ele.

mon2schur \((L)\)

The list \(L\) represents the Schur function \(S_L\): we have \(L = [i_1, i_2, \ldots, i_q]\), with \(i_1 \leq i_2 \leq \ldots \leq i_q\). The Schur function \(S_{i_1, i_2, \ldots, i_q}\) is the minor of the infinite matrix \(h_{i-j}\), \(i \geq 1, j \geq 1\), consisting of the \(q\) first rows and the columns \(i_1 + 1, i_2 + 2, \ldots, i_q + q\).

This Schur function can be written in terms of monomials by using \texttt{treinat} and \texttt{kostka}. The form returned is a symmetric polynomial in a contracted representation in the variables \(x_1, x_2, \ldots\)

\[
\begin{align*}
(%i1) & \text{mon2schur ([1, 1, 1]);} \\
(%o1) & x1 x2 x3 \\
(%i2) & \text{mon2schur ([3]);} \\
& 2 3 \\
(%o2) & x1 x2 x3 + x1 x2 + x1 \\
(%i3) & \text{mon2schur ([1, 2]);} \\
& 2 \\
(%o3) & 2 x1 x2 x3 + x1 x2
\end{align*}
\]

which means that for 3 variables this gives:

\[
2 x1 x2 x3 + x1^2 x2 + x2^2 x1 + x1^2 x3 + x3^2 x1 \\
+ x2^2 x3 + x3^2 x2
\]

Other functions for changing bases: comp2ele.
**multi_elem** \((l_{\text{elem}}, \text{multi}_{pc}, l_{\text{var}})\)

Function decomposes a multi-symmetric polynomial in the multi-contracted form \texttt{multi}_{pc} in the groups of variables contained in the list of lists \(l_{\text{var}}\) in terms of the elementary symmetric functions contained in \(l_{\text{elem}}\).

\[
\begin{align*}
\%i1 & \quad \text{multi}_{\text{elem}}([[2, e1, e2], [2, f1, f2]], a*x + a^2 + x^3, \\
& \quad \qquad \quad [[x, y], [a, b]]); \\
\%o1 & \quad -2 f2 + f1 (f1 + e1) - 3 e1 e2 + e1 \\
\%i2 & \quad \text{ratsimp} \%; \\
\%o2 & \quad -2 f2 + f1 + e1 f1 - 3 e1 e2 + e1
\end{align*}
\]

Other functions for changing bases: \texttt{comp2ele}.

**multi_pui**

Function is to the function \texttt{pui} what the function \texttt{multi}_{\text{elem}} is to the function \texttt{elem}.

\[
\begin{align*}
\%i1 & \quad \text{multi}_{\text{pui}}([[2, p1, p2], [2, t1, t2]], a*x + a^2 + x^3, \\
& \quad \qquad \quad [[x, y], [a, b]]); \\
\%o1 & \quad t2 + p1 t1 + \frac{p1}{2} \quad \frac{p1}{2} \\
\%i2 & \quad \text{ratsimp} \%;
\end{align*}
\]

**pui** \((L, \text{sym}, l_{\text{var}})\)

Function decomposes the symmetric polynomial \texttt{sym}, in the variables in the list \(l_{\text{var}}\), 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 \texttt{sym}. If values are missing in the list \(L\), formal values of the type \(p1, p2\), etc. will be added. The polynomial \texttt{sym} may be given in three different forms: contracted (\texttt{elem} should then be 1, its default value), partitioned (\texttt{elem} should be 3), or extended (i.e. the entire polynomial, and \texttt{elem} should then be 2). The function \texttt{pui} is used in the same way.

\[
\begin{align*}
\%i1 & \quad \text{pui}; \\
\%o1 & \quad 1 \\
\%i2 & \quad \text{pui}([[3, a, b], u*x*y*z, \ [x, y, z]]); \\
\%o2 & \quad \frac{a (a - b) u}{6} \quad \frac{(a b - p3) u}{3} \\
\%i3 & \quad \text{ratsimp} \%; \\
\%o3 & \quad \frac{(2 p3 - 3 a b + a) u}{6}
\end{align*}
\]

Other functions for changing bases: \texttt{comp2ele}.
Function \texttt{pui2comp}(n, lpui)

renders the list of the first \(n\) complete functions (with the length first) in terms of
the power functions given in the list \texttt{lpui}. If the list \texttt{lpui} is empty, the cardinal is \(n\),
otherwise it is its first element (as in \texttt{comp2ele} and \texttt{comp2pui}).

\begin{verbatim}
(%i1) pui2comp (2, []);
  2
[2, p1, -------]
  2
(%o1) [2, p1, -------]

(%i2) pui2comp (3, [2, a1]);
  2
2 a1 (p2 + a1)
  2
2 p3 + --------- + a1 p2
  2
p2 + a1

(%o2) [2, a1, -------, --------------------------]

(%i3) ratsimp (%);

2 3
p2 + a1 2 p3 + 3 a1 p2 + a1

(%o3) [2, a1, -------, --------------------]

2 6
\end{verbatim}

Other functions for changing bases: \texttt{comp2ele}.

Function \texttt{pui2ele}(n, lpui)

effects the passage from power functions to the elementary symmetric functions. If
the flag \texttt{pui2ele} is \texttt{girard}, it will return the list of elementary symmetric functions
from 1 to \(n\), and if the flag is \texttt{close}, it will return the \(n\)-th elementary symmetric
function.

Other functions for changing bases: \texttt{comp2ele}.

Function \texttt{puireduc}(n, lpui)

lpui is a list whose first element is an integer \(m\). \texttt{puireduc} gives the first \(n\) power
functions in terms of the first \(m\).

\begin{verbatim}
(%i1) puireduc (3, [2]);
  2
2 p1 (p1 - p2)
  2
[2, p1, p2, p1 p2 - -----------]

(%o1) [2, p1, p2, p1 p2 - -----------]

(%i2) ratsimp (%);

2 3
3 p1 p2 - p1

(%o2) [2, p1, p2, -----------]

2
\end{verbatim}

Function \texttt{schur2comp}(P, l\texttt{var})

\(P\) is a polynomial in the variables of the list \(l\texttt{var}\). Each of these variables represents
a complete symmetric function. In \(l\texttt{var}\) 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.

\[
\text{(%i1) schur2comp (h1*h2 - h3, [h1, h2, h3]);}
\]
\[
\text{(%o1)}
\]
\[
\text{(%i2) schur2comp (a*h3, [h3]);}
\]
\[
\text{(%o2)}
\]

### 32.2.2 Changing representations

**cont2part** $(pc, lvar)$

Function

returns the partitioned polynomial associated to the contracted form $pc$ whose variables are in $lvar$.

\[
\text{(%i1) pc: 2*a^3*b*x^4*y + x^5};
\]
\[
\text{(%o1)}
\]
\[
\text{(%i2) cont2part (pc, [x, y]);}
\]
\[
\text{(%o2)}
\]

**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 $\text{explose}$ performs the inverse operation. The function $\text{tcontract}$ tests the symmetry of the polynomial.

\[
\text{(%i1) psym: explode (2*a^3*b*x^4*y, [x, y, z]);}
\]
\[
\text{(%o1)}
\]
\[
\text{(%i2) contract (psym, [x, y, z]);}
\]
\[
\text{(%o2)}
\]

**explose** $(pc, lvar)$

Function

returns the symmetric polynomial associated with the contracted form $pc$. The list $lvar$ contains the variables.

\[
\text{(%i1) explode (a*x + 1, [x, y, z]);}
\]
\[
\text{(%o1)}
\]

**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 $lvar$. 
part2cont ([[2*a^3*b, 4, 1]], [x, y]);

(\%i1)
2 a b x y

partpol (psym, lvar)
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)
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)
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.

32.2.3 Groups and orbits

direct ([p_1, ..., p_n], y, f, [lvar_1, ..., lvar_n])
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]]);
(\%o1) y - e1 f1 y

2
- 4 e2 f2 - (e1 - 2 e2) (f1 - 2 f2) + e1 f1
+ ------------------------------
2
(\%i2) ratsimp (\%);
(\%o2) y - e1 f1 y + (e1 - 4 e2) f2 + e2 f1
Chapter 32: Symmetries

(%i3) ratsimp (direct ([z^3-e1*z^2+e2*z-e3,z^2 - f1* z + f2],
 z, b*v + a*u, [[u, v], [a, b]]));

\[ 6 5 2 2 2 4 \]
\[ y - 2 e1 f1 y + ((2 e1 - 6 e2) f2 + (2 e2 + e1 ) f1 ) y \]
\[ 3 3 3 \]
\[ + ((9 e3 + 5 e1 e2 - 2 e1 ) f1 f2 + (- 2 e3 - 2 e1 e2) f1 ) y \]
\[ 2 2 4 2 \]
\[ + ((9 e2 - 6 e1 e2 + e1 ) f2 \]
\[ 2 2 2 2 4 \]
\[ + (- 9 e1 e3 - 6 e2 + 3 e1 e2) f1 f2 + (2 e1 e3 + e2 ) f1 ) \]
\[ 2 2 2 2 2 2 \]
\[ y + (((9 e1 - 27 e2) e3 + 3 e1 e2 - e1 e2) f1 f2 \]
\[ 2 2 3 5 \]
\[ + ((15 e2 - 2 e1 ) e3 - e1 e2 ) f1 f2 - 2 e2 e3 f1 ) y \]
\[ 2 3 3 2 2 2 2 \]
\[ + (- 27 e3 + (18 e1 e2 - 4 e1 ) e3 - 4 e2 + e1 e2 ) f2 \]
\[ 2 2 2 2 2 2 \]
\[ + (27 e3 + (e1 - 9 e1 e2) e3 + e2 ) f1 f2 \]
\[ 2 4 2 6 \]
\[ + (e1 e2 e3 - 9 e3 ) f1 f2 + e3 f1 \]

Finding the polynomial whose roots are the sums \( a+u \) where \( a \) is a root of \( z^2-e_1z+e_2 \) and \( u \) is a root of \( z^2 - f_1z + f_2 \).

(%i1) ratsimp (direct ([z^2-e1* z + e2, z^2 - f1* z + f2],
 z, a + u, [[u], [a]]));

\[ 4 3 2 \]
\[ y + (- 2 f1 - 2 e1) y + (2 f2 + f1 + 3 e1 f1 + 2 e2 \]
\[ 2 2 \]
\[ + e1 ) y + ((- 2 f1 - 2 e1) f2 - e1 f1 + (- 2 e2 - e1 ) f1 \]
\[ 2 2 \]
\[ - 2 e1 e2) y + f2 + (e1 f1 - 2 e2 + e1 ) f2 + e2 f1 + e1 e2 f1 \]
\[ 2 \]
\[ + e2 \]

direct accepts two flags: elementaires and puissances (default) which allow decomposing the symmetric polynomials appearing in the calculation into elementary symmetric functions, or power functions, respectively.

Functions of sym used in this function:
multi_orbit (so orbit), pui_direct, multi_elem (so elem), multi_pui (so pui),
pui2ele, ele2pui (if the flag direct is in puissances).

**multi_orbit** \((P, \text{lvar}_1, \text{lvar}_2, ..., \text{lvar}_p)\)

Function

\(P\) is a polynomial in the set of variables contained in the lists \text{lvar}_1, \text{lvar}_2, ..., \text{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.

\[
\begin{align*}
\text{%i1} & \text{ multi_orbit (a*x + b*y, \{x, y\}, \{a, b\});} \\
\text{%o1} & \quad \text{[b y + a x, a y + b x]} \\
\text{%i2} & \text{ multi_orbit (x + y + 2*a, \{x, y\}, \{a, b, c\});} \\
\text{%o2} & \quad \text{[y + x + 2 c, y + x + 2 b, y + x + 2 a]}
\end{align*}
\]

Also see: orbit for the action of a single symmetric group.

**multsym** \((\text{ppart}_1, \text{ppart}_2, n)\)

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

\[
\begin{align*}
\text{%i1} & \text{ multsym ([[3, 1], [2, 1, 1]], [[5, 2]], 2);} \\
\text{%o1} & \quad \text{[[10, 3, 1], [15, 3, 0], [15, 2, 1]]}
\end{align*}
\]

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, explode, part2cont, partpol, tcontract, tpartpol.

**orbit** \((P, \text{lvar})\)

Function

computes the orbit of the polynomial \(P\) in the variables in the list \text{lvar} under the action of the symmetric group of the set of variables in the list \text{lvar}.

\[
\begin{align*}
\text{%i1} & \text{ orbit (a*x + b*y, \{x, y\});} \\
\text{%o1} & \quad \text{[a y + b x, b y + a x]} \\
\text{%i2} & \text{ orbit (2*x + x^2, \{x, y\});} \\
\text{%o2} & \quad \text{[y + 2 y, x + 2 x]}
\end{align*}
\]

See also multi_orbit for the action of a product of symmetric groups on a polynomial.

**pui_direct** \((\text{orbite}, \text{lvar}_1, ..., \text{lvar}_n, [d_1, d_2, ..., d_n])\)

Function

Let \(f\) be a polynomial in \(n\) blocks of variables \text{lvar}_1, ..., \text{lvar}_n. Let \(c_i\) be the number of variables in \text{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 \text{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. \(\text{c}_1 \text{l} \text{d}_1, \text{c}_2 \text{l} \text{d}_2, \text{ldots}, \text{c}_n \text{l} \text{d}_n\).

Let \(SD\) be the product of the symmetric groups \(S_{d_1} \times S_{d_2} \times \cdots \times S_{d_n}\). The function pui_direct returns the first \(n\) power functions of \(SD(f)\) deduced from the power functions of \(SC(f)\), where \(n\) is the size of \(SD(f)\).
The result is in multi-contracted form w.r.t. SD, i.e. only one element is kept per orbit, under the action of SD.

(%i1) l: [[x, y], [a, b]];
(%o1) [[x, y], [a, b]]
(%i2) pui_direct (multi_orbit (a*x + b*y, 1), l, [2, 2]);
(%o2) [a x, 4 a b x y + a x ]
(%i3) pui_direct (multi_orbit (a*x + b*y, 1), l, [3, 2]);
(%o3) [2 a x, 4 a b x y + 2 a x , 3 a b x y + 2 a x , 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 3 2 2 3 3 4 4 5 5 6 6 9 x y + 12 a x y + 3 x + 6 a x + 12 a x + 8 a ]

32.2.4 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

ltreillis (n, m) Function returns the list of partitions of weight n and length m.

(%i1) ltreillis (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: ltreillis, treillis and treinat.
Function \texttt{treillis} \((n)\)

returns all partitions of weight \(n\).

\begin{verbatim}
(%i1) treillis (4);
(%o1) [[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
\end{verbatim}

See also: \texttt{lgtreillis}, \texttt{ltreillis} and \texttt{treinat}.

Function \texttt{treinat} \((part)\)

returns the list of partitions inferior to the partition \texttt{part} w.r.t. the natural order.

\begin{verbatim}
(%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]]
\end{verbatim}

See also: \texttt{lgtreillis}, \texttt{ltreillis} and \texttt{treillis}.

32.2.5 Polynomials and their roots

Function \texttt{ele2polynome} \((L, z)\)

returns the polynomial in \(z\) s.t. the elementary symmetric functions of its roots are in the list \(L = [n, e_1, \ldots, e_n]\), where \(n\) is the degree of the polynomial and \(e_i\) the \(i\)-th elementary symmetric function.

\begin{verbatim}
(%i1) ele2polynome ([2, e1, e2], z);
              2
(%o1) z - e1 z + e2
(%i2) polynome2ele (x^7 - 14*x^5 + 56*x^3 - 56*x + 22, x);
(%o2) [7, 0, -14, 0, 56, 0, -56, -22]
(%i3) ele2polynome ([7, 0, -14, 0, 56, 0, -56, -22], x);
              5   3
(%o3) x - 14 x + 56 x - 56 x + 22
\end{verbatim}

The inverse: \texttt{polynome2ele} \((P, z)\).

Also see: \texttt{polynome2ele}, \texttt{pui2polynome}.

Function \texttt{polynome2ele} \((P, x)\)

gives the list \(l = [n, e_1, \ldots, e_n]\) where \(n\) is the degree of the polynomial \(P\) in the variable \(x\) and \(e_i\) is the \(i\)-the elementary symmetric function of the roots of \(P\).

\begin{verbatim}
(%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);
              5   3
(%o2) x - 14 x + 56 x - 56 x + 22
\end{verbatim}

The inverse: \texttt{ele2polynome} \((l, x)\)
prodrac \((L, k)\)  
Function  
\(L\) is a list containing the elementary symmetric functions on a set \(A\). \texttt{prodrac} returns the polynomial whose roots are the \(k\) by \(k\) products of the elements of \(A\).  
Also see \texttt{somrac}.

\[\texttt{pui2polynome} (x, \textit{lpui})\]  
Function  
calculates the polynomial in \(x\) whose power functions of the roots are given in the list \(\textit{lpui}\).

\[
\begin{align*}
(\%i1) & \text{ pui;} \\
(\%o1) & 1 \\
(\%i2) & \text{kill(labels);} \\
(\%o0) & \text{done} \\
(\%i1) & \text{polynome2ele}\left(x^3 - 4x^2 + 5x - 1, x\right); \\
(\%o1) & [3, 4, 5, 1] \\
(\%i2) & \text{ele2pui}\left(3, \%\right); \\
(\%o2) & [3, 4, 6, 7] \\
(\%i3) & \text{pui2polynome}\left(x, \%\right); \\
(\%o3) & x^3 - 4x^2 + 5x - 1
\end{align*}
\]

See also: \texttt{polynome2ele}, \texttt{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 \texttt{prodrac}.

### 32.2.6 Resolvents

\[\texttt{resolvante} (P, x, f, [x_1, \ldots, x_d])\]  
Function  
calculates the resolvent of the polynomial \(P\) in \(x\) of degree \(n >= d\) by the function \(f\) expressed in the variables \(x_1, \ldots, x_d\). For efficiency of computation it is important to not include in the list \([x_1, \ldots, x_d]\) variables which do not appear in the transformation function \(f\).

To increase the efficiency of the computation one may set flags in \texttt{resolvante} so as to use appropriate algorithms:

If the function \(f\) is unitary:

- A polynomial in a single variable,
- linear,
- alternating,
- a sum,
- symmetric,
- a product,
- the function of the Cayley resolvent (usable up to degree 5)
\[(x1*x2 + x2*x3 + x3*x4 + x4*x5 + x5*x1 - (x1*x3 + x3*x5 + x5*x2 + x2*x4 + x4*x1))^2\] general,

the flag of resolvante may be, respectively:

- unitaire,
- lineaire,
- alternee,
- somme,
- produit,
- cayley,
- generale.

(%i1) resolvante: unitaire$

(%i2) resolvante (x^7 - 14*x^5 + 56*x^3 - 56*x + 22, x, x^3 - 1, [x]);

" resolvante unitaire " [7, 0, 28, 0, 168, 0, 1120, - 154, 7840, - 2772, 56448, - 33880, 413952, - 352352, 3076668, - 3363360, 23114112, - 30494464, 175230832, - 267412992, 1338886528, - 2292126760]

3 6 3 9 6 3
[x - 1, x - 2 x + 1, x - 3 x + 3 x - 1, 12 9 6 3 15 12 9 6 3
x - 4 x + 6 x - 4 x + 1, x - 5 x + 10 x - 10 x + 5 x
18 15 12 9 6 3
- 1, x - 6 x + 15 x - 20 x + 15 x - 6 x + 1,
21 18 15 12 9 6 3
x - 7 x + 21 x - 35 x + 35 x - 21 x + 7 x - 1] [- 7, 1127, - 6139, 431767, - 5472047, 201692519, - 3603982011]

7 6 5 4 3 2
(%o2) y + 7 y - 539 y - 1841 y + 51443 y + 315133 y
+ 376999 y + 125253

(%i3) resolvante: lineaire$

(%i4) resolvante (x^4 - 1, x, x1 + 2*x2 + 3*x3, [x1, x2, x3]);

" resolvante lineaire "

24 20 16 12 8
(%o4) y + 80 y + 7520 y + 1107200 y + 49475840 y
+ 344489984 y + 655360000
Chapter 32: Symmetries

(%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]]);
" resolvante generale "
 24  20  16  12  8
(%o8) y + 80 y + 7520 y + 1107200 y + 49475840 y
 4
+ 344489984 y + 655360000

(%i9) resolvante : lineaire$
(%i10) resolvante (x^4 - 1, x, x1 + x2 + x3, [x1, x2, x3]);
" resolvante lineaire "
 4
(%o10) y - 1

(%i11) resolvante : symetrique$
(%i12) resolvante (x^4 - 1, x, x1 + x2 + x3, [x1, x2, x3]);
" resolvante symetrique "
 4
(%o12) y - 1

(%i13) resolvante (x^4 + x + 1, x, x1 - x2, [x1, x2]);
" resolvante symetrique "
 6  2
(%o13) y - 4 y - 1

(%i14) resolvante : alternee$
(%i15) resolvante (x^4 + x + 1, x, x1 - x2, [x1, x2]);
" resolvante alternee "
 12  8  6  4  2
(%o15) y + 8 y + 26 y - 112 y + 216 y + 229

(%i16) resolvante : produit$
(%i17) resolvante (x^7 - 7*x + 3, x, x1*x2*x3, [x1, x2, x3]);
" resolvante produit "
35 33 29 28 27 26
(%o17) y  - 7 y  - 1029 y  + 135 y  + 7203 y  - 756 y
24 23 22 21 20
+ 1323 y  + 352947 y  - 46305 y  - 2463339 y  + 324135 y
19 18 17 16
- 30618 y  - 453789 y  - 40246444 y  + 282225202 y
14 12 11 10
+ 44274492 y  + 155098503 y  + 12252303 y  + 2893401 y
9 8 7 6
- 171532242 y  + 6751269 y  + 2657205 y  - 94517766 y
5 3
- 3720087 y  + 26040609 y  + 14348907
(%i18) resolvante: symetrique$
(%i19) resolvante (x^7 - 7*x + 3, x, x1*x2*x3, [x1, x2, x3]);
" resolvante symetrique "
35 33 29 28 27 26
(%o19) y  - 7 y  - 1029 y  + 135 y  + 7203 y  - 756 y
24 23 22 21 20
+ 1323 y  + 352947 y  - 46305 y  - 2463339 y  + 324135 y
19 18 17 16
- 30618 y  - 453789 y  - 40246444 y  + 282225202 y
14 12 11 10
+ 44274492 y  + 155098503 y  + 12252303 y  + 2893401 y
9 8 7 6
- 171532242 y  + 6751269 y  + 2657205 y  - 94517766 y
5 3
- 3720087 y  + 26040609 y  + 14348907
(%i20) resolvante: cayley$
(%i21) resolvante (x^5 - 4*x^2 + x + 1, x, a, []);
" resolvante de Cayley "
6 5 4 3 2
(%o21) x  - 40 x  + 4080 x  - 92928 x  + 3772160 x  + 37880832 x
+ 93392896
For the Cayley resolvent, the 2 last arguments are neutral and the input polynomial must necessarily be of degree 5.

See also:
resolvante_bipartite, resolvante_produit_sym, resolvante_unitaire, resolvante_alternee1, resolvante_klein, resolvante_klein3, resolvante_vierer, resolvante_diedrale.

**resolvante_alternee1** (*P*, *x*)
Function
Calculates the transformation *P*(*x*) of degree *n* by the function \(\prod_{1 \leq i < j \leq n-1} (x_i - x_j)\).

See also:
resolvante_produit_sym, resolvante_unitaire, resolvante, resolvante_klein, resolvante_klein3, resolvante_vierer, resolvante_diedrale, resolvante_alternee1.

**resolvante_bipartite** (*P*, *x*)
Function
Calculates the transformation of *P*(*x*) of even degree *n* by the function \(x_1 x_2 \cdots x_{n/2} + x_{n/2+1} \cdots x_n\).

\[
\begin{align*}
(\%i1) \text{resolvante_bipartite} (x^6 + 108, x); \\
10 & 8 & 6 & 4 \\
(\%o1) y & - 972 y & + 314928 y & - 34012224 y
\end{align*}
\]

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 \(x_1 x_2 x_4 + x_4\).

\[
\begin{align*}
(\%i1) \text{resolvante_diedrale} (x^5 - 3x^4 + 1, x); \\
15 & 12 & 11 & 10 & 9 & 8 & 7 \\
(\%o1) x & - 21 x & - 81 x & - 21 x & + 207 x & + 1134 x & + 2331 x \\
& 6 & 5 & 4 & 3 & 2 \\
& - 945 x & - 4970 x & - 18333 x & - 29079 x & - 20745 x & - 25326 x \\
& - 697
\end{align*}
\]

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_klein, resolvante_klein3, resolvante_vierer, resolvante_diedrale.
Function \texttt{resolvante\_klein3} \((P, x)\) calculates the transformation of \(P(x)\) by the function \(x_1 x_2 x_4 + x_4\).

See also:
\texttt{resolvante\_produit\_sym, resolvante\_unitaire, resolvante\_alternee1, resolvante\_klein, resolvante\_vierer, resolvante\_diedrale}.

Function \texttt{resolvante\_produit\_sym} \((P, x)\) calculates the list of all product resolvents of the polynomial \(P(x)\).

\begin{verbatim}
(%i1) resolvante_produit_sym (x^5 + 3*x^4 + 2*x - 1, x);
      5  4  3  2  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
  4  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
\end{verbatim}

See also:
\texttt{resolvante, resolvante\_unitaire, resolvante\_alternee1, resolvante\_klein, resolvante\_klein3, resolvante\_vierer, resolvante\_diedrale}.

Function \texttt{resolvante\_unitaire} \((P, Q, x)\) computes the resolvent of the polynomial \(P(x)\) by the polynomial \(Q(x)\).

See also:
\texttt{resolvante\_produit\_sym, resolvante, resolvante\_alternee1, resolvante\_klein, resolvante\_klein3, resolvante\_vierer, resolvante\_diedrale}.

Function \texttt{resolvante\_vierer} \((P, x)\) computes the transformation of \(P(x)\) by the function \(x_1 x_2 x_3 - x_3 x_4\).

See also:
\texttt{resolvante\_produit\_sym, resolvante\_unitaire, resolvante\_alternee1, resolvante\_klein, resolvante\_klein3, resolvante, resolvante\_diedrale}.
32.2.7 Miscellaneous

\textbf{multinomial} \((r, \text{part})\)

where \(r\) is the weight of the partition \(\text{part}\). This function returns the associate multinomial coefficient: if the parts of \(\text{part}\) are \(i_1, i_2, \ldots, i_k\), the result is \(r!/(i_1! i_2! \ldots i_k!)\).

\textbf{permut} \((L)\)

returns the list of permutations of the list \(L\).
33 Groups

33.1 Functions and Variables for Groups

Function **todd_coxeter** (relations, subgroup)

Function **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,Hg2,Hg3,...]. This can be seen internally in the variable **todd_coxeter_state**.

Example:

```lisp
(%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);
   (p(i,i), j, 1, n-1, i, 1, 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);
   (concat(x,i).concat(x,j))
   (%i3) symet(5);
   ([x1 , (x1 . x2) , x2 , (x1 . x3) , (x2 . x3) ,
     x3 , (x1 . x4) , (x2 . x4) , (x3 . x4) , x4 ]
   <2> <2> <2> <3> <2> <2> <3> <2>
   (%i4) todd_coxeter(%o3);
   Rows tried 426
   (120)
   (%i5) todd_coxeter(%o3,[x1]);
   Rows tried 213
   (60)
   (%i6) todd_coxeter(%o3,[x1,x2]);
   Rows tried 71
   (20)
```
34 Runtime Environment

34.1 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:

```lisp
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.

34.2 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
: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.

34.3 Functions and Variables for Runtime Environment

`feature`  

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 \(\text{featurep}\) returns \(false\) when neither \(f\) nor the negation of \(f\) can be established.
\(\text{featurep}\) evaluates its argument.
See also \(\text{declare}\) and \(\text{features}\).

(%i1) declare (j, even)$
(%i2) featurep (j, integer);
(%o2) true

maxima_tempdir  System variable
\(\text{maxima_tempdir}\) names the directory in which Maxima creates some temporary files.
In particular, temporary files for plotting are created in \(\text{maxima_tempdir}\).
The initial value of \(\text{maxima_tempdir}\) is the user’s home directory, if Maxima can locate it; otherwise Maxima makes a guess about a suitable directory.
\(\text{maxima_tempdir}\) may be assigned a string which names a directory.

maxima_userdir  System variable
\(\text{maxima_userdir}\) names a directory which Maxima searches to find Maxima and Lisp files. (Maxima searches some other directories as well; \(\text{file_search_maxima}\) and \(\text{file_search_lisp}\) are the complete lists.)
The initial value of \(\text{maxima_userdir}\) is a subdirectory of the user’s home directory, if Maxima can locate it; otherwise Maxima makes a guess about a suitable directory.
\(\text{maxima_userdir}\) may be assigned a string which names a directory. However, assigning to \(\text{maxima_userdir}\) does not automatically change \(\text{file_search_maxima}\) and \(\text{file_search_lisp}\); those variables must be changed separately.

room ()  Function
room (true)  Function
room (false)  Function
Prints out a description of the state of storage and stack management in Maxima.
\(\text{room}\) calls the Lisp function of the same name.
• \(\text{room} ()\) prints out a moderate description.
• \(\text{room} (\text{true})\) prints out a verbose description.
• \(\text{room} (\text{false})\) prints out a terse description.

sstatus (keyword, item)  Function
When \(\text{keyword}\) is the symbol \(\text{feature}\), \(\text{item}\) is put on the list of system features.
After \(\text{sstatus (keyword, item)}\) is executed, \(\text{status (feature, item)}\) returns \(true\).
If \(\text{keyword}\) is the symbol \(\text{nofeature}\), \(\text{item}\) 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 \(\text{status}\).
status (feature) Function
status (feature, item) Function

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.

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 **shoutime: true** to make Maxima print out the computation time and elapsed time with each output line.

timedate () Function
timedate (T) Function

timedate() with no argument returns a string representing the current time and date. The string has the format YYYY-MM-DD HH:MM:SS[+|-]ZZ:ZZ, where the fields are year, month, day, hours, minutes, seconds, and time zone offset in hours and minutes.

**timedate(T)** returns the time T as a string with the format YYYY-MM-DD HH:MM:SS[+|-]ZZ:ZZ. T is interpreted as the number of seconds since midnight, January 1, 1900, as returned by **absolute_real_time**.

Example:

timedate with no argument returns a string representing the current time and date.

```
(%i1) d : timedate ();
2010-06-08 04:08:09+01:00
(%o1)
(%i2) print ("timedate reports current time", d) $
"timedate reports current time 2010-06-08 04:08:09+01:00"

(%i1) timedate (0);
1900-01-01 01:00:00+01:00
(%o1)
(%i2) timedate (absolute_real_time () - 7*24*3600);
2010-06-01 04:19:51+01:00
```

```
**absolute_real_time ()**

Returns the number of seconds since midnight, January 1, 1900 UTC. 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 ()**

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 ()**

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
```
35 Miscellaneous Options

35.1 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.

35.2 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.

35.3 Functions and Variables for Miscellaneous Options

aliases

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).

alphabetic

alphabetic is a declaration 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 6.4 [Identifiers], page 67.

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~]

args (expr)

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, and op.
**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 (x)**  
Function  
*gensym()* creates and returns a fresh symbol.  
The name of the new-symbol is the concatenation of a prefix, which defaults to "g", and a suffix, which is the decimal representation of a number that defaults to the value of a Lisp internal counter.  
If *x* is supplied, and is a string, then that string is used as a prefix instead of "g" for this call to *gensym* only.  
If *x* is supplied, and is an integer, then that integer, instead of the value of the internal Lisp integer, is used as the suffix for this call to *gensym* only.  
If and only if no explicit suffix is supplied, the Lisp internal integer is incremented after it is used.  
Examples:  
(\%i1) gensym();  
(\%o1) g887  
(\%i2) gensym("new");  
(\%o2) new888  
(\%i3) gensym(123);  
(\%o3) g123

**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 declared and undeclared arrays, created by ;, ;:, or :=.  
macros All user-defined macro functions.  
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 
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.)

integerp (expr) Function 
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

m1pbranch Option variable
Default value: false
\textbf{m1pbranch} is the principal branch for \(-1\) to a power. Quantities such as \((-1)^{\frac{1}{3}}\) (that is, an "odd" rational exponent) and \((-1)^{\frac{1}{4}}\) (that is, an "even" rational exponent) are handled as follows:

\begin{verbatim}
domain: real
(-1)^{\frac{1}{3}}: -1
(-1)^{\frac{1}{4}}: (-1)^{\frac{1}{4}}

domain: complex
m1pbranch: false  m1pbranch: true
(-1)^{\frac{1}{3}}: 1/2+\%i*sqrt(3)/2
(-1)^{\frac{1}{4}}: sqrt(2)/2+\%i*sqrt(2)/2
\end{verbatim}

\textbf{numberp} \((expr)\)

\textbf{Function}

Returns \textbf{true} if \textit{expr} is a literal integer, rational number, floating point number, or bigfloat, otherwise \textbf{false}.

\textbf{numberp} returns \textbf{false} if its argument is a symbol, even if the argument is a symbolic number such as \%pi or \%i, or declared to be even, odd, integer, rational, irrational, real, imaginary, or complex.

Examples:

\begin{verbatim}
(%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]
\end{verbatim}

\textbf{properties} \((a)\)

\textbf{Function}

Returns a list of the names of all the properties associated with the atom \textit{a}.

\textbf{props} \hspace{0.5cm} \textbf{Special symbol}

\textbf{props} are atoms which have any property other than those explicitly mentioned in \textbf{infolists}, such as specified by \textbf{atvalue}, \textbf{matchdeclare}, etc., as well as properties specified in the \textbf{declare} function.

\textbf{propvars} \((prop)\)

\textbf{Function}

Returns a list of those atoms on the \textbf{props} list which have the property indicated by \textit{prop}. Thus \textbf{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*.

Examples:

```plaintext
(%i1) put (foo, (a+b)^5, expr); 5
   (%o1) (b + a)
(%i2) put (foo, "Hello", str);  Hello
(%i3) properties (foo);
   [%[user properties, str, expr]]
(%i4) get (foo, expr);
   (b + a)
(%i5) get (foo, str);
   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.

Example:

```plaintext
(%i1) foo: aa$
(%i2) bar: bb$
(%i3) baz: cc$
(%i4) put (foo, bar, baz);
   bb
(%i5) properties (aa);
   [%[user properties, cc]]
(%i6) get (aa, cc);
   bb
(%i7) qput (foo, bar, baz);
   bar
(%i8) properties (foo);
   [value, [%[user properties, baz]]]
(%i9) get ('foo, 'baz);
   bar
```

**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 \( a_1, p_1, \ldots, a_n, p_n \)  
Function

\[ \text{remove} \left( \left[ a_1, \ldots, a_m \right], \left[ p_1, \ldots, p_n \right], \ldots \right) \]  
Function

\[ \text{remove} \left( \"a\", \text{operator} \right) \]  
Function

\[ \text{remove} \left( a, \text{transfun} \right) \]  
Function

\[ \text{remove} \left( \text{all}, p \right) \]  
Function

Removes properties associated with atoms.

- remove \( (a_1, p_1, \ldots, a_n, p_n) \) removes property \( p_k \) from atom \( a_k \).
- remove \( ([a_1, \ldots, a_m], [p_1, \ldots, p_n], \ldots) \) removes properties \( p_1, \ldots, p_n \) from atoms \( a_1, \ldots, a_m \). There may be more than one pair of lists.
- remove \( (\text{all}, p) \) removes the property \( p \) from all atoms which have it.

The removed properties may be system-defined properties such as \text{function}, \text{macro}, or \text{mode_declare}. \text{remove} does not remove properties defined by \text{put}.

A property may be \text{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 \( (\"a\", \text{operator}) \) or, equivalently, remove \( (\"a\", \text{op}) \) removes from \( a \) the operator properties declared by \text{prefix}, \text{infix}, \text{nary}, \text{postfix}, \text{matchfix}, or \text{nofix}.
Note that the name of the operator must be written as a quoted string.

- remove always returns \text{done} whether or not an atom has a specified property. This behavior is unlike the more specific remove functions \text{remvalue}, \text{remarray}, \text{remfunction}, and \text{remrule}.

\text{remove} quotes its arguments.

\text{remvalue} \( (\text{name}_1, \ldots, \text{name}_n) \)  
Function

remove \( \text{(all)} \)  
Function

Removes the values of user variables \( \text{name}_1, \ldots, \text{name}_n \) (which can be subscripted) from the system.

- \text{remvalue} \( \text{(all)} \) removes the values of all variables in \text{values}, the list of all variables given names by the user (as opposed to those which are automatically assigned by Maxima).

See also \text{values}.

\text{rncombine} \( \text{(expr)} \)  
Function

Transforms \( \text{expr} \) by combining all terms of \( \text{expr} \) that have identical denominators or denominators that differ from each other by numerical factors only. This is slightly different from the behavior of \text{combine}, which collects terms that have identical denominators.

Setting \text{pfeformat: true} and using \text{combine} yields results similar to those that can be obtained with \text{rncombine}, but \text{rncombine} takes the additional step of cross-multiplying numerical denominator factors. This results in neater forms, and the possibility of recognizing some cancellations.

\text{load(rncomb)} loads this function.
scalarp \( (\text{expr}) \)

Function

Returns \texttt{true} if \texttt{expr} is a number, constant, or variable declared \texttt{scalar} with \texttt{declare}, or composed entirely of numbers, constants, and such variables, but not containing matrices or lists.

setup_autoload \( (\text{filename, function}_1, ..., \text{function}_n) \)

Function

Specifies that if any of \texttt{function}_1, ..., \texttt{function}_n are referenced and not yet defined, \texttt{filename} is loaded via \texttt{load}. \texttt{filename} usually contains definitions for the functions specified, although that is not enforced.

\texttt{setup\_autoload} does not work for array functions.

\texttt{setup\_autoload} quotes its arguments.

Example:

\begin{verbatim}
(%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)
----------- + 3 (%pi - 1) + 1
2
(%o3) 3 (%pi - 1)
2
(%i4) legendre_p (1, %pi);
(%o4) %pi
(%i5) legendre_q (1, %pi);
(%o5) %pi + 1
%pi log(-----)
1 - %pi
----------- - 1
2
\end{verbatim}
36 Rules and Patterns

36.1 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.

36.2 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** (pro_fname, pattern, x1, ..., xn)

Function

Defines a function pro_fname(expr, x1, ..., xn) which tests expr to see if it matches pattern.

`pattern` is an expression containing the pattern arguments x1, ..., xn (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 pro_fname is an expression to be matched against the pattern and the other arguments are the actual arguments which correspond to the dummy variables x1, ..., xn in the pattern.

If the match is successful, pro_fname 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, pro_fname 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.
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`.

Define a function `checklimits(expr)` which tests `expr` to see if it is a definite integral.

```lisp
(%i1) matchdeclare (a, lambda ([e], e#0 and freeof(x, e)), b, freeof(x));
(%o1) done
(%i2) defmatch (linearp, a*x + b, x);
(%o2) linearp
(%i3) linearp (3*z + (y + 1)*z + y^2, z);
   2
(%o3) [b = y, a = y + 4, x = z]
(%i4) a;
(%o4) y + 4
(%i5) b;
(%o5) 2
(%i6) x;
(%o6) x

Define a function `checklimits(expr)` which tests `expr` to see if it is a definite integral.

```
\[
\frac{x + 2 \pi}{\int \sin(t) \, dt}
\]

\[
\frac{x + \pi}{x + 2 \pi}
\]

```
(%i8) I sin(t) dt
(%o8) \[
\]
(%i9) checklimits (%);
(%o9) [b = x + 2 \pi, a = x + \pi, x = t, f = \sin(t)]
```

**defrule** *(rulename, pattern, replacement)*

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** *(rulename_1, ..., rulename_n)*

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]
(%i6) ''%;
(%o6) [foorule1 : foo(x, y) -> baz(y) + bar(x),
   +rule1 : y + x -> special_add(x, y), quux : mumble(x) -> []]
```
Function let

**Function let**

```plaintext
Function 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` go 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`.

```plaintext
(%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 (a!/a, (a-1)!);

   a!
--- --> (a - 1)!
a

(%i6) letsimp (n*m*(n-1)!/m);

   (m - 1)! n!

(%i7) let (sin(a)^2, 1 - cos(a)^2);

   2 2
sin (a) --> 1 - cos (a)

(%i8) letsimp (sin(x)^4);

   4 2
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!
--- --> (n - 1)!
n
(%i3) letrat: false$

(%i4) letsimp (a!/a);

       a!
---
a

(%i5) letrat: true$

(%i6) letsimp (a!/a);

       (a - 1)!

letrules ()

Function

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 (expr)

Function

letsimp (expr, package_name)

Function

letsimp (expr, package_name_1, ..., package_name_n)

Function

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.

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 expres-
sion, 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) de-
pendent 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`.

```maxima
(%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.

```maxima
(%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.

```maxima
(%i1) matchdeclare (aa, atom, bb, lambda ([x], not atom(x)));
(%o1) done
(%i2) defrule (r1, aa + bb, ["all atoms" = aa, "all nonatoms" = bb]);
   bb + aa partitions 'sum'
(%o2) r1 : bb + aa -> [all atoms = aa, all nonatoms = bb]
(%i3) r1 (8 + a*b + sin(x));
(%o3) ...
```
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.

The functions propvars and printprops return information about match variables.

Function \texttt{matchfix} \hspace{1em} (\texttt{ldelimiter}, \texttt{rdelimiter})  \hspace{1em} \textbf{Function}

Declares a matchfix operator with left and right delimiters \texttt{ldelimiter} and \texttt{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 unparsable 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 \texttt{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: \texttt{expr}, \texttt{clause}, and \texttt{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 \texttt{:=} or \texttt{define}. The arguments may be written between the delimiters, or with the left delimiter as a quoted string and the arguments following in parentheses. \texttt{dispfun (ldelimitor)} displays the function definition.

The only built-in matchfix operator is the list constructor \texttt{[ ]}. Parentheses \texttt{( )} and double-quotes \texttt{" "} act like matchfix operators, but are not treated as such by the Maxima parser.

\texttt{matchfix} evaluates its arguments. \texttt{matchfix} returns its first argument, \texttt{ldelimitor}.

Examples:

- Delimiters may be almost any strings.

\begin{verbatim}
(%i1) matchfix ("@@", "---");
@@
(%i2) @@ a, b, c ~;
@@a, b, c~
(%i3) matchfix (">", "<");
>>
(%i4) >> a, b, c <=;
>>a, b, c<<
(%i5) matchfix ("foo", "oof");
foo
(%i6) foo a, b, c oof;
fooa, b, coof
(%i7) >> w + foo x, y oof + z << @ p, q ~;
>>z + foox, yooft + w<<
       ------------------------
       @p, q~
\end{verbatim}
Matchfix operators are ordinary user-defined functions.

```
(%i1) matchfix ("!-", "-!");
(%o1) "!-

(%i2) !- x, y -! := x/y - y/x;
    x y
    !-x, y-! := - -
    y x

(%i3) define (!-x, y-!, x/y - y/x);
    x y
    !-x, y-! := - -
    y x

(%i4) define ("!-" (x, y), x/y - y/x);
    x y
    !-x, y-! := - -
    y x

(%i5) dispfun ("!-");
    x y
    !-x, y-! := - -
    y x

(%o5) done

(%i6) !-3, 5-!;
    16
    --
    15

(%i7) "!-" (3, 5);
    16
    --
    15
```

**remlet** *(prod, name)* Function

**remlet** *(all)* Function

**remlet** *(all, name)* Function

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** *(op, rulename)* Function

**remrule** *(op, all)* Function

Removes rules defined by *tellsimp* or *tellsimpafter*. 
remrule \((op, rule\name)\) removes the rule with the name \(rule\name\) from the operator \(op\). When \(op\) is a built-in or user-defined operator (as defined by \texttt{infix}, \texttt{prefix}, etc.), \(op\) and \(rule\name\) must be enclosed in double quote marks.

remrule \((op, all)\) removes all rules for the operator \(op\).

See also \texttt{remlet}, which removes a rule defined by \texttt{let}.

Examples:

\begin{verbatim}
(\%i1) tellsimp (foo (aa, bb), bb - aa);
(\%o1) [foorule1, false]
(\%i2) tellsimpafter (aa + bb, special_add (aa, bb));
(\%o2) [+rule1, simplplus]
(\%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)];
(\%o7) [bb - aa, special_add(aa, bb), --, %pi - %e, %pi + %e]
(\%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)]
\end{verbatim}

tellsimp \((pattern, replacement)\) Function

is similar to \texttt{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 \texttt{tellsimpafter}.

The pattern may not be a sum, product, single variable, or number.

The system variable \texttt{rules} is the list of rules defined by \texttt{defrule}, \texttt{defmatch}, \texttt{tellsimp}, and \texttt{tellsimpafter}. 
Examples:

```lisp
(%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)
(%i12) expand (%);
2
(%o12) 2 sin(x) - cos (x) + 2
(%i13) sin(x)^2;
2
(%o13) 1 - cos (x)
(%i14) kill (rules);
(%o14) done
(%i15) matchdeclare (a, true);
(%o15) done
(%i16) tellsimp (sin(a)^2, 1 - cos(a)^2);
(%o16) ["rule3, simpexpt]
(%i17) sin(y)^2;
2
(%o17) 1 - cos (y)
```

tellsimpafter (pattern, replacement) 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 $:lisp (\text{trace } \$foorule1)$ traces the function, and $:lisp (\text{symbol-function } '\$foorule1)$ 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));
```
Chapter 36: Rules and Patterns

Chapter 36: Rules and Patterns

(%o2) [sinrule1, simp-%sin]
(%i3) sin ([1/6, 1/4, 1/3, 1/2, 1]*%pi);
   1 sqrt(2) sqrt(3) 2 2 2
(%o3) [-, --------, --------, 1, 0]
   2 2
(%i4) tellsimpafter (11^mm, map ("^", ll, mm));
(%i5) [a, b, c]^[1, 2, 3]; 2 3
   2 3
(%o5) [a, b, c]
(%i6) tellsimpafter (foo (aa (xx)), aa (foo (xx))); (%o6) [foorule1, false]
(%i7) foo (bar (u - v));
   bar(foo(u - v))
(%i8) tellsimpafter (foo (aa), bar_1 (aa));
   [foorule1, false]
(%i9) tellsimpafter (foo (aa), bar_2 (aa));
   [foorule2, foorule1, false]
(%i10) foo (42);
   bar_1(42)
(%o10) done
(%i11) matchdeclare (aa, integerp);
(%o11) done
(%i12) tellsimpafter (foo (aa), bar_1 (aa));
   [foorule1, false]
(%i13) tellsimpafter (foo (aa), bar_2 (aa));
   [foorule2, foorule1, false]
(%i14) foo (42);
   bar_1(42)
(%i15) tellsimpafter (foo (aa, bb), bar('aa=aa, 'bb=bb));
   [foorule1, false]
(%i16) tellsimpafter (foo (aa, bb), bar (u - v));
   bar(foo(u - v))
(%o16) done
(%o17) tellsimpafter (foo (aa), bar_1 (aa));
   [foorule1, false]
(%i18) tellsimpafter (foo (aa), bar_2 (aa));
   [foorule2, foorule1, false]
(%i19) foo (42);
   bar_1(42)
(%o19) done
(%i20) matchdeclare (aa, integerp, bb, atom);
(%o20) done
(%i21) tellsimpafter (foo (aa, bb), bar ('aa=aa, 'bb=bb));
   [foorule1, false]
(%i22) tellsimpafter (foo (aa, bb), bar (u - v));
   bar(foo(u - v))
(%o22) done
(%i23) foo (42, %e);
   bar(aa = 42, bb = %e)
(%o23) done
(%i24) bar(aa = 42, bb = %e)
(%o24) done
(%i25) foo (42, %e);
   bar(aa = 42, bb = %e)
(%o25) done
(%i26) define_variable (bb, true, boolean);
   true
(%o26) done
(%i27) foo (42, %e);
   bar(aa = 42, bb = %e)
Error: bb was declared mode boolean, has value: %e
-- an error. Quitting. 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);
(%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 ()

Executes kill (rules) and then resets the next rule number to 1 for addition +, multiplication *, and exponentiation \(^\)
37 Lists

37.1 Introduction to Lists

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

\[(\text{MPLUS} \ $A \ 2)\]
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

\[(\text{MLIST} \ 1 \ 2 \ 7 \ (\text{MPLUS} \ x \ y))\]
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

\[(\text{MLIST SIMP} \ 1 \ 2 \ 7 \ (\text{MPLUS SIMP} \ x \ y))\]

37.2 Functions and Variables for Lists

**append** \((\text{list}_1, \ldots, \text{list}_n)\)

Function

Returns a single list of the elements of \(\text{list}_1\) followed by the elements of \(\text{list}_2, \ldots\)

append also works on general expressions, e.g. append \((f(a,b), f(c,d,e))\); yields \(f(a,b,c,d,e)\).

Do example(append); for an example.

**assoc** \((\text{key}, \text{list}, \text{default})\)

Function

This function searches for the \text{key} in the left hand side of the input \text{list} of the form \([x,y,z,\ldots]\) where each of the \text{list} elements is an expression of a binary operand and 2 elements. For example \(x=1, 2^3, [a,b]\) etc. The \text{key} is checked against the first operand. assoc returns the second operand if the key is found. If the key is not found it either returns the default value. default is optional and defaults to false.

**atom** \(\text{expr}\)

Function

Returns true if expr is atomic (i.e. a number, name or string) else false. Thus atom(5) is true while atom(a[1]) and atom(sin(x)) are false (assuming a[1] and x are unbound).

**cons** \((\text{expr}, \text{list})\)

Function

Returns a new list constructed of the element expr as its first element, followed by the elements of list. cons also works on other expressions, e.g. cons(x, f(a,b,c)); -> f(x,a,b,c).

**copylist** \(\text{list}\)

Function

Returns a copy of the list list.
create_list \(\text{form, } x_1, \text{list}_1, \ldots, x_n, \text{list}_n\) \hspace{1cm} \text{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\(_2\), \ldots. 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.

\[
\begin{align*}
(\%i1) & \quad \text{create\_list}(x^i, i, [1, 3, 7]) ; \\
(\%o1) & \quad [x, x^3, x^7]
\end{align*}
\]

With a double iteration:

\[
\begin{align*}
(\%i1) & \quad \text{create\_list}([i, j], i, [a, b], j, [e, f, h]) ; \\
(\%o1) & \quad [[a, e], [a, f], [a, h], [b, e], [b, f], [b, h]]
\end{align*}
\]

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.

\[
\begin{align*}
(\%i1) & \quad \text{create\_list}([i, j], i, [1, 2, 3], j, 1, i) ; \\
(\%o1) & \quad [[1, 1], [2, 1], [2, 2], [3, 1], [3, 2], [3, 3]]
\end{align*}
\]

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

delete \((expr_1, expr_2)\) \hspace{1cm} \text{Function}

delete \((expr_1, expr_2, n)\) \hspace{1cm} \text{Function}

\text{delete}(expr_1, expr_2) \text{ removes from expr}_2 \text{ any arguments of its top-level operator which are the same (as determined by "=") as expr}_1 \text{. Note that "}=\" tests for formal equality, not equivalence. Note also that arguments of subexpressions are not affected. expr}_1 \text{ may be an atom or a non-atomic expression. expr}_2 \text{ may be any non-atomic expression. delete returns a new expression; it does not modify expr}_2. \text{ delete}(expr_1, expr_2, n) \text{ removes from expr}_2 \text{ the first } n \text{ arguments of the top-level operator which are the same as expr}_1. \text{ If there are fewer than } n \text{ such arguments, then all such arguments are removed.}

Examples:

Removing elements from a list.

\[
\begin{align*}
(\%i1) & \quad \text{delete} (y, [w, x, y, z, z, y, x, w]) ; \\
(\%o1) & \quad [w, x, z, z, x, w]
\end{align*}
\]

Removing terms from a sum.

\[
\begin{align*}
(\%i1) & \quad \text{delete} (\sin(x), x + \sin(x) + y) ; \\
(\%o1) & \quad y + x
\end{align*}
\]

Removing factors from a product.

\[
\begin{align*}
(\%i1) & \quad \text{delete} (u - x, (u - w)*(u - x)*(u - y)*(u - z)) ; \\
(\%o1) & \quad (u - w) (u - y) (u - z)
\end{align*}
\]

Removing arguments from an arbitrary expression.

\[
\begin{align*}
(\%i1) & \quad \text{delete} (a, \text{foo} (a, b, c, d, a)) ; \\
(\%o1) & \quad \text{foo}(b, c, d)
\end{align*}
\]

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 \texttt{expr} is determined by "=". Arguments which are \texttt{equal} but not "=" are not removed.

(%i1) \texttt{[is(equal(0, 0)), is(equal(0, 0.0)), is(equal(0, 0b0))];}
\texttt{`rat' replaced 0.0 by 0/1 = 0.0}
\texttt{`rat' replaced 0.0B0 by 0/1 = 0.0B0}
(%o1) \texttt{[true, true, true]}
(%i2) \texttt{[is (0 = 0), is (0 = 0.0), is (0 = 0b0)];}
(%o2) \texttt{[true, false, false]}
(%i3) delete (0, \texttt{[0, 0.0, 0b0]});
(%o3) \texttt{[0.0, 0.0b0]}
(%i4) is (equal ((x + y)*(x - y), x^2 - y^2));
(%o4) \texttt{true}
(%i5) is ((x + y)*(x - y) = x^2 - y^2);
(%o5) \texttt{false}
(%i6) delete ((x + y)*(x - y), \texttt{[(x + y)*(x - y), x^2 - y^2]});
\begin{align*}
\texttt{2} & \\texttt{2} \\
\texttt{x - y}
\end{align*}
eighth (\texttt{expr}) Function
Returns the 8'th item of expression or list \texttt{expr}. See \texttt{first} for more details.

endcons (\texttt{expr, list}) Function
Returns a new list consisting of the elements of \texttt{list} followed by \texttt{expr}. \texttt{endcons} also works on general expressions, e.g. \texttt{endcons(x, f(a,b,c)); \rightarrow f(a,b,c,x)}.

fifth (\texttt{expr}) Function
Returns the 5'th item of expression or list \texttt{expr}. See \texttt{first} for more details.

first (\texttt{expr}) Function
Returns the first part of \texttt{expr} which may result in the first element of a list, the first row of a matrix, the first term of a sum, etc. Note that \texttt{first} and its related functions, \texttt{rest} and \texttt{last}, work on the form of \texttt{expr} which is displayed not the form which is typed on input. If the variable \texttt{inflag} is set to \texttt{true} however, these functions will look at the internal form of \texttt{expr}. Note that the simplifier re-orders expressions. Thus \texttt{first(x+y)} will be \texttt{x} if \texttt{inflag} is \texttt{true} and \texttt{y} if \texttt{inflag} is \texttt{false} (\texttt{first(y+x)} gives the same results). The functions \texttt{second..tenth} yield the second through the tenth part of their input argument.

fourth (\texttt{expr}) Function
Returns the 4'th item of expression or list \texttt{expr}. See \texttt{first} for more details.

get (\texttt{a, i}) Function
Retrieves the user property indicated by \texttt{i} associated with atom \texttt{a} or returns \texttt{false} if \texttt{a} doesn’t have property \texttt{i}.
\texttt{get} evaluates its arguments.
(%i1) put (%e, 'transcendental, 'type);
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]]

join (l, m)
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.
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)
Returns the last part (term, row, element, etc.) of the expr.

length (expr)
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).

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.

**makelist** (expr, i, i_0, i_1)  
Function  
Constructs and returns a list, each element of which is generated from expr. makelist (expr, i, i_0, i_1) returns a list, the j’th element of which is equal to ev (expr, i=j) for j equal to i_0 through i_1.

**makelist** (expr, x, list)  
Function  
Constructs and returns a list, each element of which is generated from expr. makelist (expr, x=list[j]) for j equal to 1 through length (list).

Examples:

(%i1) makelist(concat(x,i),i,1,6);  
(%o1) [x1, x2, x3, x4, x5, x6]

(%i2) makelist(x=y,y,[a,b,c]);  
(%o2) [x = a, x = b, x = c]

**member** (expr_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);  
1 1 1
(%o6) F(1, -, -, -)  
2 4 8

(%i7) member (1/8, %);  
(%o7) true

(%i8) member ("ab", ["aa", "ab", sin(1), a + b]);  
(%o8) true
ninth (expr)  
Function  
Returns the 9'th item of expression or list expr. See first for more details.

pop (list)  
Function  
pop removes the first element from the list list and returns this element. list must be a symbol, which is bound to a list and not the list itself.  
If the argument list is not bound to a list or the list is empty, Maxima generates an error message.  
See also push for examples.  
load(basic) loads this function.

push (item, list)  
Function  
push prepends the item item to the list list and returns a copy of the new list. list must be a symbol, which is bound to a list and not the list itself. item can be any Maxima symbol or expression.  
If the argument list is not bound to a list, Maxima generates an error message.  
See also pop to remove the first item from a list.  
load(basic) loads this function.

Examples:

\begin{verbatim}
(%i1) ll:[];
(%o1) []
(%i2) push(x,ll);
(%o2) [x]
(%i3) push(x^2+y,ll);
   2
(%o3) [y + x , x]
(%i4) a:push("string",ll);
   2
(%o4) [string, y + x , x]
(%i5) pop(ll);
(%o5) string
(%i6) pop(ll);
   2
(%o6) y + x
(%i7) pop(ll);
(%o7) x
(%i8) ll;
(%o8) []
(%i9) a;
   2
(%o9) [string, y + x , x]
\end{verbatim}

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:

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

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. `expr` may be a list, matrix, or other expression.

Function `reverse (list)`

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`.

Function `second (expr)`

Returns the 2'nd item of expression or list `expr`. See `first` for more details.

Function `seventh (expr)`

Returns the 7'th item of expression or list `expr`. See `first` for more details.

Function `sixth (expr)`

Returns the 6'th item of expression or list `expr`. See `first` for more details.

Function `sublist_indices (L, P)`

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:

```lisp
(%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]
```

Function `tenth (expr)`

Returns the 10'th item of expression or list `expr`. See `first` for more details.
third (expr)
Returns the 3'rd item of expression or list expr. See first for more details.
38 Sets

38.1 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.

38.1.1 Usage

To construct a set with members $a_1, \ldots, a_n$, write \texttt{set($a_1, \ldots, a_n$)} or \{ $a_1, \ldots, a_n$ \}; to construct the empty set, write \texttt{set()} or \{}. In input, \texttt{set(...)} and \{ ... \} are equivalent. Sets are always displayed with curly braces.

If a member is listed more than once, simplification eliminates the redundant member.

\begin{verbatim}
(%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]}
\end{verbatim}

Two would-be elements $x$ and $y$ are redundant (i.e., considered the same for the purpose of set construction) if and only if \texttt{is($x = y$)} yields \texttt{true}. Note that \texttt{is(equal($x$, $y$))} can yield \texttt{true} while \texttt{is($x = y$)} yields \texttt{false}; in that case the elements $x$ and $y$ are considered distinct.

\begin{verbatim}
(%i1) x: a/c + b/c;
     b a
(%o1) - + -
     c c
(%i2) y: a/c + b/c;
     b a
(%o2) - + -
     c c
\end{verbatim}
(\%i3) \( z: (a + b)/c; \)
\begin{align*}
& b + a \\
& ------ \\
& c
\end{align*}

(\%o3) \( \frac{b + a}{c} \)

(\%i4) \( \text{is (x = y)}; \)
(\%o4) \( \text{true} \)

(\%i5) \( \text{is (y = z)}; \)
(\%o5) \( \text{false} \)

(\%i6) \( \text{is (equal (y, z))}; \)
(\%o6) \( \text{true} \)

(\%i7) \( y - z; \)
\begin{align*}
& b + a \\
& - ----- + - + - \\
& c c c
\end{align*}

(\%o7) \( \frac{b + a}{c} - \frac{b}{c} + \frac{a}{c} \)

(\%i8) \( \text{ratsimp (\%)}; \)
(\%o8) \( 0 \)

(\%i9) \( \{x, y, z\}; \)
\begin{align*}
& b + a \\
& \{-----, - + -\} \\
& c c c
\end{align*}

(\%o9) \( \{\frac{b + a}{c}, - + -\} \)

To construct a set from the elements of a list, use \text{setify}.

(\%i1) \( \text{setify ([b, a])}; \)
(\%o1) \( \{a, b\} \)

Set members \( x \) and \( y \) are equal provided \( \text{is (x = y)} \) evaluates to \text{true}. Thus \( \text{rat (x)} \) and \( x \) are equal as set members; consequently,

(\%i1) \( \{x, \text{rat(x)}\}; \)
(\%o1) \( \{x\} \)

Further, since \( \text{is ((x - 1)*(x + 1) = x^2 - 1)} \) evaluates to \text{false}, \( (x - 1)*(x + 1) \) and \( x^2 - 1 \) are distinct set members; thus

(\%i1) \( \{(x - 1)*(x + 1), x^2 - 1\}; \)
(\%o1) \( \{(x - 1) (x + 1), x - 1\} \)

To reduce this set to a singleton set, apply \text{rat} to each set member:

(\%i1) \( \{(x - 1)*(x + 1), x^2 - 1\}; \)
(\%o1) \( \{(x - 1) (x + 1), x - 1\} \)

(\%i2) \( \text{map (rat, \%)}; \)
(\%o2)/\%\/
\[
\begin{align*}
& 2 \\
& \{x - 1\}
\end{align*}
\]

To remove redundancies from other sets, you may need to use other simplification functions. Here is an example that uses \text{trigsimp}:

(\%i1) \( \{1, \cos(x)^2 + \sin(x)^2\}; \)
(\%o1) \( \{1, \sin(x) + \cos(x)\} \)

(\%i2) \( \text{map (trigsimp, \%)}; \)
(\%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,

```plaintext
(%i1) s: {a, b, c}$
(%i2) subst (c=a, s);
   {a, b}
(%o2)

(%i3) subst ([a=x, b=x, c=x], s);
   {x}
(%o3)

(%i4) map (lambda ([x], x^2), set (-1, 0, 1));
   {0, 1}
(%o4)
```

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

```plaintext
(%i1) union ([1, 2], {a, b});
   --- an error. Quiting. To debug this try debugmode(true);
(%i2) union (setify ([1, 2]), {a, b});
   {1, 2, a, b}
(%o2)
```

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

```plaintext
(%i1) subset ({x + y + z, x - y + 4, x + y - 5},
   lambda ([e], freeof (z, e)));
   {- y + x + 4, y + x - 5}
(%o1)
```

The section Section 38.2 [Functions and Variables for Sets], page 505 has a complete list of the set functions in Maxima.

### 38.1.2 Set Member Iteration

There two ways to to iterate over set members. One way is the use `map`; for example:

```plaintext
(%i1) map (f, {a, b, c});
   {f(a), f(b), f(c)}
(%o1)
```

The other way is to use `for x in s do`

```plaintext
(%i1) s: {a, b, c};
   {a, b, c}
(%o1)

(%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.
38.1.3 Bugs

The set functions use the Maxima function \texttt{orderlessp} to order set members and the (Lisp-level) function \texttt{like} to test for set member equality. Both of these functions have known bugs that may manifest if you attempt to use sets with members that are lists or matrices that contain expressions in canonical rational expression (CRE) form. An example is

\begin{verbatim}
(%i1) [{x}, [rat (x)]];
Maxima encountered a Lisp error:

The value #:X1440 is not of type LIST.

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
\end{verbatim}

This expression causes Maxima to halt with an error (the error message depends on which version of Lisp your Maxima uses). Another example is

\begin{verbatim}
(%i1) setify ([[rat(a)], [rat(b)]]);
Maxima encountered a Lisp error:

The value #:A1440 is not of type LIST.

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
\end{verbatim}

These bugs are caused by bugs in \texttt{orderlessp} and \texttt{like}; they are not caused by bugs in the set functions. To illustrate, try the expressions

\begin{verbatim}
(%i1) orderlessp ([rat(a)], [rat(b)]);
Maxima encountered a Lisp error:

The value #:B1441 is not of type LIST.

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
\end{verbatim}

Until these bugs are fixed, do not construct sets with members that are lists or matrices containing expressions in CRE form; a set with a member in CRE form, however, shouldn’t be a problem:

\begin{verbatim}
(%i1) {x, rat (x)};
(%o1)                  \{x\}
\end{verbatim}

Maxima’s \texttt{orderlessp} has another bug that can cause problems with set functions, namely that the ordering predicate \texttt{orderlessp} is not transitive. The simplest known example that shows this is

\begin{verbatim}
(%i1) q: x^2$
(%i2) r: (x + 1)^2$
(%i3) s: x*(x + 2)$
(%i4) orderlessp (q, r);
(%o4)                    \text{true}
\end{verbatim}
This bug can cause trouble with all set functions as well as with Maxima functions in general. It is probable, but not certain, that this bug can be avoided if all set members are either in CRE form or have been simplified using \texttt{ratsimp}.

Maxima’s \texttt{orderless} and \texttt{ordergreat} mechanisms are incompatible with the set functions. If you need to use either \texttt{orderless} or \texttt{ordergreat}, call those functions before constructing any sets, and do not call \texttt{unorder}.

If you find something that you think might be a set function bug, please report it to the Maxima bug database. See \texttt{bug_report}.

38.1.4 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.

38.2 Functions and Variables for Sets

\begin{itemize}
  \item \textbf{adjoin (}x, a\textbf{)}
    \begin{itemize}
      \item Returns the union of the set a with \{x\}.
      \item \texttt{adjoin} complains if \texttt{a} is not a literal set.
      \item \texttt{adjoin(x, a)} and \texttt{union(set(x), a)} are equivalent; however, \texttt{adjoin} may be somewhat faster than \texttt{union}.
      \item See also \texttt{disjoin}.
    \end{itemize}

\begin{verbatim}
(%i11) adjoin (c, {a, b});
(%o11) {a, b, c}
(%i12) adjoin (a, {a, b});
(%o12) {a, b}
\end{verbatim}

\item \textbf{belln (}n\textbf{)}
  \begin{itemize}
    \item Represents the \textit{n}-th Bell number. \texttt{belln(n)} is the number of partitions of a set with \textit{n} members.
    \item For nonnegative integers \textit{n}, \texttt{belln(n)} simplifies to the \textit{n}-th Bell number. \texttt{belln} does not simplify for any other arguments.
    \item \texttt{belln} distributes over equations, lists, matrices, and sets.
  \end{itemize}

\begin{verbatim}
(%i11) makelist (belln (i), i, 0, 6);
(%o11) [1, 1, 2, 5, 15, 52, 203]
(%i12) is (cardinality (set_partitions ({})) = belln (0));
\end{verbatim}
\end{itemize}
(%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_1, ... , b_n)  
Function
Returns a set of lists of the form [x_1, ..., x_n], where x_1, ..., x_n are elements of
the sets b_1, ..., b_n, respectively.
cartesian_product complains if any argument is not a literal set.
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]}

disjoin  (x, a)  
Function
Returns the set a without the member x. If x is not a member of a, return a un-
changed.
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:
\begin{verbatim}
(%i1) disjoin (a, {a, b, c, d});
(%o1)       \{b, c, d\}
(%i2) disjoin (a + b, {5, \%pi, a + b});
(%o2)       \{5, \%pi, z\}
(%i3) disjoin (a - b, {5, \%pi, a + b});
(%o3)       \{5, \%pi, b + a, z\}
\end{verbatim}

**disjointp \(a, b\)**

Returns \texttt{true} if and only if the sets \(a\) and \(b\) are disjoint. \texttt{disjointp} complains if either \(a\) or \(b\) is not a literal set.

Examples:
\begin{verbatim}
(%i1) disjointp ({a, b, c}, {1, 2, 3});
(%o1) true
(%i2) disjointp ({a, b, 3}, {1, 2, 3});
(%o2) false
\end{verbatim}

**divisors \(n\)**

Represents the set of divisors of \(n\).

\texttt{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.

\texttt{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.
\begin{verbatim}
(%i1) s: divisors(28);
(%o1) {1, 2, 4, 7, 14, 28}
(%i2) lreduce (+, args(s)) - 28;
(%o2) 28
\end{verbatim}

\texttt{divisors} is a simplifying function. Substituting 8 for \(a\) in \texttt{divisors(a)} yields the divisors without reevaluating \texttt{divisors(8)}.
\begin{verbatim}
(%i1) divisors (a);
(%o1) divisors(a)
(%i2) subst (8, a, %);
(%o2) \{1, 2, 4, 8\}
\end{verbatim}

\texttt{divisors} distributes over equations, lists, matrices, and sets.
\begin{verbatim}
(%i1) divisors (a = b);
(%o1) divisors(a) = divisors(b)
(%i2) divisors ([a, b, c]);
(%o2) [divisors(a), divisors(b), divisors(c)]
(%i3) divisors (matrix ([a, b], [c, d]));
    \[ divisors(a) \ divisors(b) \]
    \[ divisors(c) \ divisors(d) \]
(%i4) divisors ({a, b, c});
(%o4) \{divisors(a), divisors(b), divisors(c)}
\end{verbatim}
Function \texttt{elementp} \((x, a)\)

Returns \texttt{true} if and only if \(x\) is a member of the set \(a\).
\texttt{elementp} complains if \(a\) is not a literal set.

Examples:

\begin{verbatim}
(%i1) elementp (sin(1), {sin(1), sin(2), sin(3)}); 
(%o1) true
(%i2) elementp (sin(1), {cos(1), cos(2), cos(3)}); 
(%o2) false
\end{verbatim}

Function \texttt{emptyp} \((a)\)

Returns \texttt{true} if and only if \(a\) is the empty set or the empty list.

Examples:

\begin{verbatim}
(%i1) map (emptyp, [{}, []]); 
(%o1) [true, true]
(%i2) map (emptyp, [a + b, {}, %pi]); 
(%o2) [false, false, false]
\end{verbatim}

Function \texttt{equiv_classes} \((s, F)\)

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 \texttt{true} or \texttt{false}, or an expression \texttt{expr} such that \texttt{is(expr)} is either \texttt{true} or \texttt{false}.

When \(F\) is not an equivalence relation, \texttt{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 \texttt{true} or \texttt{false}.

\begin{verbatim}
(%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}}
\end{verbatim}

The equivalence relation is the name of a relational function which \texttt{is} evaluates to \texttt{true} or \texttt{false}.

\begin{verbatim}
(%i1) equiv_classes ({1, 1.0, 2, 2.0, 3, 3.0}, equal); 
(%o1) {{1, 1.0}, {2, 2.0}, {3, 3.0}}
\end{verbatim}

The equivalence classes are numbers which differ by a multiple of 3.

\begin{verbatim}
(%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}}
\end{verbatim}

Function \texttt{every} \((f, s)\)

Returns \texttt{true} if the predicate \(f\) is \texttt{true} for all given arguments.

Function \texttt{every} \((f, L_1, ..., L_n)\)

Returns \texttt{true} if the predicate \(f\) is \texttt{true} for all given arguments.
Given one set as the second argument, every\((f, s)\) returns \texttt{true} if \(\text{is}(f(a_i))\) returns \texttt{true} for all \(a_i\) in \(s\). \texttt{every} may or may not evaluate \(f\) for all \(a_i\) in \(s\). Since sets are unordered, \texttt{every} may evaluate \(f(a_i)\) in any order.

Given one or more lists as arguments, every\((f, L_1, \ldots, L_n)\) returns \texttt{true} if \(\text{is}(f(x_1, \ldots, x_n))\) returns \texttt{true} for all \(x_1, \ldots, x_n\) in \(L_1, \ldots, L_n\), respectively. every may or may not evaluate \(f\) for every combination \(x_1, \ldots, x_n\). every evaluates lists in the order of increasing index.

Given an empty set \(\{\}\) or empty lists \([]\) as arguments, every returns \texttt{false}.

When the global flag \texttt{maperror} is \texttt{true}, all lists \(L_1, \ldots, L_n\) must have equal lengths. When \texttt{maperror} is \texttt{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 \texttt{true} or \texttt{false} are governed by the global flag \texttt{prederror}. When \texttt{prederror} is \texttt{true}, such values are treated as \texttt{false}, and the return value from every is \texttt{false}. When \texttt{prederror} is \texttt{false}, such values are treated as \texttt{unknown}, and the return value from every is \texttt{unknown}.

Examples:

\texttt{every} applied to a single set. The predicate is a function of one argument.

\begin{verbatim}
(%i1) every (integerp, {1, 2, 3, 4, 5, 6});
(%o1) true
(%i2) every (atom, {1, 2, sin(3), 4, 5 + y, 6});
(%o2) false
\end{verbatim}

\texttt{every} applied to two lists. The predicate is a function of two arguments.

\begin{verbatim}
(%i1) every ("=", [a, b, c], [a, b, c]);
(%o1) true
(%i2) every ("#", [a, b, c], [a, b, c]);
(%o2) false
\end{verbatim}

Return values of the predicate \(f\) which evaluate to something other than \texttt{true} or \texttt{false} are governed by the global flag \texttt{prederror}.

\begin{verbatim}
(%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
\end{verbatim}

\texttt{extremal_subset} \((s, f, \text{max})\) Function
\texttt{extremal_subset} \((s, f, \text{min})\) Function

Returns the subset of \(s\) for which the function \(f\) takes on maximum or minimum values.
extremal_subset($s, f, \text{max}$) returns the subset of the set or list $s$ for which the real-valued function $f$ takes on its maximum value.

extremal_subset($s, f, \text{min}$) 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)  
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)
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 (n)**

Function

```
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 \[a_1, ..., a_m\] is a partition of a nonnegative integer n when (1) each \(a_i\) is a nonzero integer, and (2) \(a_1 + ... + a_m = n\). Thus 0 has no partitions.

Examples:

\begin{verbatim}
(%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]\}
\end{verbatim}

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.

\begin{verbatim}
(%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, S_3, S_4);
\end{verbatim}

interact \(a_1, ..., a_n\)

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:

\begin{verbatim}
(%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);
\begin{verbatim}
(%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)  \{
\end{verbatim}

\textbf{Function}  \texttt{kron}\_\texttt{delta}  \((x, y)\)

Represents the Kronecker delta function.

\texttt{kron}\_\texttt{delta} simplifies to 1 when \(x\) and \(y\) are identical or demonstrably equivalent, and it simplifies to 0 when \(x\) and \(y\) are demonstrably not equivalent. Otherwise, it is not certain whether \(x\) and \(y\) are equivalent, and \texttt{kron}\_\texttt{delta} simplifies to a noun expression. \texttt{kron}\_\texttt{delta} implements a cautious policy with respect to floating point expressions: if the difference \(x - y\) is a floating point number, \texttt{kron}\_\texttt{delta} simplifies to a noun expression when \(x\) is apparently equivalent to \(y\).

Specifically, \texttt{kron}\_\texttt{delta}(x, y) simplifies to 1 when \texttt{is}(x = y) is true. \texttt{kron}\_\texttt{delta} also simplifies to 1 when \texttt{sign(abs(x - y))} is zero and \(x - y\) is not a floating point number (neither an ordinary float nor a bigfloat). \texttt{kron}\_\texttt{delta} simplifies to 0 when \texttt{sign(abs(x - y))} is pos.

Otherwise, \texttt{sign(abs(x - y))} is something other than pos or zero, or it is zero and \(x - y\) is a floating point number. In these cases, \texttt{kron}\_\texttt{delta} returns a noun expression.

\texttt{kron}\_\texttt{delta} is declared to be symmetric. That is, \texttt{kron}\_\texttt{delta}(x, y) is equal to \texttt{kron}\_\texttt{delta}(y, x).

Examples:

The arguments of \texttt{kron}\_\texttt{delta} are identical. \texttt{kron}\_\texttt{delta} simplifies to 1.

\begin{verbatim}
(%i11) kron\_delta (a, a);
(%o11) 1
(%i12) kron\_delta (x^2 - y^2, x^2 - y^2);
(%o12) 1
(%i13) float (kron\_delta (1/10, 0.1));
(%o13) 1
\end{verbatim}

The arguments of \texttt{kron}\_\texttt{delta} are equivalent, and their difference is not a floating point number. \texttt{kron}\_\texttt{delta} simplifies to 1.

\begin{verbatim}
(%i1) assume (equal (x, y));
(%o1) [equal(x, y)]
(%i2) kron\_delta (x, y);
(%o2) 1
\end{verbatim}

The arguments of \texttt{kron}\_\texttt{delta} are not equivalent. \texttt{kron}\_\texttt{delta} simplifies to 0.

\begin{verbatim}
(%i1) kron\_delta (a + 1, a);
(%o1) 0
(%i2) assume (a > b)$
(%i3) kron\_delta (a, b);
(%o3) 0
\end{verbatim}
The arguments of `kron_delta` might or might not be equivalent. `kron_delta` simplifies to a noun expression.

```lisp
(%i11) kron_delta (a, b);
(%o11) kron_delta(a, b)
(%i12) assume(x >= y)$
(%i13) kron_delta (x, y);
(%o13) kron_delta(x, y)
```

The arguments of `kron_delta` are equivalent, but their difference is a floating point number. `kron_delta` simplifies to a noun expression.

```lisp
(%i11) 1/4 - 0.25;
(%o11) 0.0
(%i12) 1/10 - 0.1;
(%o12) 0.0
(%i13) 0.25 - 0.25b0;
Warning: Float to bigfloat conversion of 0.25
(%o13) 0.0b0
(%i14) kron_delta (1/4, 0.25);
(%o14) kron_delta(-, 0.25)
(%o4) kron_delta(, 0.25)
(%o5) kron_delta(, 0.1)
(%o6) kron_delta(0.25, 2.5b-1)
```

`kron_delta` is symmetric.

```lisp
(%i1) kron_delta (x, y);
(%o1) kron_delta(x, y)
(%i2) kron_delta (y, x);
(%o2) kron_delta(x, y)
(%i3) kron_delta (x, y) - kron_delta (y, x);
(%o3) 0
(%i4) is (equal (kron_delta (x, y), kron_delta (y, x)));
(%o4) true
(%i5) is (kron_delta (x, y) = kron_delta (y, x));
(%o5) true
```

### Function listify (a)
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:**
\begin{alltt}
(\%i1) listify \{a, b, c, d\};
(\%o1) [a, b, c, d]
(\%i2) listify (F \{a, b, c, d\});
(\%o2) F\{a, b, c, d\}
\end{alltt}

**lreduce** \((F, s)\)  
Function

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

The function \(F\) is first applied to the leftmost list elements, thus the name "lreduce".

See also \texttt{rreduce}, \texttt{xreduce}, and \texttt{tree_reduce}.

**Examples:**

\begin{alltt}
(\%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)
\end{alltt}

\begin{alltt}
(\%i1) lreduce (f, [1, 2, 3], 4);
(\%o1) f(f(f(4, 1), 2), 3)
\end{alltt}

\begin{alltt}
(\%i1) lreduce (\(^-\), args \{a, b, c, d\});
(\%o1) ((a ))
(\%i2) lreduce (\(/\), args \{a, b, c, d\});
(\%o2) ----
\end{alltt}

**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 \texttt{makelist}.

**Examples:**

\begin{alltt}
(\%i1) makeset (i/j, [i, j], [[1, a], [2, b], [3, c], [4, d]]);
(\%o1) \{'{-, -, -, -}\}
(\%i2) S : \{x, y, z}\$
\end{alltt}
(%i3) S3 : cartesian_product ([x, x, x], [x, x, y], [x, x, z], [x, y, x], [x, y, y], [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]);
(%o3) {
  [x, x, x], [x, x, y], [x, x, z], [x, y, x], [x, y, y], [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]]));
    [ - 1 0 ]
    [    ]
    [ - 1 1 ]
(%i9) moebius ({[21, 22, 23, 24]});
(%o9) {- 1, 0, 1}

multinomial_coeff (a_1, ..., a_n)

Function

Returns the multinomial coefficient.

When each \( a_k \) is a nonnegative integer, the multinomial coefficient gives the number of ways of placing \( a_1 + \ldots + a_n \) distinct objects into \( n \) boxes with \( a_k \) elements in the \( k \)'th box. In general, \( \text{multinomial_coeff}(a_1, \ldots, a_n) \) evaluates to \((a_1 + \ldots + a_n)!/(a_1! \ldots a_n!)\).
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)!
(x + 3) ---------
   2 x!
(%o1)--------------------------------------------------

(%i2) minfactorial (%);
   (x + 1) (x + 2) (x + 3)
(x + 1) (x + 2) (x + 3) -----------------------
   2
(%o2)--------------------------------------------------

(%i3) multinomial_coeff (-6, 2);
   (- 4)!
(- 4) !
(%o3)--------------------------------------------------

(%i4) minfactorial (%);
   2 (- 6)!
(- 6) !
(%o4) 10
```

**num_distinct_partitions**

Function

**num_distinct_partitions** \((n)\)

**num_distinct_partitions** \((n, \text{list})\)

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** \((n, \text{list})\) 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 \(k_1, ..., k_m\) such that \(n = k_1 + ... + k_m\).

Examples:

```
(%i1) num_distinct_partitions (12);
   15
(%o1) 15

(%i2) num_distinct_partitions (12, list);
   [1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15]
(%o2) [1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15]

(%i3) num_distinct_partitions (n);
(%o3) num_distinct_partitions(n)
```

**num_partitions**

Function

**num_partitions** \((n)\)

**num_partitions** \((n, \text{list})\)

Returns the number of integer partitions of \(n\) when \(n\) is a nonnegative integer. Otherwise, **num_partitions** returns a noun expression.

**num_partitions** \((n, \text{list})\) returns a list of the number of integer partitions of 1, 2, 3, ..., \(n\).

For a nonnegative integer \(n\), **num_partitions** \((n)\) is equal to \(\text{cardinality(integer_partitions(n))}\); however, **num_partitions** does not actually construct the set of partitions, so it is much faster.

Examples:
(%i1) num_partitions (5) = cardinality (integer_partitions (5));
(%o1)                 7 = 7
(%i2) num_partitions (8, list);
(%o2)     [1, 1, 2, 3, 5, 7, 11, 15, 22]
(%i3) num_partitions (n);
(%o3)             num_partitions(n)

partition_set (a, f)
function
Partitions the set a according to the predicate f.
partition_set returns a list of two sets. The first set comprises the elements of a
for which f evaluates to false, and the second comprises any other elements of a.
partition_set does not apply is to the return value of f.
partition_set complains if a is not a literal set.
See also subset.
Examples:

(%i1) partition_set ({2, 7, 1, 8, 2, 8}, evenp);
(%o1)        [{1, 7}, {2, 8}]
(%i2) partition_set ({x, rat(y), rat(y) + z, 1},
                   lambda ([x], ratp(x)));
(%o2)/R/    [{1, x}, {y, y + z}]

permutations (a)
function
Returns a set of all distinct permutations of the members of the list or set a. Each
permutation is a list, not a set.
When a is a list, duplicate members of a are included in the permutations.
permutations complains if a is not a literal list or set.
See also random_permutation.
Examples:

(%i1) permutations ([a, a]);
(%o1)             {[a, a]}
(%i2) permutations ([a, a, b]);
(%o2)        {[a, a, b], [a, b, a], [b, a, a]}

powerset (a)
function
powerset (a, n)
function
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^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:
Chapter 38: Sets

(%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) {{}}

Function random_permutation (a)

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]

Function rreduce (F, s)

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

rreduce(F, s) returns F(s_1, ... F(s_{n - 2}, F(s_{n - 1}, s_n))). When the optional argument s_{n + 1} is present, the result is equivalent to rreduce(F, endcons(s_{n + 1}, 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
   ---
   b d

setdifference (a, b)

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);
   {a, b, z}
(%o3) {a, b, z}
(%i4) setdifference (S_2, S_1);
   {aa, bb, zz}
(%o4) {aa, bb, zz}
(%i5) setdifference (S_1, S_1);
   {};
(%o5) {a, c, x, y, z}
(%i6) setdifference (S_1, {});
   {};
(%o6) {a, b, c, x, y, z}
(%i7) setdifference ({}, S_1);
   {};
(%o7) {}

setequalp (a, b)

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});
   true
(%o1) true
(%i2) setequalp ({a, b, c}, {1, 2, 3});
   false
(%o2) false
(%i3) setequalp ({x^2 - y^2}, {(x + y) * (x - y)});
   false
(%o3) false

setify (a)

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}

\textbf{setp} \ (a)

\textbf{Function}

Returns \texttt{true} if and only if \(a\) is a Maxima set.

\texttt{setp} returns \texttt{true} for unsimplified sets (that is, sets with redundant members) as well as simplified sets.

\texttt{setp} is equivalent to the Maxima function \texttt{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

\textbf{set_partitions} \ (a)

\textbf{Function}

\textbf{set_partitions} \ (a, n)

\textbf{Function}

Returns the set of all partitions of \(a\), or a subset of that set.

\texttt{set_partitions}(a, n) returns a set of all decompositions of \(a\) into \(n\) nonempty disjoint subsets.

\texttt{set_partitions}(a) returns the set of all partitions.

\texttt{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) \{\}\n
The cardinality of the set of partitions of a set can be found using \texttt{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);
    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);

{0, 1, 2, 3, 4, 5}

Function \texttt{some} \((f, a)\)

Returns \texttt{true} if the predicate \( f \) is \texttt{true} for one or more given arguments.

Given one set as the second argument, \texttt{some}(f, s) returns \texttt{true} if \( \text{is}(f(a_i)) \) returns \texttt{true} for one or more \( a_i \) in \( s \). \texttt{some} may or may not evaluate \( f \) for all \( a_i \) in \( s \). Since sets are unordered, \texttt{some} may evaluate \( f(a_i) \) in any order.

Given one or more lists as arguments, \texttt{some}(f, L_1, \ldots, L_n) returns \texttt{true} if \( \text{is}(f(x_1, \ldots, x_n)) \) returns \texttt{true} for one or more \( x_1, \ldots, x_n \) in \( L_1, \ldots, L_n \), respectively. \texttt{some} may or may not evaluate \( f \) for some combinations \( x_1, \ldots, x_n \). \texttt{some} evaluates lists in the order of increasing index.

Given an empty set \( \{} \) or empty lists \( [] \) as arguments, \texttt{some} returns \texttt{false}.

When the global flag \texttt{maperror} is \texttt{true}, all lists \( L_1, \ldots, L_n \) must have equal lengths. When \texttt{maperror} is \texttt{false}, list arguments are effectively truncated to the length of the shortest list.

Return values of the predicate \( f \) which evaluate (via \texttt{is}) to something other than \texttt{true} or \texttt{false} are governed by the global flag \texttt{prederror}. When \texttt{prederror} is \texttt{true}, such values are treated as \texttt{false}. When \texttt{prederror} is \texttt{false}, such values are treated as \texttt{unknown}.

Examples:

\texttt{some} applied to a single set. The predicate is a function of one argument.

(%i1) some (integerp, {1, 2, 3, 4, 5, 6});
    true
(%i2) some (atom, {1, 2, sin(3), 4, 5 + y, 6});
    true

\texttt{some} applied to two lists. The predicate is a function of two arguments.

(%i1) some ("=", [a, b, c], [a, b, c]);
    true
(%i2) some ("#", [a, b, c], [a, b, c]);
    false

Return values of the predicate \( f \) which evaluate to something other than \texttt{true} or \texttt{false} are governed by the global flag \texttt{prederror}.

(%i1) prederror : false;
    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)

Represents the Stirling number of the first kind.

When \( n \) and \( m \) are nonnegative integers, the magnitude of \( \text{stirling1}(n, m) \) is the number of permutations of a set with \( n \) members that have \( m \) cycles. For details, see Graham, Knuth and Patashnik *Concrete Mathematics*. Maxima uses a recursion relation to define \( \text{stirling1}(n, m) \) for \( m \) less than 0; it is undefined for \( n \) less than 0 and for non-integer arguments.

\( \text{stirling1} \) is a simplifying function. Maxima knows the following identities.

1. \( \text{stirling1}(0, n) = \text{kron}\_\delta(n, n) \) (Ref. [1])
2. \( \text{stirling1}(n, n) = 1 \) (Ref. [1])
3. \( \text{stirling1}(n, n - 1) = \text{binomial}(n, 2) \) (Ref. [1])
4. \( \text{stirling1}(n + 1, 0) = 0 \) (Ref. [1])
5. \( \text{stirling1}(n + 1, 1) = n! \) (Ref. [1])
6. \( \text{stirling1}(n + 1, 2) = 2^n - 1 \) (Ref. [1])

These identities are applied when the arguments are literal integers or symbols declared as integers, and the first argument is nonnegative. \( \text{stirling1} \) does not simplify for non-integer arguments.

**References:**


**Examples:**

\[(%i1)\] declare (n, integer)$
\%(i2)\] assume (n \(\geq\) 0)$
\%(i3)\] stirling1 (n, n);
\%(o3)\] 1

\(\text{stirling1} \) does not simplify for non-integer arguments.

\%(i1)\] stirling1 (sqrt(2), sqrt(2));
\%(o1)\] \(\text{stirling1}(\sqrt{2}, \sqrt{2})\)

Maxima applies identities to \( \text{stirling1} \).
%i1) declare (n, integer)$
%i2) assume (n >= 0)$
%i3) stirling1 (n + 1, n);
  n (n + 1)  
  ---------  2 
%o3)
%i4) stirling1 (n + 1, 1);
  n!
%o4)

stirling2 (n, m)

Function

Represents the Stirling number of the second kind.

When \( n \) and \( m \) are nonnegative integers, \( \text{stirling2} (n, m) \) is the number of ways a set with cardinality \( n \) can be partitioned into \( m \) disjoint subsets. Maxima uses a recursion relation to define \( \text{stirling2} (n, m) \) for \( m \) less than 0; it is undefined for \( n \) less than 0 and for non-integer arguments.

\( \text{stirling2} \) is a simplifying function. Maxima knows the following identities.

1. \( \text{stirling2}(0, n) = \text{kron}_d\delta (0, n) \) (Ref. [1])
2. \( \text{stirling2}(n, n) = 1 \) (Ref. [1])
3. \( \text{stirling2}(n, n - 1) = \text{binomial}(n, 2) \) (Ref. [1])
4. \( \text{stirling2}(n + 1, 1) = 1 \) (Ref. [1])
5. \( \text{stirling2}(n + 1, 2) = 2^n - 1 \) (Ref. [1])
6. \( \text{stirling2}(n, 0) = \text{kron}_d\delta (n, 0) \) (Ref. [2])
7. \( \text{stirling2}(n, m) = 0 \) when \( m > n \) (Ref. [2])
8. \( \text{stirling2}(n, m) = \sum ((-1)^{m - k}\text{binomial}(mk)k^n, i, 1, m)/m! \) when \( m \) and \( n \) are integers, and \( n \) is nonnegative. (Ref. [3])

These identities are applied when the arguments are literal integers or symbols declared as integers, and the first argument is nonnegative. \( \text{stirling2} \) does not simplify for non-integer arguments.

References:

Examples:

(\%i1) declare (n, integer)$
(\%i2) assume (n >= 0)$
(\%i3) stirling2 (n, n);
(\%o3) 1

\( \text{stirling2} \) does not simplify for non-integer arguments.

(\%i1) stirling2 (%pi, %pi);
(\%o1)

Maxima applies identities to \( \text{stirling2} \).
subset \((a, f)\)

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:

\[
(%i1) \text{subset}\{1, 2, x, x + y, z, x + y + z\}, \text{atom};
\]
\[
(%o1) \{1, 2, x, z\}
\]

\[
(%i2) \text{subset}\{1, 2, 7, 8, 9, 14\}, \text{evenp};
\]
\[
(%o2) \{2, 8, 14\}
\]

subsetp \((a, b)\)

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:

\[
(%i1) \text{subsetp}\{1, 2, 3\}, \{a, 1, b, 2, c, 3\};
\]
\[
(%o1) \text{true}
\]

\[
(%i2) \text{subsetp}\{a, 1, b, 2, c, 3\}, \{1, 2, 3\};
\]
\[
(%o2) \text{false}
\]

symmdifference \((a_1, \ldots, a_n)\)

Returns the symmetric difference of sets \(a_1, \ldots, a_n\).

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) \text{symmdifference}\();
\]
(\%o4) \{\}\n(\%i5) \textbf{symmdifference} (S_1);\n(\%o5) \{a, b, c\}\n(\%i6) \textbf{symmdifference} (S_1, S_2);\n(\%o6) \{1, a\}\n(\%i7) \textbf{symmdifference} (S_1, S_2, S_3);\n(\%o7) \{1, b, z\}\n(\%i8) \textbf{symmdifference} (\{\}, S_1, S_2, S_3);\n(\%o8) \{1, b, z\}

\textbf{tree\_reduce} (F, s) \hfill \textbf{Function} \\
\textbf{tree\_reduce} (F, s, s_0) \hfill \textbf{Function}

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

\textbf{tree\_reduce} is equivalent to the following: Apply $F$ to successive pairs of elements to form a new list $[F(s_{1}, s_{2}), F(s_{3}, s_{4}), \ldots]$, 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 $\text{tree\_reduce}(F, \text{cons}(s_0, s))$.

For addition of floating point numbers, $\text{tree\_reduce}$ may return a sum that has a smaller rounding error than either $\text{rreduce}$ or $\text{lreduce}$.

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

Examples:

\textbf{tree\_reduce} applied to a list with an even number of elements.

(\%i1) \textbf{tree\_reduce} (f, \{a, b, c, d\});
(\%o1) $f(f(a, b), f(c, d))$

\textbf{tree\_reduce} applied to a list with an odd number of elements.

(\%i1) \textbf{tree\_reduce} (f, \{a, b, c, d, e\});
(\%o1) $f(f(f(a, b), f(c, d)), e)$

\textbf{union} (a_1, ..., a_n) \hfill \textbf{Function}

Returns the union of the sets $a_1$ through $a_n$.

$\text{union()}$ (with no arguments) returns the empty set.

$\text{union}$ 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) \text{union} ();
Chapter 38: Sets 527

(%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\}

\textbf{xreduce} (F, s) \quad \text{Function}

\textbf{xreduce} (F, s, s_0) \quad \text{Function}

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, \texttt{xreduce} is the same as \texttt{lreduce}. The argument \(s\) is a list.

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

When the optional argument \(s_0\) is present, the result is equivalent to \texttt{xreduce}((\(s\), \texttt{cons}((\(s_0\)), \(s\))).

Floating point addition is not exactly associative; be that as it may, \texttt{xreduce} applies Maxima's \(n\)-ary addition when \(s\) contains floating point numbers.

Examples:

\texttt{xreduce} applied to a function known to be \(n\)-ary. \(F\) is called once, with all arguments.

\%(i1) declare (F, nary);\n\%(o1) done\n\%(i2) F ([L]) := L;\n\%(o2) F([L]) := L\n\%(i3) xreduce (F, [a, b, c, d, e]);\n\%(o3) \[[[[["[", simp], a], b], c], d], e\]

\texttt{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;\n\%(o1) \%\(G([L]) := L\n\%(i2) xreduce (G, [a, b, c, d, e]);\n\%(o2) \[[[[["[", simp], a], b], c], d], e\]
\%(i3) lreduce (G, [a, b, c, d, e]);\n\%(o3) \[[[[a, b], c], d], e\]
39 Structures

39.1 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.

39.2 Functions and Variables for Structures

`structures` Global variable

`structures` is the list of user-defined structures defined by `defstruct`.

`defstruct (S(a_1, ..., a_n))` Function

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

Define a structure, which is a list of named fields $a_1, ..., a_n$ 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);
```
new (S)  Function
new (S (v₁, ..., vₙ))  Function

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(v₁, ..., vₙ)) creates a new instance of S in which fields are assigned the values v₁, ..., vₙ.

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

@  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;
Chapter 39: Structures

(%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
Chapter 40: Function Definition

40 Function Definition

40.1 Introduction to Function Definition

40.2 Function

40.2.1 Ordinary functions

To define a function in Maxima you use the := operator. E.g.

\[
f(x) := \sin(x)
\]
defines a function \( f \). Anonmyous functions may also be created using lambda. For example

\[
\text{lambda ([i, j], ...)}
\]
can be used instead of \( f \) where

\[
f(i, j) := \text{block ([]} \ldots);
\]
\[
\text{map (lambda ([i], i+1), l)}
\]
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:

\[
\begin{align*}
\text{(%i1) } & f ([u]) := u; \\
\text{(%o1) } & f([u]) := u \\
\text{(%i2) } & f (1, 2, 3, 4); \\
\text{(%o2) } & [1, 2, 3, 4] \\
\text{(%i3) } & f (a, b, [u]) := [a, b, u]; \\
\text{(%o3) } & f(a, b, [u]) := [a, b, u] \\
\text{(%i4) } & f (1, 2, 3, 4, 5, 6); \\
\text{(%o4) } & [1, 2, [3, 4, 5, 6]]
\end{align*}
\]

The right hand side of a function is an expression. Thus if you want a sequence of expressions, you do

\[
f(x) := (\text{expr1, expr2, ..., exprn});
\]
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.

\[
\text{block ([]} \ldots, \text{if (a > 10) then return(a), ...}, \text{exprn})
\]
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 \( \mathbf{c} : \mathbf{a} \) in the above, the value of \( \mathbf{c} \) would have been the value of \( \mathbf{a} \) at the time you just entered the block, but before \( \mathbf{a} \) was bound. Thus doing something like

\[
\text{block ([a: a], expr1, ... a: a+3, ..., exprn)}
\]

will protect the external value of \( \mathbf{a} \) 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 \( \text{block ([x], ...} \) would cause the \( \mathbf{x} \) 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

\[
f(\mathbf{x}) := (\text{expr1, ..., exprn});
\]

and

\[
f(1);
\]

we would have a similar context for evaluation of the expressions as if we had done

\[
\text{block ([x: 1], expr1, ..., exprn)}
\]

Inside functions, when the right hand side of a definition, may be computed at runtime, it is useful to use \text{define} and possibly \text{buildq}.

40.2.2 Array functions

An array function stores the function value the first time it is called with a given argument, and returns the stored value, without recomputing it, when that same argument is given. Such a function is often called a memoizing function.

Array function names are appended to the global list \text{arrays} (not the global list \text{functions}). \text{arrayinfo} returns the list of arguments for which there are stored values, and \text{listarray} returns the stored values. \text{dispfun} and \text{fundef} return the array function definition.

\text{arraymake} constructs an array function call, analogous to \text{funmake} for ordinary functions. \text{arrayapply} applies an array function to its arguments, analogous to \text{apply} for ordinary functions. There is nothing exactly analogous to \text{map} for array functions, although \text{map(lambda([x], a[x]), L)} or \text{makelist(a[x], x, L)}, where \( L \) is a list, are not too far off the mark.

\text{remarray} removes an array function definition (including any stored function values), analogous to \text{remfunction} for ordinary functions.

\text{kill(a[x])} removes the value of the array function \( \mathbf{a} \) stored for the argument \( \mathbf{x} \); the next time \( \mathbf{a} \) is called with argument \( \mathbf{x} \), the function value is recomputed. However, there is no way to remove all of the stored values at once, except for \text{kill(a)} or \text{remarray(a)}, which also remove the function definition.

40.3 Macros

\text{buildq} \ (L, \ expr) \hspace{1cm} \text{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 \text{buildq} carries out the substitution.
The elements of $L$ are symbols or assignment expressions $\text{symbol: value}$, 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 $\text{buildq}$ 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 $\text{buildq}$ is the same as in the context from which $\text{buildq}$ was called.

Then the variables named by $L$ are substituted into $\text{expr}$ 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 $\text{splice (x)}$ in $\text{expr}$, then $x$ must be bound to a list, and the list is spliced (interpolated) into $\text{expr}$ instead of substituted.

Any variables in $\text{expr}$ not appearing in $L$ are carried into the result verbatim, even if they have bindings in the context from which $\text{buildq}$ 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);
   x + c + 29
(%i3) `'%
   x + 1758
```

e is bound to a list, which appears as such in the arguments of $\text{foo}$, and interpolated into the arguments of $\text{bar}$.

```
(%i1) buildq ([e: [a, b, c]], foo (x, e, y));
   foo(x, [a, b, c], y)
(%i2) buildq ([e: [a, b, c]], bar (x, splice (e), y));
   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));
   2 c + 2 b + 2 a
(%i2) buildq ([e: [a, b, c]], 2 * splice (e));
   2 a b c
```

The variables in $L$ are bound in parallel; if bound sequentially, the first result would be $\text{foo (b, b)}$. Substitutions are carried out in parallel; compare the second result with the result of $\text{subst}$, which carries out substitutions sequentially.

```
(%i1) buildq ([a: b, b: a], foo (a, b));
   foo(b, a)
(%i2) buildq ([u: v, v: w, w: x, x: y, y: z, z: u],
   bar (u, v, w, x, y, z));
   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));
   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`.

```lisp
(%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.

```lisp
(%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)*

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

```lisp
(%i1) g (x) ::= x / 99;
(%o1) g(x) ::= --
 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
74
(%i5) h (y);
(%o5) y - 1234
74
```
**Function**

**macroexpand1** (*expr*)

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;
   g(x) ::= --
   99
(%o1)

(%i2) h (x) ::= buildq ([x], g (x - a));
   h(x) ::= buildq([x], g(x - a))
(%o2)

(%i3) a: 1234;
   a = 1234
(%o3)

(%i4) macroexpand1 (h (y));
   g(y - a)
(%o4)

(%i5) h (y);
   y - 1234
   --------
   99
(%o5)
```

**Global variable**

**macros**

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**.

**Function**

**splice** (*a*)

Splices (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 ".

Examples

```
apply \((F, [x_1, \ldots, x_n])\)  

Constructs and evaluates an expression \(F(arg_1, \ldots, arg_n)\).

apply does not attempt to distinguish array functions from ordinary functions; when \(F\) is the name of an array function, apply evaluates \(F(\ldots)\) (that is, a function call with parentheses instead of square brackets). arrayapply evaluates a function call with square brackets in this case.

Examples:

apply evaluates its arguments. In this example, \(\text{min}\) is applied to the value of \(L\).

\begin{verbatim}
(%i1) L : [1, 5, -10.2, 4, 3];
(%o1) [1, 5, -10.2, 4, 3]
(%i2) apply (min, L);
(%o2) -10.2
\end{verbatim}

apply evaluates arguments, even if the function \(F\) quotes them.

\begin{verbatim}
(%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);
fname is not the name of a user function.
-- an error. Quitting. To debug this try debugmode(true);
(%i5) apply (dispfun, [fname]);
(%t5) F(x) := ----
       1729
\end{verbatim}
apply evaluates the function name $F$. Single quote ' defeats evaluation. demoivre is the name of a global variable and also a function.

```lisp
(%i1) demoivre;
(%o1) false
(%i2) demoivre (exp (%i * x));
(%o2) %i sin(x) + cos(x)
(%i3) apply (demoivre, [exp (%i * x)]);
```

demoivre evaluates to false
Improper name or value in functional position.
-- an error. Quitting. To debug this try debugmode(true);

```lisp
(%i4) apply ('demoivre, [exp (%i * x)]);
```

Function block

```lisp
block ([v_1, ..., v_m], expr_1, ..., expr_n)
```

Function block

block evaluates expr_1, ..., expr_n in sequence and returns the value of the last expression evaluated. 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. Some variables $v_1, ..., v_m$ can be declared local to the block; these are distinguished from global variables of the same names. If no variables are declared local then the list may be omitted. Within the block, any variable other than $v_1, ..., v_m$ is a global variable.

block saves the current values of the variables $v_1, ..., v_m$ (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.

The declaration `local($v_1, ..., v_m$) within block saves the properties associated with the symbols $v_1, ..., v_m$, 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.

**break** (expr_1, ..., expr_n)

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)

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.

```lisp
(%i1) lambda ([x], if x < 0 then throw(x) else f(x));
(%i2) g(l) := 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.

**compfile** (filename, f_1, ..., f_n)

**compfile** (filename, functions)

**compfile** (filename, all)

Translates Maxima functions into Lisp and writes the translated code into the file filename.

`compfile(filename, f_1, ..., f_n)` translates the specified functions. `compfile (filename, functions)` and `compfile (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** (f_1, ..., f_n)

**compile** (functions)

**compile** (all)

Translates Maxima functions f_1, ..., 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.
Chapter 40: Function Definition

compile (all) or compile (functions) compiles all user-defined functions.
compile quotes its arguments; the quote-quote operator ’’ defeats quotation.

\texttt{define \((f(x_1, \ldots, x_n), \text{expr})\)} Function
\texttt{define \((f[x_1, \ldots, x_n], \text{expr})\)} Function
\texttt{define \((\text{funmake} \,(f, [x_1, \ldots, x_n]), \text{expr})\)} Function
\texttt{define \((\text{arraymake} \,(f, [x_1, \ldots, x_n]), \text{expr})\)} Function
\texttt{define \((\text{ev} \,(\text{expr}_1), \text{expr}_2)\)} Function

Defines a function named \(f\) with arguments \(x_1, \ldots, x_n\) and function body \(\text{expr}\).

\texttt{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 an array 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 \texttt{define} accepts a variable number of arguments. Actual arguments are assigned one-to-one to formal arguments \(x_1, \ldots, x_{(n - 1)}\), and any further actual arguments, if present, are assigned to \(x_n\) as a list.

When the first argument of \texttt{define} is an expression of the form \(f(x_1, \ldots, x_n)\) or \(f[x_1, \ldots, x_n]\), 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 \texttt{funmake}, \texttt{arraymake}, or \texttt{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, \texttt{local(f)} makes the definition of function \(f\) effective only within the block or other compound expression in which \texttt{local} appears.

If some formal argument \(x_k\) is a quoted symbol (after evaluation), the function defined by \texttt{define} does not evaluate the corresponding actual argument. Otherwise all actual arguments are evaluated.

See also \(:=\) and \(::=\).

Examples:

\texttt{define} always evaluates its second argument (unless explicitly quoted).

\begin{verbatim}
(%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)
\end{verbatim}

The function defined by \texttt{define} may be an ordinary Maxima function or an array 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 \( x_n \) is a list of one element, the function defined by \texttt{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 \texttt{funmake}, \texttt{arraymake}, or \texttt{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)

\texttt{define_variable} \ (\text{\texttt{name}, default_value, mode})

\texttt{define_variable} \ introduces a global variable into the Maxima environment. \texttt{define_variable} \ is useful in user-written packages, which are often translated or compiled.

\texttt{define_variable} \ carries out the following steps:

1. \texttt{mode_declare} \ (\texttt{name, mode}) \ declares the mode of \texttt{name} to the translator. See \texttt{mode_declare} \ for a list of the possible modes.

2. If the variable is unbound, \texttt{default_value} \ is assigned to \texttt{name}.

3. \texttt{declare} \ (\texttt{name, special}) \ declares it special.

4. Associates \texttt{name} \ with a test function to ensure that \texttt{name} \ is only assigned values of the declared mode.

The \texttt{value_check} \ property can be assigned to any variable which has been defined via \texttt{define_variable} \ with a mode other than \texttt{any}. The \texttt{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 \texttt{value_check} \ function is the would-be assigned value.

\texttt{define_variable} \ evaluates \texttt{default_value}, and quotes \texttt{name} \ and \texttt{mode}. \texttt{define_variable} \ returns the current value of \texttt{name}, which is \texttt{default_value} \ if \texttt{name} \ was unbound before, and otherwise it is the previous value of \texttt{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;  
Error: foo was declared mode boolean, has value: %pi  
-- an error. Quitting. 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. Quitting. 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 $\neq$ 'baz_quux then error("Cannot assign to 'baz_quux'."))(y=sqrt(2))
-- an error. Quitting. To debug this try debugmode(true);
(%i6) baz_quux;
(%o6) baz_quux

\textbf{dispfun (f_1, \ldots, f_n)} \quad \text{Function}

\textbf{dispfun (all)} \quad \text{Function}

Displays the definition of the user-defined functions $f_1, \ldots, f_n$. Each argument may be the name of a macro (defined with \texttt{::=}), an ordinary function (defined with \texttt{:=} or \texttt{define}), an array function (defined with \texttt{:=} or \texttt{define}, but enclosing arguments in square brackets \([\ ]\)), a subscripted function, (defined with \texttt{:=} or \texttt{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.

\texttt{dispfun (all)} displays all user-defined functions as given by the \texttt{functions}, \texttt{arrays}, and \texttt{macros} lists, omitting subscripted functions defined with constant subscripts.

\texttt{dispfun} creates an intermediate expression label (%t1, %t2, etc.) for each displayed function, and assigns the function definition to the label. In contrast, \texttt{fundef} returns the function definition.

\texttt{dispfun} quotes its arguments; the quote-quote operator \texttt{''} defeats quotation.

\texttt{dispfun} returns the list of intermediate expression labels corresponding to the displayed functions.

Examples:

(%i1) m(x, y) ::= x\^{-y};
- y
(%o1) m(x, y) ::= x

(%i2) f(x, y) := x\^{-y};
- y
(%o2) f(x, y) := x

(%i3) g[x, y] := x\^{-y};
- y
(%o3) g := x

(%i4) h[x](y) := x\^{-y};
- y
(%o4) h (y) := x

(%i5) i[8](y) := 8\^{-y};
- y
(%o5) i (y) := 8

(%i6) dispfun (m, f, g, h, h[5], h[10], i[8]);
- y
(%o6) m(x, y) ::= x
- y

(%o7) f(x, y) ::= x
Chapter 40: Function Definition

-module.

(%t8) g := x
   x, y
   - y

(%t9) h (y) := x
   x

(%t10) h (y) := --
      5 y
      5

(%t11) h (y) := ---
     10 y
     10

(%t12) i (y) := 8
     8

(%o12) [%t6, %t7, %t8, %t9, %t10, %t11, %t12]

(%i12) ''%;

(%o12) [m(x, y) ::= x , f(x, y) := x , g := x ,
     x, y
     - y
     - y
     - y
     h (y) := x , h (y) := -- , h (y) := --- , i (y) := 8 ]
     x 5 y 10 y 8

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.

Array 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;
\[
\begin{align*}
(\%o2) & \quad F_2(x, y) := -x/y \\
(\%i3) & \quad \text{define (F}_3(x):= \text{sqrt}(x)); \\
(\%o3) & \quad F_3(x) := \text{sqrt}(x) \\
(\%i4) & \quad G_1[x] := x - 100; \\
(\%o4) & \quad G_1 := x - 100 \\
(\%i5) & \quad G_2[x, y] := x / y; \\
(\%o5) & \quad G_2 := x/y \\
(\%i6) & \quad \text{define (G}_3[x], \text{sqrt}(x)); \\
(\%o6) & \quad G_3 := \text{sqrt}(x) \\
(\%i7) & \quad H_1[x](y) := x^y; \\
(\%o7) & \quad H_1(y) := x^y \\
(\%i8) & \quad \text{functions;} \\
(\%o8) & \quad [F_1(x), F_2(x, y), F_3(x)] \\
(\%i9) & \quad \text{arrays;} \\
(\%o9) & \quad [G_1, G_2, G_3, H_1]
\end{align*}
\]

### 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 \text{define} \), an array function (defined with \( := \) or \text{define} \), but enclosing arguments in square brackets [ \( ] \), a subscripted function, (defined with \( := \) or \text{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.

\text{fundef} quotes its argument; the quote-quote operator \( '' \) defeats quotation.

\text{fundef} \( (f) \) returns the definition of \( f \). In contrast, \text{dispfun} \( (f) \) creates an intermediate expression label and assigns the definition to the label.

### funmake \( (F, [arg_1, ..., arg_n]) \)

Function

Returns an expression \( F(arg_1, ..., arg_n) \). The return value is simplified, but not evaluated, so the function \( F \) is not called, even if it exists.

\text{funmake} does not attempt to distinguish array functions from ordinary functions; when \( F \) is the name of an array function, \text{funmake} returns \( F(...) \) (that is, a function call with parentheses instead of square brackets). \text{arraymake} returns a function call with square brackets in this case.

\text{funmake} evaluates its arguments.

Examples:

\text{funmake} applied to an ordinary Maxima function.
funmake applied to a macro.

\[
\text{ (%i1) } G(x) ::= \frac{x - 1}{2};
\]
\[
\text{ (%o1) } G(x) ::= \frac{x - 1}{2}
\]

funmake applied to a subscripted function.

\[
\text{ (%i1) } H[a] (x) := (x - 1)^a;
\]
\[
\text{ (%o1) } H(x) := (x - 1)^a
\]

funmake applied to a symbol which is not a defined function of any kind.

\[
\text{ (%i1) } \text{funmake (A, [u]);}
\]
\[
\text{ (%o1) } A(u)
\]

funmake evaluates its arguments, but not the return value.

\[
\text{ (%i1) } \text{det(a,b,c) := b^2 - 4*a*c;}
\]
\[
\text{ (%o1) } \text{det(a, b, c) := b - 4 a c}
\]
(%i13) f : det;
(%o13) det
(%i14) funmake (f, [x, y, z]);
(%o14) det(8, 10, 12)
(%i15) ''%;
(%o15) -284

Maxima simplifies funmake's return value.
(%i11) funmake (sin, [%pi / 2]);
(%o11) 1

lambda (\[x_1, \ldots, x_m\], expr_1, \ldots, expr_n) Function
lambda (\[[L]\], expr_1, \ldots, expr_n) Function
lambda (\[x_1, \ldots, x_m, [L]\], expr_1, \ldots, expr_n) Function

 Defines and returns a lambda expression (that is, an anonymous function). The function may have required arguments \(x_1, \ldots, x_m\) 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 \(x_1, \ldots, x_m\) 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.
  (\%i11) f: lambda ([x], x^2);
  (\%o11) lambda([x], x )
  (\%i12) f(a);
  (\%o12) a
• A lambda expression may appear in contexts in which a function evaluation is expected.
Chapter 40: Function Definition

(%i3) lambda ([x], x^2) (a);
    2
(%o3) a
(%i4) apply (lambda ([x], x^2), [a]);
    2
(%o4) a
(%i5) map (lambda ([x], x^2), [a, b, c, d, e]);
2 2 2 2 2
(%o5) [a, b, c, d, e]

- 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 "'.

  (%i6) a: %pi$
  (%i7) b: %e$
  (%i8) g: lambda ([a], a*b);
  (%o8) lambda([a], a b)
  (%i9) b: %gamma$
  (%i10) g(1/2);

  2
  (%o10) ------
       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.

  (%i11) g2: lambda ([a], a*''b);
  (%o11) lambda([a], a %gamma)
  (%i12) b: %e$
  (%i13) g2(1/2);

  2
  (%o13) ------
       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.

  (%i14) h: lambda ([a, b], h2: lambda ([a], a*b), h2(1/2));

  1
  (%o14) lambda([a, b], h2 : lambda([a], a b), h2(-))

  (%i15) h(%pi, %gamma);

  2
  (%o15) ------
       2
(%i19) i2(1/2);
   x
(%o19) lambda([x], -)
   2

(%i20) i2(1/2)(%pi);
%pi
(%o20) ----
  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);
   lambda([aa, bb, [cc]], aa cc + bb)
(%i2) f (foo, %i, 17, 29, 256);
   [17 foo + %i, 29 foo + %i, 256 foo + %i]

(%i3) g : lambda ([[[aa]], apply ("+", aa));
   lambda([[[aa]], apply(+, aa))
(%i4) g (17, 29, x, y, z, %e);
   z + y + x + %e + 46

\textbf{local} \ (v_1, \ldots, v_n)

Function

Saves the properties associated with the symbols \(v_1, \ldots, v_n\), removes any properties before evaluating other expressions, and restores any saved properties on exit from the block or other compound expression in which \texttt{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 \texttt{local} is to make such declarations effective only within the block or other compound expression in which \texttt{local} appears; otherwise such declarations are global declarations.

\texttt{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.

\texttt{local} quotes its arguments. \texttt{local} returns \texttt{done}.

Example:

A local function definition.

\begin{verbatim}
(%i1) foo (x) := 1 - x;
   foo(x) := 1 - x
(%o1)

(%i2) foo (100);  
   - 99
(%o2)

(%i3) block (local (foo), foo (x) := 2 * x, foo (100));
   200
(%o3)

(%i4) foo (100);  
   - 99
(%o4)
\end{verbatim}

\textbf{macroexpansion}

Option variable

Default value: \texttt{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);
    h(x)
(%o1) f(x) := ----
    g(x)
(%i2) g (x) ::= block (print("x + 99 is equal to", x),
    return (x + 99));
(%o2) g(x) ::= block(print("x + 99 is equal to", x),
    return(x + 99))
(%i3) h (x) ::= block (print("x - 99 is equal to", x),
    return (x - 99));
(%o3) h(x) ::= block(print("x - 99 is equal to", x),
    return(x - 99))
(%i4) macroexpansion: false;
(%o4) false
(%i5) f (a * b);
    a b - 99
(%o5) ------
    a b + 99
(%i6) dispfun (f);
    h(x)
(%t6) f(x) := ----
    g(x)
(%o6) done
(%i7) f (a * b);
```
x - 99 is equal to x
x + 99 is equal to x

(%o7)
\[
\frac{a \cdot b - 99}{a \cdot b + 99}
\]

When \texttt{macroexpansion} is \texttt{expand}, a macro function is called once, and the calling expression is not modified.

(%i1) f (x) := h (x) / g (x);
(%o1)
\[
f(x) := \frac{h(x)}{g(x)}
\]

(%i2) g (x) ::= block (print ("x + 99 is equal to", x),
\[\text{\hspace{1cm} return (x + 99)}\])
(%o2) g(x) ::= block(print("x + 99 is equal to", x),
\[\text{\hspace{1cm} return(x + 99)})

(%i3) h (x) ::= block (print ("x - 99 is equal to", x),
\[\text{\hspace{1cm} return (x - 99)}\])
(%o3) h(x) ::= block(print("x - 99 is equal to", x),
\[\text{\hspace{1cm} return(x - 99)})

(%i4) macroexpansion: \texttt{expand};
(%o4) \texttt{expand}

(%i5) f (a * b);
x - 99 is equal to x
x + 99 is equal to x
\[
\frac{a \cdot b - 99}{a \cdot b + 99}
\]

(%i6) dispfun (f);
(h(x)
(f(x) := \frac{h(x)}{g(x)}

(%i6) done

(%i7) f (a * b);
\[
\frac{a \cdot b - 99}{a \cdot b + 99}
\]

When \texttt{macroexpansion} is \texttt{expand}, a macro function is called once, and the calling expression is modified.

(%i1) f (x) := h (x) / g (x);
(%o1)
\[
f(x) := \frac{h(x)}{g(x)}
\]

(%i2) g (x) ::= block (print ("x + 99 is equal to", x),
\[\text{\hspace{1cm} return (x + 99)}\])
(%o2) g(x) ::= block(print("x + 99 is equal to", x),
\[\text{\hspace{1cm} return(x + 99)})

(%i3) h (x) ::= block (print ("x - 99 is equal to", x),
\[\text{\hspace{1cm} return (x - 99)}\])
(%o3) h(x) ::= block(print("x - 99 is equal to", x),
\[\text{\hspace{1cm} return(x - 99)})

(x - 99 is equal to x
x + 99 is equal to x

(%o7)
\[
\frac{a \cdot b - 99}{a \cdot b + 99}
\]
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
------
a b + 99
x - 99

(%i6) dispfun (f);
f(x) := ------
x + 99

(%o6) done
(%i7) f (a * b);
a b - 99
------
a b + 99

mode_checkp Option variable
Default value: true
When mode_checkp is true, mode_declare checks the modes of bound variables.

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_declare (y_1, mode_1, ..., y_n, mode_n) Function
mode_declare is used to declare the modes of variables and functions for subsequent translation or compilation of functions. mode_declare is typically placed at the beginning of a function definition, at the beginning of a Maxima script, or executed at the interactive prompt.

The arguments of mode_declare are pairs consisting of a variable and a mode which is one of boolean, fixnum, number, rational, or float. Each variable may also be a list of variables all of which are declared to have the same mode.

If a variable is an array, and if every element of the array which is referenced has a value then array (yi, complete, dim1, dim2, ...) rather than array(yi, dim1, dim2, ...)
should be used when first declaring the bounds of the array. If all the elements of the array are of mode `fixnum (float)`, use `fixnum (float)` instead of `complete`. Also if every element of the array is of the same mode, say `m`, then

\[
\text{mode\_declare (completearray (yi), m)}
\]

should be used for efficient translation.

Numeric code using arrays might run faster by declaring the expected size of the array, as in:

\[
\text{mode\_declare (completearray (a [10, 10]), float)}
\]

for a floating point number array which is 10 x 10.

One may declare the mode of the result of a function by using `function (f_1, f_2, ...,)` as an argument; here `f_1, f_2, ...` are the names of functions. For example the expression,

\[
\text{mode\_declare ([function (f_1, f_2, ...)], fixnum)}
\]

declares that the values returned by `f_1, f_2, ...` are single-word integers.

\text{mode\_declare} is a synonym for `mode\_declare`.

\textbf{mode\_identity (arg\_1, arg\_2)}

A special form used with `mode\_declare` and `macros` to declare, e.g., a list of lists of flonums, or other compound data object. The first argument to `mode\_identity` is a primitive value mode name as given to `mode\_declare` (i.e., one of `float, fixnum, number, list, or any`), and the second argument is an expression which is evaluated and returned as the value of `mode\_identity`. However, if the return value is not allowed by the mode declared in the first argument, an error or warning is signalled. The important thing is that the mode of the expression as determined by the Maxima to Lisp translator, will be that given as the first argument, independent of anything that goes on in the second argument. E.g., \texttt{x: 3.3; mode\_identity (fixnum, x)}; yields an error. \texttt{mode\_identity (flonum, x)} returns 3.3. This has a number of uses, e.g., if you knew that \texttt{first (l)} returned a number then you might write \texttt{mode\_identity (number, first (l))}. However, a more efficient way to do it would be to define a new primitive,

\[
\text{firstnumb (x) ::= buildq ([x], mode\_identity (number, x))}
\]

and use `firstnumb` every time you take the first of a list of numbers.

\textbf{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.

\texttt{compfile} sets `transcompile: true` for the duration.

\textbf{translate (\ell_1, ..., \ell_n)}

\textbf{translate (functions)}

\textbf{translate (all)}

Translates the user-defined functions \texttt{\ell_1, ..., \ell_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:

\[
f (x_1, x_2, \ldots) := \text{block} ([v_1, v_2, \ldots], \\
\text{mode\_declare} (v_1, \text{mode}_1, v_2, \text{mode}_2, \ldots), \ldots)
\]

where the \(x_1, x_2, \ldots\) are the parameters to the function and the \(v_1, v_2, \ldots\) 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** (maxima\_filename) Function

**translate_file** (maxima\_filename, lisp\_filename) Function

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,
tr_warn_undefined_variable, and
tr_windy.

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 messag esp**
Default value: false
When \texttt{tr_file tty messag esp} is true, messages generated by \texttt{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**
Default value: true
Tells the Maxima-to-Lisp translator to assume that the functions \texttt{acos}, \texttt{asin}, \texttt{asec}, and \texttt{acsc} can return complex results.
The ostensible effect of \texttt{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 \texttt{acos(x)} is of mode \texttt{any} even if \( x \) is of mode \texttt{float} (as set by \texttt{mode declare}). When false then \texttt{acos(x)} is of mode \texttt{float} if and only if \( x \) is of mode \texttt{float}.

**tr function call default**
Default value: general
\texttt{false} means give up and call \texttt{meval}, \texttt{expr} means assume Lisp fixed arg function. \texttt{general}, the default gives code good for \texttt{mexprs} and \texttt{mlexprs} but not \texttt{macros}. \texttt{general} assures variable bindings are correct in compiled code. In \texttt{general} mode, when translating \( F(X) \), if \( F \) is a bound variable, then it assumes that \texttt{apply(f, [x])} is meant, and translates a such, with apropriate 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**
Default value: false
When \texttt{tr numer} is true, \texttt{numer} properties are used for atoms which have them, e.g. %pi.

**tr optimize max loop**
Default value: 100
\texttt{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**
Default value: false
When \texttt{tr semicompile} is true, \texttt{translate file} and \texttt{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 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: `compfile`

- 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: `compfile`

- 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.

**tr_windy**

Option variable

Default value: `true`

- Generate helpful comments and programming hints.
**Chapter 40: Function Definition**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>compile_file (filename)</code></td>
<td>Translates the Maxima file <code>filename</code> into Lisp, executes the Lisp compiler, and, if the translation and compilation succeed, loads the compiled code into Maxima. 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 <code>false</code>. 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 <code>:=</code> operator, macros define with the <code>::=</code> operator, <code>alias</code>, <code>declare</code>, <code>define_variable</code>, <code>mode_declare</code>, and <code>infix</code>, <code>matchfix</code>, <code>nofix</code>, <code>postfix</code>, <code>prefix</code>, and <code>compfile</code>. Assignments and function calls are not evaluated until the compiled code is loaded. In particular, within the Maxima file, assignments to the translation flags (<code>tr_numer</code>, etc.) have no effect on the translation. <code>filename</code> may not contain <code>:lisp</code> statements. <code>compile_file</code> evaluates its arguments.</td>
</tr>
<tr>
<td><code>declare_translated (f_1, f_2, ...)</code></td>
<td>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. <code>(MFUNCTION-CALL fn arg1 arg2 ...)</code> is generated when the translator does not know <code>fn</code> is going to be a Lisp function.</td>
</tr>
</tbody>
</table>
Chapter 41: Program Flow

41 Program Flow

41.1 Introduction to Program Flow

Maxima provides a do loop for iteration, as well as more primitive constructs such as go.

41.2 Functions and Variables for Program Flow

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
do

The do statement is used for performing iteration. Due to its great generality the do statement will be described in two parts. First the usual form will be given which is analogous to that used in several other programming languages (Fortran, Algol, PL/I, etc.); then the other features will be mentioned.

There are three variants of this form that differ only in their terminating conditions. They are:

- for variable: initial_value step increment thru limit do body
- for variable: initial_value step increment while condition do body
- for variable: initial_value step increment unless condition do body

(Alternatively, the step may be given after the termination condition or limit.)

initial_value, increment, limit, and body can be any expressions. If the increment is 1 then "step 1" may be omitted.

The execution of the do statement proceeds by first assigning the initial_value to the variable (henceforth called the control-variable). Then: 1) If the control-variable 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 \( \text{while } i <= 10 \) is equivalent to \( \text{unless } i > 10 \) and also \( \text{thru} \ 10 \).

\[(\%i1)\ \text{series: 1}$
\[(\%i2)\ \text{term: exp (sin (x))}$
\[(\%i3)\ for\ p: 1\ unless\ p > 7\ do\]
\[
\quad (\text{term: diff (term, x)/p},
\quad \text{series: series + subst (x=0, term)*x^p})$
\[(\%i4)\ \text{series;}
\quad 7 x x x x x
\quad 6 x x x x
\quad 5 x x
\quad 4 x
\quad 2$
\[(\%o4)\ \text{which gives 8 terms of the Taylor series for } e^{-\sin(x)}.$

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\ \text{next } 3*\text{count thru 20\ do\ display (count)}$
\[
\quad \text{count} = 2$
\[
\quad \text{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) 2.236067977499797
(%i4) sqrt(5), numer;
(%o4) 2.236067977499797

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))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%i1) for f in [log, rho, atan] do ldisp(f(1))$
**errcatch** \((\text{expr}_1, \ldots, \text{expr}_n)\) \hspace{1cm} \text{Function}

Evaluates \(\text{expr}_1, \ldots, \text{expr}_n\) one by one and returns \([\text{expr}_n]\) (a list) if no error occurs. If an error occurs in the evaluation of any argument, \textit{errcatch} prevents the error from propagating and returns the empty list \([]\) without evaluating any more arguments.

\textit{errcatch} is useful in batch files where one suspects an error might occur which would terminate the batch if the error weren’t caught.

**error** \((\text{expr}_1, \ldots, \text{expr}_n)\) \hspace{1cm} \text{System variable}

Evaluates and prints \(\text{expr}_1, \ldots, \text{expr}_n\), and then causes an error return to top level Maxima or to the nearest enclosing \textit{errcatch}.

The variable \textit{error} is set to a list describing the error. The first element of \textit{error} is a format string, which merges all the strings among the arguments \(\text{expr}_1, \ldots, \text{expr}_n\), and the remaining elements are the values of any non-string arguments.

\textit{errormsg()} formats and prints \textit{error}. This is effectively reprinting the most recent error message.

**errormsg** () \hspace{1cm} \text{Function}

Reprints the most recent error message. The variable \textit{error} holds the message, and \textit{errormsg} formats and prints it.

**errormsg** \hspace{1cm} \text{Option variable}

Default value: \texttt{true}

When \texttt{false} the output of error messages is suppressed.

The option variable \textit{errormsg} can not be set in a block to a local value. The global value of \textit{errormsg} is always present.

\begin{verbatim}
(%i1) errormsg;
(%o1) true
(%i2) sin(a,b);
Wrong number of arguments to sin
   -- an error. To debug this try: debugmode(true);
(%i3) errormsg:false;
(%o3) false
(%i4) sin(a,b);
   -- an error. To debug this try: debugmode(true);
\end{verbatim}

The option variable \textit{errormsg} can not be set in a block to a local value.

\begin{verbatim}
(%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;
\end{verbatim}
for

Used in iterations. See do for a description of Maxima's iteration facilities.

Special operator

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.

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.

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, if cond_1 then expr_1 is equivalent to if cond_1 then expr_1 else false, and 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, ..., 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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than</td>
<td>&lt;</td>
<td>relational infix</td>
</tr>
<tr>
<td>less than or equal to</td>
<td>&lt;=</td>
<td>relational infix</td>
</tr>
<tr>
<td>equality (syntactic)</td>
<td>=</td>
<td>relational infix</td>
</tr>
<tr>
<td>negation of</td>
<td>#</td>
<td>relational infix</td>
</tr>
</tbody>
</table>
equality (value) equal relational function
negation of equal notequal relational function
greater than > relational infix
or equal to >= relational infix
greater than > relational infix
and and logical infix
or or logical infix
not not logical prefix

map \( f, \text{expr}_1, ..., \text{expr}_n \) 
Returns an expression whose leading operator is the same as that of the expressions \( \text{expr}_1, ..., \text{expr}_n \) 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 \( \text{expr}_i \) if not all of the \( \text{expr}_i \) are of the same length and (2) apply \( f \) to \([\text{expr}_1, \text{expr}_2, ...]\) if the \( \text{expr}_i \) 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.

\[
(\%i1) \quad \text{map}(f,x+a*y+b*z);
(\%o1) \quad f(b \ z) + f(a \ y) + f(x)
\]

\[
(\%i2) \quad \text{map}(\lambda(u), \text{partfrac}(u, x)), x+1/(x^3+4*x^2+5*x+2));
(\%o2) \quad \quad \frac{1}{x + 2} + \frac{1}{x + 1} + \frac{1}{(x + 1)}
\]

\[
(\%i3) \quad \text{map}(\text{ratsimp}, x/(x^2+x)+(y^2+y)/y);
(\%o3) \quad \quad \frac{1}{x + 1}
\]

\[
(\%i4) \quad \text{map}(\"\=\", [a,b],[-0.5,3]);
(\%o4) \quad \quad [a = -0.5, b = 3]
\]

mapatom \( \text{expr} \) 
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

\[
\text{map} (f, \text{expr}_1, \text{expr}_2, \ldots)
\]
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 \([ expr_1, expr_2, \ldots ]\) if the \( expr_i \) are not all the same type of object.

If \texttt{maperror} is \texttt{true} then an error message is displayed in the above two instances.

\texttt{mapprint} \hfill \text{Option variable}

Default value: \texttt{true}

When \texttt{mapprint} is \texttt{true}, various information messages from \texttt{map}, \texttt{mapl}, and \texttt{fullmap} are produced in certain situations. These include situations where \texttt{map} would use \texttt{apply}, or \texttt{map} is truncating on the shortest list.

If \texttt{mapprint} is \texttt{false}, these messages are suppressed.

\texttt{maplist} (\( f, expr_1, \ldots, expr_n \)) \hfill \text{Function}

Returns a list of the applications of \( f \) to the parts of the expressions \( expr_1, \ldots, expr_n \).
\( f \) is the name of a function, or a lambda expression.

\texttt{maplist} differs from \texttt{map} \(( f, expr_1, \ldots, expr_n \)) which returns an expression with the same main operator as \( expr_j \) has (except for simplifications and the case where \texttt{map} does an \texttt{apply}).

\texttt{prederror} \hfill \text{Option variable}

Default value: \texttt{false}

When \texttt{prederror} is \texttt{true}, an error message is displayed whenever the predicate of an \texttt{if} statement or an \texttt{is} function fails to evaluate to either \texttt{true} or \texttt{false}.

If \texttt{false}, \texttt{unknown} is returned instead in this case. The \texttt{prederror: false} mode is not supported in translated code; however, \texttt{maybe} is supported in translated code.

See also \texttt{is} and \texttt{maybe}.

\texttt{return} \(( value \)) \hfill \text{Function}

May be used to exit explicitly from a block, bringing its argument. See \texttt{block} for more information.

\texttt{scanmap} \(( f, expr \)) \hfill \text{Function}

\texttt{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 \$

\[ \%i2 \]: \( \text{scanmap(factor,exp)}; \)

\[ \%o2 \]: \( \frac{2}{(a+1)} y + x \)

Note the way in which \texttt{scanmap} applies the given function \texttt{factor} to the constituent subexpressions of \( expr \); if another form of \( expr \) is presented to \texttt{scanmap} then the result may be different. Thus, \%o2 is not recovered when \texttt{scanmap} is applied to the expanded form of \( expr \):
Chapter 41: Program Flow

(%i13) scanmap(factor, expand(exp));
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:

(%i14) expr : u*v^{a*x+b} + c$
(%i15) scanmap('f, expr);
(f(f(a) f(x)) + f(b))

(%o5) f(f(f(u) f(f(v)) + f(b)) + 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(a)*f(x)+f(b))
scanmap(f, a*x+b, bottomup) -> f(a)*f(x)+f(b) -> 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.

**while**  
Special operator

**unless**  
Special operator
See do.

**outermap** (f, a_1, ..., 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]...[i_m] has the same structure as the second list, nested list, or matrix argument, x[i_1]...[i_m][j_1]...[j_n] has the same structure as the third list, nested list, or matrix argument, and so on, where m, n, ... 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 \texttt{map}, \texttt{maplist}, and \texttt{apply}.

Examples:

Elementary examples of \texttt{outermap}. To show the argument combinations more clearly, \( F \) is left undefined.

\begin{verbatim}
(%i1) outermap(F, [a, b, c], [1, 2, 3]);
(%o1) [[F(a, 1), F(a, 2), F(a, 3)], [F(b, 1), F(b, 2), F(b, 3)],
     [F(c, 1), F(c, 2), F(c, 3)]]
(%i2) outermap(F, matrix([a, b], [c, d]), matrix([1, 2], [3, 4]));
    [     [ F(a, 1) F(a, 2) ] [ F(b, 1) F(b, 2) ] ]
    [     [     ] [     ] ]
    [     [ F(a, 3) F(a, 4) ] [ F(b, 3) F(b, 4) ] ]
(%o2) [ ]
    [     [ F(c, 1) F(c, 2) ] [ F(d, 1) F(d, 2) ] ]
    [     [     ] [     ] ]
    [     [ F(c, 3) F(c, 4) ] [ F(d, 3) F(d, 4) ] ]
(%i3) outermap(F, [a, b], x, matrix([1, 2], [3, 4]));
    [ [ F(a, 1) F(a, x, 1) ] [ F(b, 1) F(b, x, 1) ] ]
    [ [ F(a, 2) F(a, x, 2) ] [ F(b, 2) F(b, x, 2) ] ]
(%o3) [ ]
    [ [ F(c, 1) F(c, x, 1) ] [ F(d, 1) F(d, x, 1) ] ]
    [ [ F(c, 2) F(c, x, 2) ] [ F(d, 2) F(d, x, 2) ] ]
(%i4) outermap(F, [a, b], matrix([1, 2]), matrix([x], [y]));
    [ [ F(a, 1, x) ] [ F(a, 2, x) ] ]
    [ [ F(b, 1, x) ] [ F(b, 2, x) ] ]
(%o4) [ ]
    [ [ F(a, 1, y) ] [ F(a, 2, y) ] ]
    [ [ F(b, 1, y) ] [ F(b, 2, y) ] ]
(%i5) outermap("+", [a, b, c], [1, 2, 3]);
(%o5) [[a + 1, a + 2, a + 3], [b + 1, b + 2, b + 3],
     [c + 1, c + 2, c + 3]]
\end{verbatim}

A closer examination of the \texttt{outermap} 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.

\begin{verbatim}
(%i1) arg_1 : matrix([a, b], [c, d]);
    [ a b ]
(%o1) [ ]
    [ c d ]
(%i2) arg_2 : [11, 22];
(%o2) [11, 22]
(%i3) arg_3 : matrix([xx, yy]);
(%o3) [ xx yy ]
(%i4) xx_0 : outermap(lambda([x, y, z], x / y + z), arg_1,
             arg_2, arg_3);
    [ [ a a ] [ a a ] ]
    [ [ xx + -- yy + -- ], [ xx + -- yy + -- ] ]
    [ [ 11 11 ] [ 22 22 ] ]
(%o4) Col 1 =
    [ [ c c ] [ c c ] ]
\end{verbatim}
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
42 Debugging

42.1 Source Level Debugging

Maxima has a built-in source level debugger. The user can set a breakpoint at a function, and then step line by line from there. The call stack may be examined, together with the variables bound at that level.

The command :help or :h shows the list of debugger commands. (In general, commands may be abbreviated if the abbreviation is unique. If not unique, the alternatives will be listed.) Within the debugger, the user can also use any ordinary Maxima functions to examine, define, and manipulate variables and expressions.

A breakpoint is set by the :br command at the Maxima prompt. Within the debugger, the user can advance one line at a time using the :n ("next") command. The :bt ("backtrace") command shows a list of stack frames. The :r ("resume") command exits the debugger and continues with execution. These commands are demonstrated in the example below.

```maxima
(%i1) load ("/tmp/foobar.mac");
(%o1) /tmp/foobar.mac

(%i2) :br foo
Turning on debugging debugmode(true)
Bkpt 0 for foo (in /tmp/foobar.mac line 1)

(%i2) bar (2,3);
Bkpt 0:(foobar.mac 1)
/tmp/foobar.mac:1::

(dbm:1) :bt <-- :bt typed here gives a backtrace
#0: foo(y=5)(foobar.mac line 1)
#1: bar(x=2,y=3)(foobar.mac line 9)

(dbm:1) :n <-- Here type :n to advance line
(foobar.mac 2)
/tmp/foobar.mac:2::

(dbm:1) :n <-- Here type :n to advance line
(foobar.mac 3)
/tmp/foobar.mac:3::

(dbm:1) u; <-- Investigate value of u
28

(dbm:1) u: 33; <-- Change u to be 33
33

(dbm:1) :r <-- Type :r to resume the computation
```

---

**Chapter 42: Debugging**

42 Debugging

42.1 Source Level Debugging

Maxima has a built-in source level debugger. The user can set a breakpoint at a function, and then step line by line from there. The call stack may be examined, together with the variables bound at that level.

The command :help or :h shows the list of debugger commands. (In general, commands may be abbreviated if the abbreviation is unique. If not unique, the alternatives will be listed.) Within the debugger, the user can also use any ordinary Maxima functions to examine, define, and manipulate variables and expressions.

A breakpoint is set by the :br command at the Maxima prompt. Within the debugger, the user can advance one line at a time using the :n ("next") command. The :bt ("backtrace") command shows a list of stack frames. The :r ("resume") command exits the debugger and continues with execution. These commands are demonstrated in the example below.

```maxima
(%i1) load ("/tmp/foobar.mac");
(%o1) /tmp/foobar.mac

(%i2) :br foo
Turning on debugging debugmode(true)
Bkpt 0 for foo (in /tmp/foobar.mac line 1)

(%i2) bar (2,3);
Bkpt 0:(foobar.mac 1)
/tmp/foobar.mac:1::

(dbm:1) :bt <-- :bt typed here gives a backtrace
#0: foo(y=5)(foobar.mac line 1)
#1: bar(x=2,y=3)(foobar.mac line 9)

(dbm:1) :n <-- Here type :n to advance line
(foobar.mac 2)
/tmp/foobar.mac:2::

(dbm:1) :n <-- Here type :n to advance line
(foobar.mac 3)
/tmp/foobar.mac:3::

(dbm:1) u; <-- Investigate value of u
28

(dbm:1) u: 33; <-- Change u to be 33
33

(dbm:1) :r <-- Type :r to resume the computation
```
The file /tmp/foobar.mac is the following:

```lisp
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.

### 42.2 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.

```
(%s1) :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

42.3 Functions and Variables for Debugging

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.
**Option variable**

**setcheck**

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.

**Option variable**

**setcheckbreak**

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**.

**System variable**

**setval**

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**.

**Function**

**timer (f₁, ..., fₙ)**

Given functions f₁, ..., fₙ, **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. \texttt{untimer} removes functions from the list of timed functions.

\texttt{timer} quotes its arguments. \texttt{f(x) := x^2 \ g: f \ timer(g) \}} does not put \texttt{f} on the timer list.

If \texttt{trace(f)} is in effect, then \texttt{timer(f)} has no effect; \texttt{trace} and \texttt{timer} cannot both be in effect at the same time.

See also \texttt{timer_devalue}.

\begin{verbatim}
untimer (f_1, ..., f_n) Function
untimer () Function
\end{verbatim}

Given functions \( f_1, ..., f_n \), \texttt{untimer} removes each function from the timer list.

With no arguments, \texttt{untimer} removes all functions currently on the timer list.

After \texttt{untimer} \((f)\) is executed, \texttt{timer_info} \((f)\) still returns previously collected timing statistics, although \texttt{timer_info()} \((\text{with no arguments})\) does not return information about any function not currently on the timer list. \texttt{timer} \((f)\) resets all timing statistics to zero and puts \texttt{f} on the timer list again.

\begin{verbatim}
timer_devalue Option variable
\end{verbatim}

Default value: \texttt{false}

When \texttt{timer_devalue} is \texttt{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 \texttt{timer} and \texttt{timer_info}.

\begin{verbatim}
timer_info (f_1, ..., f_n) Function
timer_info () Function
\end{verbatim}

Given functions \( f_1, ..., f_n \), \texttt{timer_info} returns a matrix containing timing information for each function. With no arguments, \texttt{timer_info} returns timing information for all functions currently on the timer list.

The matrix returned by \texttt{timer_info} contains the function name, time per function call, number of function calls, total time, and \texttt{gctime}, which meant "garbage collection time" in the original Macsyma but is now always zero.

The data from which \texttt{timer_info} constructs its return value can also be obtained by the \texttt{get} function:

\begin{verbatim}
get(f, 'calls); get(f, 'runtime); get(f, 'gctime);
\end{verbatim}

See also \texttt{timer}.

\begin{verbatim}
trace (f_1, ..., f_n) Function
trace (all) Function
trace () Function
\end{verbatim}

Given functions \( f_1, ..., f_n \), \texttt{trace} instructs Maxima to print out debugging information whenever those functions are called. \texttt{trace(f)}$ \texttt{trace(g)}$ puts \texttt{f} and then \texttt{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 \((f, \text{option}_1, \ldots, \text{option}_n)\) Function

trace_options \((f)\) Function

Sets the trace options for function \( f \). Any previous options are superseded. trace_options \((f, \ldots)\) 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 \( \text{--> true} \) 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 \( \text{foo} \) is not put into effect by specifying \( \text{foo: true} \) 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:

\[
\begin{align*}
(\%i1) \quad & f(n) := \text{if equal}(n, 0) \text{ then 1 else } n \ast f(n - 1) \\
(\%i2) \quad & \text{trace}(f) \\
(\%i3) \quad & \text{trace_options}(f, \text{lisp_print}, \text{break}) \\
(\%i4) \quad & f(3);
\end{align*}
\]

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 (f_1, ..., f_n) Function
untrace () Function

Given functions f_1, ..., 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.
43 asympa

43.1 Introduction to asympa

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.

43.2 Functions and variables for asympa
44 augmented_lagrangian

44.1 Functions and Variables for augmented_lagrangian

\texttt{augmented_lagrangian_method} (\texttt{FOM, xx, C, yy}) \hspace{1cm} \text{Function}

\texttt{augmented_lagrangian_method} (\texttt{FOM, xx, C, yy, optional_args}) \hspace{1cm} \text{Function}

\texttt{augmented_lagrangian_method} ([\texttt{FOM, grad}], xx, C, yy) \hspace{1cm} \text{Function}

\texttt{augmented_lagrangian_method} ([\texttt{FOM, grad}], xx, C, yy, \texttt{optional_args}) \hspace{1cm} \text{Function}

Returns an approximate minimum of the expression \texttt{FOM} with respect to the variables \texttt{xx}, holding the constraints \texttt{C} equal to zero. \texttt{yy} is a list of initial guesses for \texttt{xx}. The method employed is the augmented Lagrangian method (see Refs [1] and [2]).

\texttt{grad}, if present, is the gradient of \texttt{FOM} with respect to \texttt{xx}, represented as a list of expressions, one for each variable in \texttt{xx}. If not present, the gradient is constructed automatically.

\texttt{FOM} and each element of \texttt{grad}, if present, must be ordinary expressions, not names of functions or lambda expressions.

\texttt{optional_args} represents additional arguments, specified as \texttt{symbol = value}. The optional arguments recognized are:

\begin{itemize}
  \item \texttt{niter} \hspace{1cm} Number of iterations of the augmented Lagrangian algorithm
  \item \texttt{lbfgs_tolerance} \hspace{1cm} Tolerance supplied to LBFGS
  \item \texttt{iprints} \hspace{1cm} IPRINT parameter (a list of two integers which controls verbosity) supplied to LBFGS
  \item \texttt{lambda} \hspace{1cm} Initial value of \texttt{lambda} to be used for calculating the augmented Lagrangian
\end{itemize}

This implementation minimizes the augmented Lagrangian by applying the limited-memory BFGS (LBFGS) algorithm, which is a quasi-Newton algorithm.

\texttt{load(augmented_lagrangian)} loads this function.

See also \texttt{lbfgs}.

References:


Examples:

\begin{verbatim}
(%i1) load (lbfgs);
(%o1) /maxima/share/lbfgs/lbfgs.mac
(%i2) load (augmented_lagrangian);
(%o2) /maxima/share/contrib/augmented_lagrangian.mac
(%i3) FOM: x^2 + 2*y^2;
\end{verbatim}
%o3) \( 2 y + x \)
(%i4) xx: [x, y];
(%o4) [x, y]
(%i5) C: [x + y - 1];
(%o5) [y + x - 1]
(%i6) yy: [1, 1];
(%o6) [1, 1]
(%i7) augmented_lagrangian_method(FOM, xx, C, yy, iprint=[-1,0]);
(%o7) \[x = 0.666659841080025, y = .3333402724554462, \]
\( \%\lambda = [- 1.333337940892543] \)

Same example as before, but this time the gradient is supplied as an argument.

(%i1) load (lbfgs)$
(%i2) load (augmented_lagrangian)$
(%i3) FOM: x^2 + 2*y^2;
 2 2
(%o3) 2 y + x
(%i4) FOM: x^2 + 2*y^2;
 2 2
(%o4) 2 y + x
(%i5) xx: [x, y];
(%o5) [x, y]
(%i6) grad : [2*x, 4*y];
(%o6) [2 x, 4 y]
(%i7) C: [x + y - 1];
(%o7) [y + x - 1]
(%i8) yy: [1, 1];
(%o8) [1, 1]
(%i9) augmented_lagrangian_method ([FOM, grad], xx, C, yy, iprint = [-1, 0]);
(%o9) \[x = 0.666659841080025, y = .3333402724554462, \]
\( \%\lambda = [- 1.333337940892543] \)
45 bode

45.1 Functions and Variables for bode

bode_gain \( (H, \text{range}, \ldots\text{plot_opts}\ldots) \)

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
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
Chapter 46: cobyla

46 cobyla

46.1 Introduction to cobyla

fmin_cobyla is a Common Lisp translation (via f2cl) of the Fortran constrained optimization routine COBYLA by Powell[1][2][3].

COBYLA minimizes an objective function \( F(X) \) subject to \( M \) inequality constraints of the form \( g(X) \geq 0 \) 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 \( g(X)\geq0 \) and \( -g(X)\geq0 \). 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 \( \text{RHO} \) controls the size of the simplex and it is reduced automatically from \( \text{RHOBEG} \) to \( \text{RHOEND} \). For each \( \text{RHO} \) the subroutine tries to achieve a good vector of variables for the current size, and then \( \text{RHO} \) is reduced until the value \( \text{RHOEND} \) is reached. Therefore \( \text{RHOBEG} \) and \( \text{RHOEND} \) 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

46.2 Functions and Variables for cobyla

fmin_cobyla \((F, X, Y)\)

fmin_cobyla \((F, X, Y, \text{optional_args})\)

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.

\text{optional_args} represents additional arguments, specified as \text{symbol} = \text{value}. The optional arguments recognized are:
constraints
List of inequality and equality constraints that must be satisfied by $X$. The inequality constraints must be actual inequalities of the form $g(X) \geq h(X)$ or $g(X) \leq h(X)$. The equality constraints must be of the form $g(X) = h(X)$.

rhobeg
Initial value of the internal RHO variable which controls the size of simplex. (Defaults to 1.0)

rhoend
The desired final value rho parameter. It is approximately the accuracy in the variables. (Defaults to 1d-6.)

iprint
Verbose output level. (Defaults to 0)
  0 - No output
  1 - Summary at the end of the calculation
  2 - Each new value of RHO and SIGMA is printed, including the vector of variables, some function information when RHO is reduced.
  3 - Like 2, but information is printed when F(X) is computed.

maxfun
The maximum number of function evaluations. (Defaults to 1000).

On return, a vector is given:
1. The value of the variables giving the minimum. This is a list of elements of the form $\text{var} = \text{value}$ for each of the variables listed in $X$.
2. The minimized function value
3. The number of function evaluations.
4. Return code with the following meanings
   1. 0 - No errors.
   2. 1 - Limit on maximum number of function evaluations reached.
   3. 2 - Rounding errors inhibiting progress.

load(fmin_cobyla)
loads this function.

46.3 Examples for cobyla
Minimize $x1*x2$ with $1-x1^2-x2^2 >= 0$. The theoretical solution is $x1 = 1/\sqrt{2}$, $x2 = -1/\sqrt{2}$.

(%i1) load(fmin_cobyla)$

(%i2) fmin_cobyla(x1*x2, [x1, x2], [1,1], constraints = [x1^2+x2^2<=1], iprint=1);
Normal return from subroutine COBYLA

NFVALS = 66  F = -5.000000E-01  MAXCV = 1.999845E-12
X = 7.071058E-01 -7.071077E-01

(%i2) [[x1 = 0.70710584934848, x2 = -0.7071077130248], -0.49999999999926,]
   [[[-1.999955756559757e-12], []], 66]

There are additional examples in the share/cobyla/ex directory.
Chapter 47: contrib_ode

47 contrib_ode

47.1 Introduction to contrib_ode

Maxima's ordinary differential equation (ODE) solver ode2 solves elementary linear ODEs of first and second order. The function contrib_ode extends ode2 with additional methods for linear and non-linear first order ODEs and linear homogeneous second order ODEs. The code is still under development and the calling sequence may change in future releases. Once the code has stabilized it may be moved from the contrib directory and integrated into Maxima.

This package must be loaded with the command load('contrib_ode) before use.

The calling convention for contrib_ode is identical to ode2. It takes three arguments: an ODE (only the left hand side need be given if the right hand side is 0), the dependent variable, and the independent variable. When successful, it returns a list of solutions.

The form of the solution differs from ode2. As non-linear equations can have multiple solutions, contrib_ode returns a list of solutions. Each solution can have a number of forms:

- an explicit solution for the dependent variable,
- an implicit solution for the dependent variable,
- a parametric solution in terms of variable \( \%t \), or
- a tranformation into another ODE in variable \( \%u \).

\( \%c \) is used to represent the constant of integration for first order equations. \( \%k1 \) and \( \%k2 \) are the constants for second order equations. If contrib_ode cannot obtain a solution for whatever reason, it returns false, after perhaps printing out an error message.

It is necessary to return a list of solutions, as even first order non-linear ODEs can have multiple solutions. For example:

\begin{verbatim}
(%i1) load('contrib_ode)$
(%i2) eqn:x*'diff(y,x)^2-(1+x*y)*'diff(y,x)+y=0;
(%o2) x (--) - (x y + 1) -- + y = 0
    dx           dx
(%i3) contrib_ode(eqn,y,x);
(%o3) [y = log(x) + \%c, y = \%c \%e ]

(%i4) method;
(%o4) factor
\end{verbatim}

Nonlinear ODEs can have singular solutions without constants of integration, as in the second solution of the following example:

\begin{verbatim}
(%i1) load('contrib_ode)$
(%i2) eqn:x*'diff(y,x)^2-(1+x*y)*'diff(y,x)+y=0;
(%o2) x (--) - (x y + 1) -- + y = 0
    dx           dx
(%i3) contrib_ode(eqn,y,x);
(%o3) [y = log(x) + \%c, y = \%c \%e ]

(%i4) method;
(%o4) factor
\end{verbatim}
(%i2) eqn:'diff(y,x)^2+x*'diff(y,x)-y=0;
   2
(%o2) (--) + x -- - y = 0
    dx    dx

(%i3) contrib_ode(eqn,y,x);
  2
(%o3) [y = %c x + %c , y = --]
      4

(%i4) method;
(%o4) clairault

The following ODE has two parametric solutions in terms of the dummy variable %t. In this case the parametric solutions can be manipulated to give explicit solutions.

(%i1) load('contrib_ode)$
(%i2) eqn:'diff(y,x)=(x+y)^2;
   2
(%o2) -- = (y + x)
    dx

(%i3) contrib_ode(eqn,y,x);
(%o3) [[x = %c - atan(sqrt(%t)), y = - x - sqrt(%t)],
      [x = atan(sqrt(%t)) + %c, y = sqrt(%t) - x]]
(%i4) method;  
(%o4) clairault

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;
   2  2
(%o3) x -- - x sqrt(y + x ) - y
    dx

(%i4) contrib_ode(eqn,y,x);
(%o4) [x - asinh(=) = %c]
      x

(%i5) method;  

Chapter 47: contrib_ode

47.2 Functions and Variables for contrib_ode

contrib_ode (eqn, y, x) Function
Returns a list of solutions of the ODE eqn with independent variable x and dependent variable y.

odelin (eqn, y, x) Function
odelin solves linear homogeneous ODEs of first and second order with independent variable x and dependent variable y. It returns a fundamental solution set of the ODE.

For second order ODEs, odelin uses a method, due to Bronstein and Lafaille, that searches for solutions in terms of given special functions.

(%i1) load('contrib_ode);
(%i2) odelin(x*(x+1)*'diff(y,x,2)+(x+5)*'diff(y,x,1)+(-4)*y,y,x);
...trying factor method
...solving 7 equations in 4 variables
...trying the Bessel solver
...solving 1 equations in 2 variables
...trying the F1 solver
...solving 1 equations in 3 variables
...trying the spheroidal wave solver
...solving 1 equations in 4 variables
...trying the square root Bessel solver
...solving 1 equations in 2 variables
...trying the 2F1 solver
...solving 9 equations in 5 variables

\[
\text{gauss}_a(-6, -2, -3, -x) \text{ gauss}_b(-6, -2, -3, -x)
\]

\[
\{ \frac{x}{4}, \frac{x}{4} \}
\]

\[
\text{ode_check (eqn, soln)}
\]

Returns the value of ODE \( \text{eqn} \) after substituting a possible solution \( \text{soln} \). The value is equivalent to zero if \( \text{soln} \) is a solution of \( \text{eqn} \).

\[(\%i1) \text{ load('contrib_ode)}\$
\]

\%(i2) eqn:'diff(y,x,2)+(a*x+b)*y;

\[
\frac{d y}{d x} + (a x + b) y
\]

\%(i3) ans:[y = \text{bessel}_y(1/3,2*(a*x+b)^{3/2}/(3*a))*%k2*sqrt(a*x+b)
+\text{bessel}_j(1/3,2*(a*x+b)^{3/2}/(3*a))*%k1*sqrt(a*x+b)];

\%(i4) ode_check(eqn,ans[1]);

\%(o4) 0

\text{method} System variable

The variable \text{method} is set to the successful solution method.

\%c Variable

\%c is the integration constant for first order ODEs.
Variable

%k1

%k1 is the first integration constant for second order ODEs.

Variable

%k2

%k2 is the second integration constant for second order ODEs.

Function

gauss_a (a, b, c, x)

gauss_a(a, b, c, x) and gauss_b(a, b, c, x) are 2F1 geometric functions. They represent any two independent solutions of the hypergeometric differential equation $x(1-x) \frac{d^2y}{dx^2} + [c-(a+b+1)x] \frac{dy}{dx} - aby = 0$ (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.

Function

gauss_b (a, b, c, x)

See gauss_a.

Function

dgauss_a (a, b, c, x)

The derivative with respect to $x$ of gauss_a(a, b, c, x).

Function

dgauss_b (a, b, c, x)

The derivative with respect to $x$ of gauss_b(a, b, c, x).

Function

kummer_m (a, b, x)

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.

Function

kummer_u (a, b, x)

Kummer’s U function, as defined in Abramowitz and Stegun, Handbook of Mathematical Functions, Section 13.1.3.

See kummer_m.

Function

dkummer_m (a, b, x)

The derivative with respect to $x$ of kummer_m(a, b, x).

Function

dkummer_u (a, b, x)

The derivative with respect to $x$ of kummer_u(a, b, x).
47.3 Possible improvements to contrib.ode

These routines are work in progress. I still need to:

- Extend the FACTOR method \texttt{ode1\_factor} to work for multiple roots.
- Extend the FACTOR method \texttt{ode1\_factor} to attempt to solve higher order factors. At present it only attempts to solve linear factors.
- Fix the LAGRANGE routine \texttt{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 \texttt{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.

47.4 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 \texttt{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 \texttt{ode1\_lie} are overly complex and impossible to check.
- There are some crashes.

47.5 References for contrib.ode

2. G. M. Murphy, Ordinary Differential Equations and Their Solutions, Van Nostrand, New York, 1960
48 descriptive

48.1 Introduction to descriptive

Package descriptive contains a set of functions for making descriptive statistical computations and graphing. Together with the source code there are three data sets in your Maxima tree: pidigits.data, wind.data and biomed.data.

Any statistics manual can be used as a reference to the functions in package descriptive.

For comments, bugs or suggestions, please contact me at 'mario AT edu DOT xunta DOT es'.

Here is a simple example on how the descriptive functions in descriptive do they work, depending on the nature of their arguments, lists or matrices,

```
(%i1) load (descriptive)$
(%i2) /* univariate sample */ mean ([a, b, c]);
   c + b + a
---
  3
(%o2)

(%i3) matrix ([a, b], [c, d], [e, f]);
   [ a   b ]
   [       ]
   [ c   d ]
   [       ]
   [ e   f ]

(%i4) /* multivariate sample */ mean (%);
   e + c + a  f + d + b
----------  ----------
  3          3
(%o4)
```

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
----------  -----
  3          2
(%o2)
```

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]);
(%o1)

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 \texttt{pidigits.data} contains the first 100 digits of number \%pi: 

\begin{verbatim}
3
1
4
1
5
9
2
6
5
3 ...
\end{verbatim}

In order to load these digits in Maxima,

\begin{verbatim}
(%i1) s1 : read_list (file_search ("pidigits.data"))$
(%i2) length (s1);
(%o2) 100
\end{verbatim}

On the other hand, file \texttt{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) \textit{Space-time Modelling with Long-memory Dependence: Assessing Ireland’s Wind Power Resource, with Discussion}. Applied Statistics 38, 1-50). This loads the data:

\begin{verbatim}
(%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]
\end{verbatim}

Some samples contain non numeric data. As an example, file \texttt{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,

\begin{verbatim}
(%i1) s3 : read_matrix (file_search ("biomed.data"))$
\end{verbatim}
Chapter 48: descriptive

48.2 Functions and Variables for data manipulation

**continuous_freq** *(list)*

The argument of `continuous_freq` must be a list of numbers, which will be then grouped in intervals and counted how many of them belong to each group. Optionally, function `continuous_freq` admits a second argument indicating the number of classes, 10 is default.

```
(%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]]
```

The first list contains the interval limits and the second the corresponding counts: there are 16 digits inside the interval [0, 1.8], that is 0’s and 1’s, 24 digits in (1.8, 3.6], that is 2’s and 3’s, and so on.

**discrete_freq** *(list)*

Counts absolute frequencies in discrete samples, both numeric and categorical. Its unique argument is a list,

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) discrete_freq (s1);
(%o3) [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
     [8, 8, 12, 12, 10, 8, 9, 8, 12, 13]]
```

The first list gives the sample values and the second their absolute frequencies. Commands `? col` and `? transpose` should help you to understand the last input.

**subsample** *(data_matrix, predicate_function)*

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 $v[i]$.

\begin{verbatim}
(%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 ]
  [                     ]
  [ 20.25 21.46 19.95 27.71 23.38 ]
  [                     ]
  [ 18.79 18.96 14.46 26.38 21.84 ]
(%o3) [                     ]

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.

\begin{verbatim}
(%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 ]
  [                     ]
  [ 16.92 13.21 21.21 ]
  [                     ]
  [ 17.25 18.46 23.87 ]
(%o4) [                     ]
\end{verbatim}

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.

\begin{verbatim}
(%i1) load (descriptive)$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
\end{verbatim}
Chapter 48: descriptive

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);
```

This is the multivariate mean of \( s3 \),

```
(%i1) load (descriptive)$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i3) mean (s3);
```

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 \texttt{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 \texttt{NA} symbols, although this deserves some loss of information.

```lisp
(%i1) load (descriptive)$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i4) mean (subsample (s3, g, 3, 4, 5, 6));
(%o4) [79.4923076923077, 86.2032967032967, 16.93186813186813,
     2514
     ----]
     13
```

48.3 Functions and Variables for descriptive statistics

\textbf{mean} \texttt{(list)} \hspace{6cm} \textbf{mean} \texttt{(matrix)}

This is the sample mean, defined as

\[ \bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \]

Example:

```lisp
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) mean (s1);
(%o3) 471
     ---
     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]
```

\textbf{var} \texttt{(list)} \hspace{6cm} \textbf{var} \texttt{(matrix)}

This is the sample variance, defined as

\[ \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2 \]

Example:

```lisp
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) var (s1), numer;
(%o3) 8.425899999999999
```

See also function \texttt{var1}. 

\textbf{mean} \texttt{(list)} \hspace{6cm} \textbf{mean} \texttt{(matrix)}
Chapter 48: descriptive

Function \texttt{var1} (\texttt{list})

This is the sample variance, defined as
\[
\frac{1}{n - 1} \sum_{i=1}^{n} (x_i - \bar{x})^2
\]

Example:

```lisp
(%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 \texttt{var}.

Function \texttt{std} (\texttt{list})

This is the square root of function \texttt{var}, the variance with denominator \(n\).

Example:

```lisp
(%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 \texttt{var} and \texttt{std1}.

Function \texttt{std1} (\texttt{list})

This is the square root of function \texttt{var1}, the variance with denominator \(n - 1\).

Example:

```lisp
(%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, 4.96687617451963]
```

See also functions \texttt{var1} and \texttt{std}. 

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 (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 (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) [.4192246091090204, .3829365309260502, 0.363779605385983, 
        .3627381836021478, .3346021393989506]
```
See also functions std and mean.
**smin** (list) Function
**smin** (matrix) Function

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** (list) Function
**smax** (matrix) Function

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** (list) Function
**range** (matrix) Function

The range is the difference between the extreme values.

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) range (s1);
(%o3) 9
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) range (s2);
(%o5) [19.67, 20.96, 17.37, 24.38, 22.46]
```

**quantile** (list, p) Function
**quantile** (matrix, p) Function

This is the p-quantile, with p a number in \([0, 1]\), of the sample list. Although there are several definitions for the sample quantile (Hyndman, R. J., Fan, Y. (1996) Sample quantiles in statistical packages. American Statistician, 50, 361-365), the one based on linear interpolation is implemented in package `descriptive`.

Example:
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) /* 1st and 3rd quartiles */
   [quantile (s1, 1/4), quantile (s1, 3/4)], numer;
   [2.0, 7.25]
(%o3)  
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) quantile (s2, 1/4);
   [7.2575, 7.477500000000001, 7.82, 11.28, 11.48]

median (list) Function
median (matrix) Function

Once the sample is ordered, if the sample size is odd the median is the central value, otherwise it is the mean of the two central values.

Example:
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) median (s1);
   9
(%o3)  
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) median (s2);
   [10.06, 9.855, 10.73, 15.48, 14.105]

The median is the 1/2-quantile.
See also function quantile.

qrange (list) Function
qrange (matrix) Function

The interquartile range is the difference between the third and first quartiles, quantile(list,3/4) - quantile(list,1/4),

(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) qrange (s1);
   21
(%o3)  
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) qrange (s2);
   [5.385, 5.57249999999998, 6.02250000000001, 8.72999999999999, 6.64999999999999]

See also function quantile.

mean_deviation (list) Function
mean_deviation (matrix) Function

The mean deviation, defined as
\[
\frac{1}{n} \sum_{i=1}^{n} |x_i - \bar{x}|
\]
Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) mean_deviation (s1);
    51
(%o3) --
    20
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) mean_deviation (s2);
    3.28796, 3.07534, 3.23907, 4.71566, 4.02854
(%o5) [3.28796, 3.07534, 3.23907, 4.71566, 4.02854]
```

See also function `mean`.

**median_deviation (list)**

**median_deviation (matrix)**

The median deviation, defined as

\[
\frac{1}{n} \sum_{i=1}^{n} |x_i - \text{med}|
\]

where `med` is the median of `list`.

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) median_deviation (s1);
    5
(%o3) -
    2
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) median_deviation (s2);
    2.75, 2.755, 3.08, 4.315, 3.31
(%o5) [2.75, 2.755, 3.08, 4.315, 3.31]
```

See also function `mean`.

**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;
    3.90186
(%o3) 3.901858027632205
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) harmonic_mean (s2);
    6.948015590052786, 7.391967752360356, 9.05658197151745,
    13.44199028193692, 13.81439145898509
(%o5) [6.948015590052786, 7.391967752360356, 9.05658197151745,
    13.44199028193692, 13.81439145898509]
```

See also functions `mean` and `geometric_mean`.
geometric_mean (list) Function
geometric_mean (matrix) Function

The geometric mean, defined as

$$\left( \prod_{i=1}^{n} x_i \right)^{\frac{1}{n}}$$

Example:

```plaintext
(%i1) load (descriptive)$
(%i2) y : [5, 7, 2, 5, 9, 5, 6, 4, 9, 2, 4, 2, 5]$%
(%i3) geometric_mean (y), numer;
(%o3) 4.454845412337012
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) geometric_mean (s2);%
(%o5) [8.82476274347979, 9.22652604739361, 10.0442675714889, 14.61274126349021, 13.96184163444275]
```

See also functions mean and harmonic_mean.

kurtosis (list) Function
kurtosis (matrix) Function

The kurtosis coefficient, defined as

$$\frac{1}{ns^4} \sum_{i=1}^{n} (x_i - \bar{x})^4 - 3$$

Example:

```plaintext
(%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) [-0.2715445622195385, 0.119998784429451, -0.4275233490482861, -0.6405361979019522, -0.4952382132352935]
```

See also functions mean, var and skewness.

skewness (list) Function
skewness (matrix) Function

The skewness coefficient, defined as

$$\frac{1}{ns^3} \sum_{i=1}^{n} (x_i - \bar{x})^3$$

Example:
(\%i1) load (descriptive)$
(\%i2) s1 : read_list (file_search ("pidigits.data"))$
(\%i3) skewness (s1), numer;
(\%o3) .009196180476450424
(\%i4) s2 : read_matrix (file_search ("wind.data"))$
(\%i5) skewness (s2);
(\%o5) [.1580509020000978, .2926379232061854, .09242174416107717, .2059984348148687, .2142520248890831]

See also functions mean, var and kurtosis.

\textbf{pearson\_skewness} (list)

\textbf{pearson\_skewness} (matrix)

Pearson’s skewness coefficient, defined as

\[3 (\bar{x} - \text{med})\]

\[s\]

where \text{med} is the median of \text{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.

\textbf{quartile\_skewness} (list)

\textbf{quartile\_skewness} (matrix)

The quartile skewness coefficient, defined as

\[c_{3_4} - 2c_{1_2} + c_{1_4}\]

\[c_{3_4} - c_{1_4}\]

where \(c_p\) is the \(p\)-quantile of sample \text{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.
Function \texttt{cov} \hspace{1cm} (\textit{matrix})

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:

\begin{verbatim}
(%i1) load (descriptive)$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) fpprintprec : 7$ /* change precision for pretty output */
(%i4) cov (s2);
\end{verbatim}

\begin{verbatim}
[ ]
[ ]
(%o4) [ 14.37217 13.30448 15.47573 17.32544 16.18171 ]
[ ]
[ 19.39624 15.15834 17.32544 32.17651 20.44685 ]
[ ]
\end{verbatim}

See also function \texttt{cov1}.

Function \texttt{cov1} \hspace{1cm} (\textit{matrix})

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:

\begin{verbatim}
(%i1) load (descriptive)$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) fpprintprec : 7$ /* change precision for pretty output */
(%i4) cov1 (s2);
\end{verbatim}

\begin{verbatim}
[ ]
[ 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 ]
\end{verbatim}

See also function \texttt{cov}.
**global_variances** (*matrix*)

**global_variances** (*matrix, logical_value*)

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 standard deviation**: `determinant(S_1)^(1/(2*p))`.

where \( p \) is the dimension of the multivariate random variable and \( S_1 \) the covariance matrix returned by `cov1`.

Example:

```maxima
(%i1) load (descriptive)$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) global_variances (s2);
(%o3) [105.338342060606, 21.06766841212119, 12874.34690469686, 113.4651792608501, 6.636590811800795, 2.576158149609762]
```

Function `global_variances` has an optional logical argument: `global_variances(x,true)` tells Maxima that \( x \) is the data matrix, making the same as `global_variances(x)`. On the other hand, `global_variances(x,false)` means that \( x \) is not the data matrix, but the covariance matrix, avoiding its recalculation,

```maxima
(%i1) load (descriptive)$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) s : cov1 (s2)$
(%i4) global_variances (s, false);
(%o4) [105.338342060606, 21.06766841212119, 12874.34690469686, 113.4651792608501, 6.636590811800795, 2.576158149609762]
```

See also `cov` and `cov1`.

**cor** (*matrix*)

**cor** (*matrix, logical_value*)

The correlation matrix of the multivariate sample.

Example:

```maxima
(%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  .782502 ]
[                ]
(%o4) [ .8803515  .8735834  1.0  .7764065  .8323358 ]
[                ]
[ .8239624  .6902622  .7764065  1.0  .7293848 ]
[                ]
[ .7519506  .782502  .8323358  .7293848  1.0 ]

Function cor has an optional logical argument: cor(x,true) tells Maxima that x
is the data matrix, making the same as cor(x). On the other hand, cor(x,false)
means that x is not the data matrix, but the covariance matrix, avoiding its recalcu-
lation,

(%i11) load (descriptive)$
(%i12) fpprintprec : 7 $  
(%i13) s2 : read_matrix (file_search ("wind.data"))$
(%i14) s : cov1 (s2)$
(%i15) cor (s, false); /* this is faster */
[ 1.0  .8476339  .8803515  .8239624  .7519506 ]
[                ]
[ .8476339  1.0  .8735834  .6902622  .782502 ]
[                ]
(%o15) [ .8803515  .8735834  1.0  .7764065  .8323358 ]
[                ]
[ .8239624  .6902622  .7764065  1.0  .7293848 ]
[                ]
[ .7519506  .782502  .8323358  .7293848  1.0 ]

See also cov and cov1.

**list_correlations** (matrix) Function
**list_correlations** (matrix, logical_value) Function

Function list_correlations returns a list of correlation measures:

- **precision matrix**: the inverse of the covariance matrix $S_1$,
  
  $$S_1^{-1} = (s_{ij}^{ij})_{i,j=1,2,...,p}$$

- **multiple correlation vector**: $(R_1^2, R_2^2, ..., R_p^2)$, with
  
  $$R_i^2 = 1 - \frac{1}{s_{ii}s_{ii}}$$

  being an indicator of the goodness of fit of the linear multivariate regression
  model on $X_i$ when the rest of variables are used as regressors.

- **partial correlation matrix**: with element $(i, j)$ being
  
  $$r_{ij,\text{rest}} = -\frac{s_{ij}}{\sqrt{s_{ii}s_{jj}}}$$

Example:
Chapter 48: descriptive

```lisp
(%i11) load (descriptive)$
(%i12) s2 : read_matrix (file_search ("wind.data"))$
(%i13) z : list_correlations (s2)$
(%i14) fprintprec : 5$ /* for pretty output */
(%i15) z[1]; /* precision matrix */
[ [ .38486 - .13856 - .15626 - .10239 .031179 ]
  [ ]
  [ - .13856 .34107 - .15233 .038447 - .052842 ]
  [ ]
  [ - .15626 - .15233 .47296 - .024816 - .10054 ]
  [ ]
  [ - .10239 .038447 - .024816 .10937 - .034033 ]
  [ ]
  [ .031179 - .052842 - .10054 - .034033 .14834 ]
(%o5) [ - .13856 .34107 - .15233 .038447 - .052842 ]
(%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 ]
  [ ]
  [ .36627 .37927 - 1.0 .10911 .37956 ]
  [ ]
  [ .49908 - .19907 .10911 - 1.0 .26719 ]
  [ ]
  [ - .13049 .23492 .37956 .26719 - 1.0 ]
(%o7) [ - 1.0 .38244 .36627 .49908 - .13049 ]
(%i8) z[3]; /* partial correlation matrix */
```

Function `list_correlations` also has an optional logical argument: `list_correlations(x,true)` tells Maxima that x is the data matrix, making the same as `list_correlations(x)`. On the other hand, `list_correlations(x,false)` means that x is not the data matrix, but the covariance matrix, avoiding its recalculation.

See also `cov` and `cov1`.

### 48.4 Functions and Variables for statistical graphs

- **histogram** (`list`) Function
- **histogram** (`list, option_1, option_2, ...`) Function
- **histogram** (`one_column_matrix`) Function
- **histogram** (`one_column_matrix, option_1, option_2, ...`) Function
- **histogram** (`one_row_matrix`) Function
- **histogram** (`one_row_matrix, option_1, option_2, ...`) Function

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:
- Those defined in the `draw` package. See also `bars` and `barsplot`.
- `nclasses`: number of classes of the histogram (10 by default).
See also `discrete_freq` and `continuous_freq` to count data, and `bars` and `barsplot` to display bar graphs.

Examples:

A simple histogram 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)$
```

### `scatterplot`

- **Function**
- **Function**
- **Function**
- **Function**

Plots scatter diagrams both for univariate (`list`) and multivariate (`matrix`) samples.

Available options are:
- Those defined in the `draw` package.
- `nclasses`: number of classes of the histogram (10 by default).

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,
    terminal = eps,
    eps_width = 10,
    eps_height = 2)$
```

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.
Five dimensional scatter plot, with five classes histograms.

For plotting isolated or line-joined points in two and three dimensions, see `points`. For histogram related options, see `bars`.

See also `histogram`.

**barsplot**(data1, data2, ..., option1, option2, ...)

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:

- Those defined in the `draw` package.
- `box_width`: relative width of rectangles (3/4 by default). This value must be in the range [0,1].
- `groups_gap`: a positive integer number representing the gap between two consecutive groups of bars. Default value is 1.
- `bars_colors`: a list of colors for multiple samples. Defaults to the empty list `[]`. When there are more samples than specified colors, the extra necessary colors are chosen at random. See `color` to learn more about them.
- `relative_frequencies`: if `false`, absolute frequencies are used; if `true`, tics on the y-axis are relative frequencies. Default value is `false`.
- `ordering`: possible values are `orderlessp` or `ordergreatp`, indicating how statistical outcomes should be ordered on the x-axis. Default value is `orderlessp`.
- `sample_keys`: a list with the strings to be used in the legend. Defaults to the empty list `[]`. When the list length is other than 0 or the number of samples, an error message is returned.

Examples:

Univariate sample in matrix form. Absolute frequencies.

```lisp
(%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],
   relative_frequencies = true,
   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)$
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) draw_compound : false$
(%i5) bp1 : 
   barsplot(
   l1,
   box_width = 1,
   fill_density = 0.5,
   bars_colors = [blue],
   relative_frequencies = true)$
(%i6) bp2 : 
   barsplot(
   l2,
   box_width = 1,
   fill_density = 0.5,
   bars_colors = [red],
   relative_frequencies = true)$
Chapter 48: descriptive

(%i7) draw(apply(gr2d,bp1),
     apply(gr2d,bp2))$

For bars diagrams related options, see bars of package draw. See also functions histogram and piechart.

piechart (list) Function
piechart (list, option_1, option_2, ...) Function
piechart (one-column_matrix) Function
piechart (one-column_matrix, option_1, option_2, ...) Function
piechart (one-row_matrix) Function
piechart (one-row_matrix, option_1, option_2, ...) Function

Similar to barsplot, but plots sectors instead of rectangles.

Available options are:

- Those defined in the draw package.
- pie_center: diagram’s center ([0,0] by default).
- pie_radius: diagram’s radius (1 by default).

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],
    axis_top = false,
    axis_right = false,
    axis_left = false,
    axis_bottom = false,
    xtics = none,
    ytics = none,
    title = "Digit frequencies in pi")$

See also function barsplot.

boxplot (data) Function
boxplot (data, option_1, option_2, ...) Function

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.

Available options are:

- Those defined in the draw package.
- box_width: relative width of boxes (3/4 by default). This value must be in the range [0,1].
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)$
```
49 diag

49.1 Functions and Variables for diag

**diag** (*lm*)

Function

Constructs a square matrix with the matrices of *lm* in the diagonal. *lm* is a list of matrices or scalars.

Example:

```
(%i1) load("diag")

(%i2) a1:matrix([1,2,3],[0,4,5],[0,0,6])

(%i3) a2:matrix([1,1],[1,0])

(%i4) diag([a1,x,a2]);

[ 1 2 3 0 0 0 ]
[ ]
[ 0 4 5 0 0 0 ]
[ ]
[ 0 0 6 0 0 0 ]
[ ]

(%o4)
```

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)

(%i3) JF(3,2);

[ 3 1 ]
```
Function \texttt{jordan} \texttt{(mat)}

Returns the Jordan form of matrix \texttt{mat}, but codified in a Maxima list. To get the corresponding matrix, call function \texttt{dispJordan} using as argument the output of \texttt{jordan}.

Example:

\begin{verbatim}
(%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],
[2,0,2,0,0,0,0,0],
[-7,2,0,2,0,0,0,0],
[9,0,-2,0,1,2,0,0],
[-34,7,1,-2,-1,1,2,0],
[145,-17,-16,3,9,-2,0,3])$

(%i34) jordan(a);
(%o34) \[
[2, 3, 3, 1], \[3, 1]\]
\end{verbatim}

To use this function write first \texttt{load("diag").}

See also \texttt{dispJordan} and \texttt{minimalPoly}.

Function \texttt{dispJordan} \texttt{(l)}

Returns the Jordan matrix associated to the codification given by the Maxima list \texttt{l}, which is the output given by function \texttt{jordan}.

Example:

\begin{verbatim}
(%i1) load("diag")$
(%i2) b1:matrix([0,0,1,1,1],
[0,0,0,0,2,1,0,0],
[0,0,0,0,2,0,0,0],
[0,0,0,0,2,0,0,0],
[0,0,0,0,0,0,0,3])$

(%o5) \[
[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 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 ]
\end{verbatim}

To use this function write first \texttt{load("diag").} See also \texttt{dispJordan} and \texttt{minimalPoly}. 
Chapter 49: diag

\[ [0,0,0,1,1], \]
\[ [0,0,0,0,1], \]
\[ [0,0,0,0,0], \]
\[ [0,0,0,0,0] \]$

(%i3) jordan(b1);$
(%o3) \[ [0, 3, 2] \]$

(%i4) dispJordan(%);$
\[
\begin{bmatrix}
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 
\end{bmatrix}
\]$

(%o4) \[ [0, 0, 0, 0, 0] \]$

To use this function write first load("diag"). See also jordan and minimalPoly.

**minimalPoly (l)**

Function

Returns the minimal polynomial associated to the codification given by the Maxima list \( l \), which is the output given by function jordan.

Example:

(%i1) load("diag")$

(%i2) a:matrix([2,1,2,0],
[-2,2,1,2],
[-2,-1,-1,1],
[3,1,2,-1])$

(%i3) jordan(a);$
(%o3) \[ [- 1, 1], [1, 3] \]$

(%i4) minimalPoly(%);$
\[ (x - 1) (x + 1) \]$

To use this function write first load("diag"). See also jordan and dispJordan.

**ModeMatrix (A,l)**

Function

Returns the matrix \( M \) such that \( Mm1.A.M = J \), where \( J \) is the Jordan form of \( A \). The Maxima list \( l \) is the codified form of the Jordan form as returned by function jordan.

Example:

(%i1) load("diag")$

(%i2) a:matrix([2,1,2,0],
[-2,2,1,2],
[-2,-1,-1,1],
[3,1,2,-1])$

(%i3) jordan(a);$
(%o3) \[ [- 1, 1], [1, 3] \]$

(%i4) minimalPoly(%);$
\[ (x - 1) (x + 1) \]$

To use this function write first load("diag"). See also jordan and dispJordan.
624 Maxima 5.24.0 Manual

(%i3) jordan(a);
(%o3) [[- 1, 1], [1, 3]]

(%i4) M: ModeMatrix(a,%);
[ 1 - 1 1 1 ]
[ ]
[ 1     ]
[ - - 1 0 0 ]
[ 9     ]
[ ]
[ ]
(%o4)
[ 13 ]
[ - -- 1 - 1 0 ]
[ 9     ]
[ ]
[ ]
[ 17 ]
[ -- - 1 1 1 ]
[ 9     ]

(%i5) is( (M^^-1).a.M = dispJordan(%o3) );
(%o5) true

Note that dispJordan(%o3) is the Jordan form of matrix a.
To use this function write first load("diag"). See also jordan and dispJordan.

mat_function (f,m) Function
Returns f(mat), where f is an analytic function and mat a matrix. This computation
is based on Cauchy's integral formula, which states that if f(x) is analytic and
mat = diag([JF(m1,n1),...,JF(mk,nk)]),
then
f(mat) = ModeMatrix*diag([f(JF(m1,n1)), ..., f(JF(mk,nk))])
*ModeMatrix^^(-1)

Note that there are about 6 or 8 other methods for this calculation.
Some examples follow.
Example 1:
(%i11) load("diag")$
(%i12) b2:matrix([0,1,0], [0,0,1], [-1,-3,-3])$
(%i13) mat_function(exp,t*b2);
2 - t
  t %e  - t - t
(%o3) matrix([--------- + t %e + %e ,
2
- t - t 
2 %e %e - t - t - t
2 t (- ------ - ------ + %e ) + t (2 %e - ------)
2 t

$
\[-t \quad -t \quad -t \]
\[+ 2 \%e \quad , \quad t (\%e \quad - \quad -----) + t (----- \quad -----)\]
\[\quad \frac{2 \quad - \quad t \quad - \quad t \quad - \quad t \quad - \quad t}{2 \quad t \quad 2} \]
\[\quad \frac{-t \quad t \quad \%e \quad t \quad \%e \quad - \quad t \quad + \quad \%e \quad \%e \quad - \quad t}{2 \quad t \quad 2} \]
\[\quad \frac{2 \quad \%e \quad \%e \quad t \quad \%e \quad - \quad t \quad - \quad t \quad - \quad t \quad - \quad t}{2 \quad t \quad 2} \]
\[\quad \frac{-t \quad - \quad t \quad 2 \quad - \quad t \quad - \quad t \quad (----- \quad -----) \quad , \quad (----- \quad ----- \quad + \quad \%e \quad \%e \quad - \quad t \quad \%e \quad \%e \quad - \quad t \quad - \quad t \quad (----- \quad -----) \quad , \quad (----- \quad ----- \quad + \quad \%e \quad \%e \quad - \quad t \quad \%e \quad \%e \quad - \quad t \quad (----- \quad -----))}{2 \quad t \quad t} \]
\[\quad \frac{2 \quad \%e \quad \%e \quad \%e \quad t \quad \%e \quad - \quad t \quad - \quad t \quad t \quad \%e \quad \%e \quad - \quad t \quad \%e \quad \%e \quad - \quad t \quad (----- \quad ----- \quad - \quad t \quad \%e \quad \%e \quad - \quad t \quad \%e \quad \%e \quad - \quad t \quad (----- \quad -----) \quad - \quad t \quad \%e \quad \%e \quad - \quad t \quad \%e \quad \%e \quad - \quad t \quad (----- \quad -----))}{2 \quad t \quad t} \]
\[\quad \text{(\%o4) ratsimp(\%);} \]
\[\quad \text{Col 1 = } \]
\[\quad \text{Col 2 = } \]
Example 2:

(%i5) b1: matrix([0,0,1,1,1],
               [0,0,0,1,1],
               [0,0,0,0,1],
               [0,0,0,0,0],
               [0,0,0,0,0])$

(%i6) mat_function(exp, t*b1);

(%o6) [ 2 1 0 t t -- + t ]
    [ 2 ]

(%i7) minimalPoly(jordan(b1));

(%o7) 3

(%i8) ident(5)+t*b1+1/2*(t^2)*b1^^2;

(%o8) [ 2 1 0 t t -- + t ]
    [ 2 ]

(%i9) mat_function(exp, %i*t*b1);

(%o9) [ 2 ]
Chapter 49: diag

Example 3:

(%i11) a1: matrix([2,1,0,0,0,0],
             [-1,4,0,0,0,0],
             [-1,1,2,1,0,0],
             [-1,1,-1,4,0,0],
             [-1,1,-1,1,3,0],
             [-1,1,-1,1,1,2])$

(%i12) fpow(x):=block([k],
declare(k,integer),x^k)$

(%i13) mat_function(fpow,a1);

(%o13) Col 1 =

(%i14) Col 2 =

To use this function write first `load("diag")`. 
Chapter 50: distrib

50 distrib

50.1 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, $D[X] = \sqrt{V[X]}$, 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

$$ E[X] = \sum x_i \, f(x_i), $$

$$ V[X] = \sum (x_i - E[X])^2 \, f(x_i), $$

$$ D[X] = \sqrt{V[X]}, $$

$$ SK[X] = \frac{\sum (x_i - E[X])^3 \, f(x_i)}{D[X]^3}, $$

$$ KU[X] = \frac{\sum (x_i - E[X])^4 \, f(x_i)}{D[X]^4} - 3 $$
\[ V[X] = \sum_{x_i} f(x_i) (x_i - E[X])^2, \]

\[ D[X] = \sqrt{V[X]}, \]

\[ SK[X] = \frac{\sum_{x_i} f(x) (x - E[X])^3}{D[X]^3} \]

and

\[ KU[X] = \frac{\sum_{x_i} f(x) (x - E[X])^4}{D[X]^4} - 3, \]

respectively.

There is a naming convention in package \texttt{distrib}. Every function name has two parts, the first one makes reference to the function or parameter we want to calculate,

Functions:
- Density function (pdf_*
- Distribution function (cdf_*
- Quantile (quantile_*
- Mean (mean_*
- Variance (var_*
- Standard deviation (std_*
- Skewness coefficient (skewness_*
- Kurtosis coefficient (kurtosis_*
- Random variate (random_*

The second part is an explicit reference to the probabilistic model,

Continuous distributions:
- Normal (*normal)
- Student (*student_t)
- Chi^2 (*chi2)
- Noncentral Chi^2 (*noncentral_chi2)
- F (*f)
- Exponential (*exp)
- Lognormal (*lognormal)
- Gamma (*gamma)
- Beta (*beta)
- Continuous uniform (*continuous_uniform)
- Logistic (*logistic)
- Pareto (*pareto)
- Weibull (*weibull)
- Rayleigh (*rayleigh)
- Laplace (*laplace)
- Cauchy (*cauchy)
- Gumbel (*gumbel)

Discrete distributions:
- Binomial (*binomial)
Chapter 50: distrib

Poisson (*poisson)
Bernoulli (*bernoulli)
Geometric (*geometric)
Discrete uniform (*discrete_uniform)
hypergeometric (*hypergeometric)
Negative binomial (*negative_binomial)

For example, pdf_student_t(x,n) is the density function of the Student distribution with $n$ degrees of freedom, std_pareto(a,b) is the standard deviation of the Pareto distribution with parameters $a$ and $b$ and kurtosis_poisson(m) is the kurtosis coefficient of the Poisson distribution with mean $m$.

In order to make use of package distrib you need first to load it by typing

```
(%i1) load(distrib)$
```

For comments, bugs or suggestions, please contact the author at 'mario AT edu DOT xunta DOT es'.

50.2 Functions and Variables for continuous distributions

**pdf_normal** $(x,m,s)$ Function
Returns the value at $x$ of the density function of a Normal$(m, s)$ random variable, with $s > 0$. To make use of this function, write first load(distrib).

**cdf_normal** $(x,m,s)$ Function
Returns the value at $x$ of the distribution function of a Normal$(m, s)$ random variable, with $s > 0$. This function is defined in terms of Maxima’s built-in error function $erf$.

```
(%i1) load (distrib)$
(%i2) assume(s>0)$ cdf_normal(x,m,s);
```

```
x - m
erf(----------) + -
sqrt(2) s 1
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);
```

```
sqrt(2) inverse_erf(---)
10
```

```
1.644853626951472
```
**mean_normal** \((m,s)\)

Function

Returns the mean of a \(Normal(m, s)\) random variable, with \(s > 0\), namely \(m\). To make use of this function, write first `load(distrib)`.

**var_normal** \((m,s)\)

Function

Returns the variance of a \(Normal(m, s)\) random variable, with \(s > 0\), namely \(s^2\). To make use of this function, write first `load(distrib)`.

**std_normal** \((m,s)\)

Function

Returns the standard deviation of a \(Normal(m, s)\) random variable, with \(s > 0\), namely \(s\). To make use of this function, write first `load(distrib)`.

**skewness_normal** \((m,s)\)

Function

Returns the skewness coefficient of a \(Normal(m, s)\) random variable, with \(s > 0\), which is always equal to 0. To make use of this function, write first `load(distrib)`.

**kurtosis_normal** \((m,s)\)

Function

Returns the kurtosis coefficient of a \(Normal(m, s)\) random variable, with \(s > 0\), which is always equal to 0. To make use of this function, write first `load(distrib)`.

**random_normal** \((m,s)\)

Function

**random_normal** \((m,s,n)\)

Function

Returns a \(Normal(m, s)\) random variate, with \(s > 0\). Calling `random_normal` with a third argument \(n\), a random sample of size \(n\) will be simulated.

This is an implementation of the Box-Mueller algorithm, as described in Knuth, D.E. (1981) *Seminumerical Algorithms. The Art of Computer Programming*. Addison-Wesley.

To make use of this function, write first `load(distrib)`.

**pdf_student_t** \((x,n)\)

Function

Returns the value at \(x\) of the density function of a Student random variable \(t(n)\), with \(n > 0\) degrees of freedom. To make use of this function, write first `load(distrib)`.

**cdf_student_t** \((x,n)\)

Function

Returns the value at \(x\) of the distribution function of a Student random variable \(t(n)\), with \(n > 0\) degrees of freedom.

```plaintext
(%i1) load (distrib)$
(%i2) cdf_student_t(1/2, 7/3);

7 1 28
beta_incomplete_regularized(-, -, --)
6 2 31

(%o2) 1 - -------------------------------------

(%i3) float(%);
(%o3) .6698450596140415
```
quantile_student_t (q,n) 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) 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) Returns the variance of a Student random variable t(n), with n > 2.

std_student_t (n) 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) 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) 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) 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, \( \text{Chi}^2(n) \), 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) 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.
634 Maxima 5.24.0 Manual

(%i1) load (distrib)$
(%i2) expand(pdf_noncentral_student_t(3,5,0.1));
   .01370030107589574 sqrt(5)
(%o2) -----------------------------------
   sqrt(2) sqrt(14) sqrt(%pi)
   1.65456288411515E-4 sqrt(5)
+ -----------------------------------
   sqrt(%pi)
   .02434921505438663 sqrt(5)
+ -----------------------------------
   %pi
(%i3) float(%);
(%o3) .02080593159405669

cdf_noncentral_student_t (x,n,ncp)
Function
Returns the value at \( x \) of the distribution function of a noncentral Student random variable \( nc_t(n,ncp) \), with \( n > 0 \) degrees of freedom and noncentrality parameter \( ncp \). This function has no closed form and it is numerically computed if the global variable \( \text{numer} \) equals \( \text{true} \) or at least one of the arguments is a float, otherwise it returns a nominal expression.

(%i1) load (distrib)$
(%i2) cdf_noncentral_student_t(-2,5,-5);
(%o2) cdf_noncentral_student_t(- 2, 5, - 5)
(%i3) cdf_noncentral_student_t(-2.0,5,-5);
(%o3) .9952030093319743

quantile_noncentral_student_t (q,n,ncp)
Function
Returns the \( q \)-quantile of a noncentral Student random variable \( nc_t(n,ncp) \), with \( n > 0 \) degrees of freedom and noncentrality parameter \( ncp \); in other words, this is the inverse of \( \text{cdf_noncentral_student_t} \). Argument \( q \) must be an element of \([0,1]\). To make use of this function, write first \( \text{load(distrib)} \).

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 \( \text{load(distrib)} \).

(%i1) load (distrib)$
(%i2) (assume(df>1), mean_noncentral_student_t(df,k));
   df - 1
   gamma(-- ---) sqrt(df) k
   2
(%o2) -----------------------------------
   df
   sqrt(2) gamma(--)
   2
var_noncentral_student_t (n,ncp)  
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)  
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)  
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)  
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)  
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)  
Returns the value at x of the density function of a Chi-square random variable Chi^2(n), with n > 0.

The Chi^2(n) random variable is equivalent to the Gamma(n/2,2), therefore when Maxima has not enough information to get the result, a noun form based on the gamma density is returned.

(%i1) load (distrib)$  
(%i2) pdf.chi2(x,n);  
(%o2) pdf_gamma(x, -, 2)
(%i3) assume(x>0, n>0)$ pdf_chisq(x,n);
   n/2 - 1 - x/2
   x %e
---------------------
n/2 n
2 gamma(-)
2

Function cdf_chisq (x,n)
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_chisq(3,4);
   3
   1 - gamma_incomplete_regularized(2, -)
   2
(%o2) 1 - gamma_incomplete_regularized(2, -)
2
(%i3) float(%);
   %o3) 1 - gamma_incomplete_regularized(2, -)
   2
   .4421745996289256

Function quantile_chisq (q,n)
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_chisq. Argument q must be an element of \([0, 1]\).
This function has no closed form and it is numerically computed if the global variable numer equals true, otherwise it returns a nominal expression based on the gamma quantile function, since the \( Chi^2(n) \) random variable is equivalent to the \( Gamma(n/2, 2) \).

(%i1) load (distrib)$
(%i2) quantile_chisq(0.99,9);
   9
   21.66599433346194
   2
(%o2) 21.66599433346194
(%i3) quantile_chisq(0.99,n);
   n
   quantile_gamma(0.99, -, 2)
   2
(%o3) quantile_gamma(0.99, -, 2)

Function mean_chisq (n)
Returns the mean of a Chi-square random variable \( Chi^2(n) \), with \( n > 0 \).
The \( Chi^2(n) \) random variable is equivalent to the \( Gamma(n/2, 2) \), therefore when Maxima has not enough information to get the result, a noun form based on the gamma mean is returned.

(%i1) load (distrib)$
(%i2) mean_chisq(n);
   n
   mean_gamma(-, 2)
   2
(%o2) mean_gamma(-, 2)
(%i3) assume(n>0)$ mean_chisq(n);
(%o3) mean_gamma(-, 2)
(%o4) n
\textbf{var\_chi2} \((n)\) 

Returns the variance of a Chi-square random variable \(\chi^2(n)\), with \(n > 0\).

The \(\chi^2(n)\) random variable is equivalent to the \(\Gamma(n/2, 2)\), therefore when Maxima has not enough information to get the result, a noun form based on the gamma variance is returned.

\begin{verbatim}
(%i1) load (distrib)$
(%i2) var\_chi2(n);
(%o2) var\_gamma(-, 2)

(%i3) assume(n>0)$ var\_chi2(n);
(%o4) \frac{n}{2}
\end{verbatim}

\textbf{std\_chi2} \((n)\) 

Returns the standard deviation of a Chi-square random variable \(\chi^2(n)\), with \(n > 0\).

The \(\chi^2(n)\) random variable is equivalent to the \(\Gamma(n/2, 2)\), therefore when Maxima has not enough information to get the result, a noun form based on the gamma standard deviation is returned.

\begin{verbatim}
(%i1) load (distrib)$
(%i2) std\_chi2(n);
(%o2) std\_gamma(-, 2)

(%i3) assume(n>0)$ std\_chi2(n);
(%o4) \frac{\sqrt{2} \sqrt{n}}{2}
\end{verbatim}

\textbf{skewness\_chi2} \((n)\) 

Returns the skewness coefficient of a Chi-square random variable \(\chi^2(n)\), with \(n > 0\).

The \(\chi^2(n)\) random variable is equivalent to the \(\Gamma(n/2, 2)\), therefore when Maxima has not enough information to get the result, a noun form based on the gamma skewness coefficient is returned.

\begin{verbatim}
(%i1) load (distrib)$
(%i2) skewness\_chi2(n);
(%o2) skewness\_gamma(-, 2)

(%i3) assume(n>0)$ skewness\_chi2(n);
(%o4) \frac{2 \sqrt{2}}{\sqrt{n}}
\end{verbatim}

\textbf{kurtosis\_chi2} \((n)\) 

Returns the kurtosis coefficient of a Chi-square random variable \(\chi^2(n)\), with \(n > 0\).

The \(\chi^2(n)\) random variable is equivalent to the \(\Gamma(n/2, 2)\), therefore when Maxima has not enough information to get the result, a noun form based on the gamma kurtosis coefficient is returned.
(%i1) load (distrib)$
(%i2) kurtosis_chi2(n);
   n
(%o2) kurtosis_gamma(-, 2)
   2
(%i3) assume(n>0)$ kurtosis_chi2(n);
   12
   --
   n

\textbf{random\_chi2\ (n)} \quad \text{Function}

\text{Returns a Chi-square random variate } \chi^2(n), \text{ with } n > 0. \text{ Calling } \text{random\_chi2} \text{ with a second argument } m, \text{ a random sample of size } m \text{ will be simulated.}

\text{The simulation is based on the Ahrens-Cheng algorithm. See } \text{random\_gamma} \text{ for details.}

\text{To make use of this function, write first } \text{load(distrib)}.\n
\textbf{pdf\_noncentral\_chi2\ (x,u,ncp)} \quad \text{Function}

\text{Returns the value at } x \text{ of the density function of a noncentral Chi-square random variable } \chi^2(n, ncp), \text{ with } n > 0 \text{ and noncentrality parameter } ncp >= 0. \text{ To make use of this function, write first } \text{load(distrib)}.\n
\textbf{cdf\_noncentral\_chi2\ (x,u,ncp)} \quad \text{Function}

\text{Returns the value at } x \text{ of the distribution function of a noncentral Chi-square random variable } \chi^2(n, ncp), \text{ with } n > 0 \text{ and noncentrality parameter } ncp >= 0. \text{ To make use of this function, write first } \text{load(distrib)}.\n
\textbf{quantile\_noncentral\_chi2\ (q,u,ncp)} \quad \text{Function}

\text{Returns the } q \text{-quantile of a noncentral Chi-square random variable } \chi^2(n, ncp), \text{ with } n > 0 \text{ and noncentrality parameter } ncp >= 0; \text{ in other words, this is the inverse of } \text{cdf\_noncentral\_chi2}. \text{ Argument } q \text{ must be an element of } [0, 1]. \text{ This function has no closed form and it is numerically computed if the global variable } \text{numer} \text{ equals } \text{true}, \text{ otherwise it returns a nominal expression.}\n
\textbf{mean\_noncentral\_chi2\ (n,ncp)} \quad \text{Function}

\text{Returns the mean of a noncentral Chi-square random variable } \chi^2(n, ncp), \text{ with } n > 0 \text{ and noncentrality parameter } ncp >= 0.\n
\textbf{var\_noncentral\_chi2\ (n,ncp)} \quad \text{Function}

\text{Returns the variance of a noncentral Chi-square random variable } \chi^2(n, ncp), \text{ with } n > 0 \text{ and noncentrality parameter } ncp >= 0.\n
\textbf{std\_noncentral\_chi2\ (n,ncp)} \quad \text{Function}

\text{Returns the standard deviation of a noncentral Chi-square random variable } \chi^2(n, ncp), \text{ with } n > 0 \text{ and noncentrality parameter } ncp >= 0.
skewness_noncentral_chi2 \((n, ncp)\)

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 \((n, ncp)\)

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 \((n, ncp)\)

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 \texttt{load(distrib)}.

pdf_f \((x, m, n)\)

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 \texttt{load(distrib)}.

cdf_f \((x, m, n)\)

Returns the value at \(x\) of the distribution function of a F random variable \(F(m, n)\), with \(m, n > 0\).

\[
\begin{align*}
\texttt{(i1)} & \quad \texttt{load (distrib)} \\
\texttt{(i2)} & \quad \texttt{cdf_f(2,3,9/4)}; \\
\texttt{(o2)} & \quad 9 3 3 \\
& \quad 1 - \texttt{beta_incomplete_regularized(-,-,--)} \\
& \quad 8 2 11 \\
\texttt{(i3)} & \quad \texttt{float(o2)}; \\
\texttt{(o3)} & \quad 0.66756728179008
\end{align*}
\]

quantile_f \((q, m, n)\)

Returns the \(q\)-quantile of a F random variable \(F(m, n)\), with \(m, n > 0\); in other words, this is the inverse of \texttt{cdf_f}. Argument \(q\) must be an element of \([0, 1]\).

This function has no closed form and it is numerically computed if the global variable \texttt{numer} equals \texttt{true}, otherwise it returns a nominal expression.

\[
\begin{align*}
\texttt{(i1)} & \quad \texttt{load (distrib)} \\
\texttt{(i2)} & \quad \texttt{quantile_f(2/5,sqrt(3),5)}; \\
\texttt{(o2)} & \quad \texttt{quantile_f(-, sqrt(3), 5)} \\
\texttt{(i3)} & \quad \%,	exttt{numer}; \\
\texttt{(o3)} & \quad 0.518947838573693
\end{align*}
\]

mean_f \((m, n)\)

Returns the mean of a F random variable \(F(m, n)\), with \(m > 0, n > 2\). To make use of this function, write first \texttt{load(distrib)}.
Function var_f (m,n)
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).

Function std_f (m,n)
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).

Function skewness_f (m,n)
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).

Function kurtosis_f (m,n)
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).

Function 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).

Function pdf_exp (x,m)
Returns the value at x of the density function of an Exponential(m) random variable, with m > 0.

The Exponential(m) random variable is equivalent to the Weibull(1, 1/m), therefore when Maxima has not enough information to get the result, a noun form based on the Weibull density is returned.

```
(%i1) load (distrib)$
(%i2) pdf_exp(x,m);
(%o2) pdf_weibull(x, 1, -)

(%i3) assume(x>0,m>0)$ pdf_exp(x,m);
(%o4) m %e
```


**cdf_exp** \((x, m)\)

Returns the value at \(x\) of the distribution function of an \(\text{Exponential}(m)\) random variable, with \(m > 0\).

The \(\text{Exponential}(m)\) random variable is equivalent to the \(\text{Weibull}(1, 1/m)\), therefore when Maxima has not enough information to get the result, a noun form based on the Weibull distribution is returned.

```maxima
(%i1) load (distrib)$
(%i2) cdf_exp(x, m);
  1
(%o2) cdf_weibull(x, 1, -)
   m
(%i3) assume(x>0, m>0)$
(%o3) cdf_exp(x, m);
  - m x
(%o4) 1 - %e
```

**quantile_exp** \((q, m)\)

Returns the \(q\)-quantile of an \(\text{Exponential}(m)\) random variable, with \(m > 0\); in other words, this is the inverse of \(\text{cdf_exp}\). Argument \(q\) must be an element of \([0, 1]\).

The \(\text{Exponential}(m)\) random variable is equivalent to the \(\text{Weibull}(1, 1/m)\), therefore when Maxima has not enough information to get the result, a noun form based on the Weibull quantile is returned.

```maxima
(%i1) load (distrib)$
(%i2) quantile_exp(0.56, 5);
= .1641961104139661
(%o2) quantile_exp(0.56, m);
  1
(%o3) quantile_weibull(0.56, 1, -)
   m
```

**mean_exp** \((m)\)

Returns the mean of an \(\text{Exponential}(m)\) random variable, with \(m > 0\).

The \(\text{Exponential}(m)\) random variable is equivalent to the \(\text{Weibull}(1, 1/m)\), therefore when Maxima has not enough information to get the result, a noun form based on the Weibull mean is returned.

```maxima
(%i1) load (distrib)$
(%i2) mean_exp(m);
  1
(%o2) mean_weibull(1, -)
   m
(%i3) assume(m>0)$
(%o3) mean_exp(m);
  1
(%o4) -
   m
```

**var_exp** \((m)\)

Returns the variance of an \(\text{Exponential}(m)\) random variable, with \(m > 0\).
The $\text{Exponential}(m)$ random variable is equivalent to the $\text{Weibull}(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull variance is returned.

```maxima
(%i1) load (distrib)$
(%i2) var_exp(m);
1
(%o2) var_weibull(1, -)
m
(%i3) assume(m>0)$ var_exp(m);
1
(%o4) --
2
m
```

**std_exp** ($m$)

Function

Returns the standard deviation of an $\text{Exponential}(m)$ random variable, with $m > 0$.

The $\text{Exponential}(m)$ random variable is equivalent to the $\text{Weibull}(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull standard deviation is returned.

```maxima
(%i1) load (distrib)$
(%i2) std_exp(m);
1
(%o2) std_weibull(1, -)
m
(%i3) assume(m>0)$ std_exp(m);
1
(%o4) --
2
m
```

**skewness_exp** ($m$)

Function

Returns the skewness coefficient of an $\text{Exponential}(m)$ random variable, with $m > 0$.

The $\text{Exponential}(m)$ random variable is equivalent to the $\text{Weibull}(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull skewness coefficient is returned.

```maxima
(%i1) load (distrib)$
(%i2) skewness_exp(m);
1
(%o2) skewness_weibull(1, -)
m
(%i3) assume(m>0)$ skewness_exp(m);
1
(%o4) 2
```

**kurtosis_exp** ($m$)

Function

Returns the kurtosis coefficient of an $\text{Exponential}(m)$ random variable, with $m > 0$.

The $\text{Exponential}(m)$ random variable is equivalent to the $\text{Weibull}(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull kurtosis coefficient is returned.

```maxima
```
random_exp (m) Function
random_exp (m,k) Function
Returns an Exponential(m) random variate, with m > 0. Calling random_exp with a second argument k, a random sample of size k will be simulated.
The simulation algorithm is based on the general inverse method.
To make use of this function, write first load(distrib).

pdf_lognormal (x,m,s) Function
Returns the value at x of the density function of a Lognormal(m, s) random variable, with s > 0. To make use of this function, write first load(distrib).
cdf_lognormal (x,m,s) Function
Returns the value at x of the distribution function of a Lognormal(m, s) random variable, with s > 0. This function is defined in terms of Maxima’s built-in error function erf.

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).

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 \((m,s)\)  
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 \((m,s)\)  
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 \((m,s)\)  
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 \((m,s)\)  
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 \((m,s)\)  
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 \((x,a,b)\)  
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 \((x,a,b)\)  
Returns the value at \(x\) of the distribution function of a Gamma\((a, b)\) random variable, with \(a, b > 0\).

```plaintext
(%i1) load (distrib)$
(%i2) cdf_gamma(3,5,21);
(%o2) 1 - gamma_incomplete_regularized(5, -) 7
(%i3) float(%);
(%o3) 4.402663157376807E-7
```

quantile_gamma \((q,a,b)\)  
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 \((a,b)\)  
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** \((a,b)\)  
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** \((a,b)\)  
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** \((a,b)\)  
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** \((a,b)\)  
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** \((a,b)\)  
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 \(a\):  
To make use of this function, write first `load(distrib)`.  

**pdf.beta** \((x,a,b)\)  
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** \((x,a,b)\)  
Returns the value at \(x\) of the distribution function of a Beta\((a, b)\) random variable, with \(a, b > 0\).  
\[
\begin{align*}
(\%i1) & \text{ load (distrib)}$
(\%i2) & \text{ cdf_beta(1/3,15,2);} \\
(\%o2) & 11 \\
(\%i3) & \text{ float(\%);} \\
(\%o3) & 7.666089131388195E-7
\end{align*}
\]

**quantile.beta** \((q,a,b)\)  
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)`.
Function \texttt{mean} \((a,b)\)

Returns the mean of a \(Beta(a,b)\) random variable, with \(a, b > 0\). To make use of this function, write first \texttt{load(distrib)}.

Function \texttt{var} \((a,b)\)

Returns the variance of a \(Beta(a,b)\) random variable, with \(a, b > 0\). To make use of this function, write first \texttt{load(distrib)}.

Function \texttt{std} \((a,b)\)

Returns the standard deviation of a \(Beta(a,b)\) random variable, with \(a, b > 0\). To make use of this function, write first \texttt{load(distrib)}.

Function \texttt{skewness} \((a,b)\)

Returns the skewness coefficient of a \(Beta(a,b)\) random variable, with \(a, b > 0\). To make use of this function, write first \texttt{load(distrib)}.

Function \texttt{kurtosis} \((a,b)\)

Returns the kurtosis coefficient of a \(Beta(a,b)\) random variable, with \(a, b > 0\). To make use of this function, write first \texttt{load(distrib)}.

Function \texttt{random} \((a,b)\)

Function \texttt{random} \((a,b,n)\)

Returns a \(Beta(a,b)\) random variate, with \(a, b > 0\). Calling \texttt{random\_beta} with a third argument \(n\), a random sample of size \(n\) will be simulated.


To make use of this function, write first \texttt{load(distrib)}.

Function \texttt{pdf\_continuous\_uniform} \((x,a,b)\)

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 \texttt{load(distrib)}.

Function \texttt{cdf\_continuous\_uniform} \((x,a,b)\)

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 \texttt{load(distrib)}.

Function \texttt{quantile\_continuous\_uniform} \((q,a,b)\)

Returns the \(q\)-quantile of a \(ContinuousUniform(a,b)\) random variable, with \(a < b\); in other words, this is the inverse of \texttt{cdf\_continuous\_uniform}. Argument \(q\) must be an element of \([0,1]\). To make use of this function, write first \texttt{load(distrib)}.

Function \texttt{mean\_continuous\_uniform} \((a,b)\)

Returns the mean of a \(ContinuousUniform(a,b)\) random variable, with \(a < b\). To make use of this function, write first \texttt{load(distrib)}.
**var_continuous_uniform** \((a,b)\)  
Returns the variance of a \emph{ContinuousUniform}(a,b) random variable, with \(a < b\).  
To make use of this function, write first \texttt{load(distrib)}.

**std_continuous_uniform** \((a,b)\)  
Returns the standard deviation of a \emph{ContinuousUniform}(a,b) random variable, with \(a < b\). To make use of this function, write first \texttt{load(distrib)}.

**skewness_continuous_uniform** \((a,b)\)  
Returns the skewness coefficient of a \emph{ContinuousUniform}(a,b) random variable, with \(a < b\). To make use of this function, write first \texttt{load(distrib)}.

**kurtosis_continuous_uniform** \((a,b)\)  
Returns the kurtosis coefficient of a \emph{ContinuousUniform}(a,b) random variable, with \(a < b\). To make use of this function, write first \texttt{load(distrib)}.

**random_continuous_uniform** \((a,b)\)  
**random_continuous_uniform** \((a,b,n)\)  
Returns a \emph{ContinuousUniform}(a,b) random variate, with \(a < b\). Calling \texttt{random_continuous_uniform} with a third argument \(n\), a random sample of size \(n\) will be simulated.  
This is a direct application of the \texttt{random} built-in Maxima function.  
See also \texttt{random}. To make use of this function, write first \texttt{load(distrib)}.

**pdf_logistic** \((x,a,b)\)  
Returns the value at \(x\) of the density function of a \emph{Logistic}(a,b) random variable, with \(b > 0\). To make use of this function, write first \texttt{load(distrib)}.

**cdf_logistic** \((x,a,b)\)  
Returns the value at \(x\) of the distribution function of a \emph{Logistic}(a,b) random variable, with \(b > 0\). To make use of this function, write first \texttt{load(distrib)}.

**quantile_logistic** \((q,a,b)\)  
Returns the \(q\)-quantile of a \emph{Logistic}(a,b) random variable, with \(b > 0\); in other words, this is the inverse of \texttt{cdf_logistic}. Argument \(q\) must be an element of [0,1].  
To make use of this function, write first \texttt{load(distrib)}.

**mean_logistic** \((a,b)\)  
Returns the mean of a \emph{Logistic}(a,b) random variable, with \(b > 0\). To make use of this function, write first \texttt{load(distrib)}.

**var_logistic** \((a,b)\)  
Returns the variance of a \emph{Logistic}(a,b) random variable, with \(b > 0\). To make use of this function, write first \texttt{load(distrib)}.
std_logistic \((a,b)\)  
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 \((a,b)\)  
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 \((a,b)\)  
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 \((a,b)\)  
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 \((x,a,b)\)  
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 \((x,a,b)\)  
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 \((q,a,b)\)  
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 \((a,b)\)  
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 \((a,b)\)  
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 \((a,b)\)  
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 \((a,b)\)  
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)`.
Chapter 50: 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

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)`.

$random\_pareto\ (a,b,n)$

Function

$pdf\_weibull\ (x,a,b)$

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)`.
Function \texttt{random}\_weibull \((a,b)\)  

Returns a \textit{Weibull}(a,b) random variate, with \(a,b > 0\). Calling \texttt{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 \texttt{load(distrib)}.

Function \texttt{pdf}\_rayleigh \((x,b)\)  

Returns the value at \(x\) of the density function of a \textit{Rayleigh}(b) random variable, with \(b > 0\).

The \textit{Rayleigh}(b) random variable is equivalent to the \textit{Weibull}(2, 1/b), therefore when Maxima has not enough information to get the result, a noun form based on the Weibull density is returned.

\begin{verbatim}
(%i1) load (distrib)$
(%i2) pdf_rayleigh(x,b);
   1
(%o2) pdf_weibull(x, 2, ~)
   b
(%i3) assume(x>0,b>0)$ pdf_rayleigh(x,b);
         2 2
2   b x
---
x
2 b x %e

(%o4)
\end{verbatim}

Function \texttt{cdf}\_rayleigh \((x,b)\)  

Returns the value at \(x\) of the distribution function of a \textit{Rayleigh}(b) random variable, with \(b > 0\).

The \textit{Rayleigh}(b) random variable is equivalent to the \textit{Weibull}(2, 1/b), therefore when Maxima has not enough information to get the result, a noun form based on the Weibull distribution is returned.

\begin{verbatim}
(%i1) load (distrib)$
(%i2) cdf_rayleigh(x,b);
   1
(%o2) cdf_weibull(x, 2, ~)
   b
(%i3) assume(x>0,b>0)$ cdf_rayleigh(x,b);
         2 2
2   b x
---
x
2 b x %e

(%o4)
\end{verbatim}

Function \texttt{quantile}\_rayleigh \((q,b)\)  

Returns the \(q\)-quantile of a \textit{Rayleigh}(b) random variable, with \(b > 0\); in other words, this is the inverse of \texttt{cdf}\_rayleigh. Argument \(q\) must be an element of \([0,1]\).

The \textit{Rayleigh}(b) random variable is equivalent to the \textit{Weibull}(2, 1/b), therefore when Maxima has not enough information to get the result, a noun form based on the Weibull quantile is returned.
mean\_rayleigh \ (b) \ Function

Returns the mean of a Rayleigh \( b \) random variable, with \( b > 0 \).

The Rayleigh \( b \) random variable is equivalent to the Weibull \((2, 1/b)\), therefore when Maxima has not enough information to get the result, a noun form based on the Weibull mean is returned.

\begin{verbatim}
(%i1) load (distrib)$
(%i2) mean\_rayleigh(b);
1
(%o2) mean\_weibull(2, -)
   b

(%i3) assume(b>0)$ mean\_rayleigh(b);

sqrt(\%pi)
(%o4) -------------
2 b
\end{verbatim}

var\_rayleigh \ (b) \ Function

Returns the variance of a Rayleigh \( b \) random variable, with \( b > 0 \).

The Rayleigh \( b \) random variable is equivalent to the Weibull \((2, 1/b)\), therefore when Maxima has not enough information to get the result, a noun form based on the Weibull variance is returned.

\begin{verbatim}
(%i1) load (distrib)$
(%i2) var\_rayleigh(b);
1
(%o2) var\_weibull(2, -)
   b

(%i3) assume(b>0)$ var\_rayleigh(b);

\%pi
1 - ---
4

(%o4) -------------
2 b
\end{verbatim}

std\_rayleigh \ (b) \ Function

Returns the standard deviation of a Rayleigh \( b \) random variable, with \( b > 0 \).

The Rayleigh \( b \) random variable is equivalent to the Weibull \((2, 1/b)\), therefore when Maxima has not enough information to get the result, a noun form based on the Weibull standard deviation is returned.

\begin{verbatim}
(%i1) load (distrib)$
(%i2) std\_rayleigh(b);
1
(%o2) std\_weibull(2, -)
   b

(%i3) assume(b>0)$ std\_rayleigh(b);

\%pi
1 - ---
4

(%o4) -------------
2 b
\end{verbatim}
(\%i1) load (distrib)$
(\%i2) std_rayleigh(b);
(\%o2) std_weibull(2, -)
        b
(\%i3) assume(b>0)$ std_rayleigh(b);
        \%pi
(\%o3) sqrt(1 - ----)
        4
        -------
        b

skewness_rayleigh (b)

Function

Returns the skewness coefficient of a Rayleigh(b) random variable, with \( b > 0 \).

The Rayleigh(b) random variable is equivalent to the Weibull(2, 1/b), therefore when Maxima has not enough information to get the result, a noun form based on the Weibull skewness coefficient is returned.

(\%i1) load (distrib)$
(\%i2) skewness_rayleigh(b);
(\%o2) skewness_weibull(2, -)
        b
(\%i3) assume(b>0)$ skewness_rayleigh(b);
        3/2
        \%pi
(\%o3) 3 sqrt(\%pi)
        -------
        4
        -------
        4

(\%o4)

kurtosis_rayleigh (b)

Function

Returns the kurtosis coefficient of a Rayleigh(b) random variable, with \( b > 0 \).

The Rayleigh(b) random variable is equivalent to the Weibull(2, 1/b), therefore when Maxima has not enough information to get the result, a noun form based on the Weibull kurtosis coefficient is returned.

(\%i1) load (distrib)$
(\%i2) kurtosis_rayleigh(b);
(\%o2) kurtosis_weibull(2, -)
        b
(\%i3) assume(b>0)$ kurtosis_rayleigh(b);
        2
        \%pi
(\%o3) 3 -------
        2
        -------
        16

(\%o4)
\[ \frac{\pi}{2} (1 - \frac{\pi}{4}) \]

**Function random_rayleigh \((b)\)**

Returns a Rayleigh\((b)\) random variate, with \(b > 0\). Calling `random_rayleigh` with a second argument \(n\), a random sample of size \(n\) will be simulated.

The implemented algorithm is based on the general inverse method. To make use of this function, write first `load(distrib)`.

**Function pdf_laplace \((x,a,b)\)**

Returns the value at \(x\) of the density function of a Laplace\((a,b)\) random variable, with \(b > 0\). To make use of this function, write first `load(distrib)`.

**Function cdf_laplace \((x,a,b)\)**

Returns the value at \(x\) of the distribution function of a Laplace\((a,b)\) random variable, with \(b > 0\). To make use of this function, write first `load(distrib)`.

**Function quantile_laplace \((q,a,b)\)**

Returns the \(q\)-quantile of a Laplace\((a,b)\) random variable, with \(b > 0\); in other words, this is the inverse of `cdf_laplace`. Argument \(q\) must be an element of \([0,1]\). To make use of this function, write first `load(distrib)`.

**Function mean_laplace \((a,b)\)**

Returns the mean of a Laplace\((a,b)\) random variable, with \(b > 0\). To make use of this function, write first `load(distrib)`.

**Function var_laplace \((a,b)\)**

Returns the variance of a Laplace\((a,b)\) random variable, with \(b > 0\). To make use of this function, write first `load(distrib)`.

**Function std_laplace \((a,b)\)**

Returns the standard deviation of a Laplace\((a,b)\) random variable, with \(b > 0\). To make use of this function, write first `load(distrib)`.

**Function skewness_laplace \((a,b)\)**

Returns the skewness coefficient of a Laplace\((a,b)\) random variable, with \(b > 0\). To make use of this function, write first `load(distrib)`.

**Function kurtosis_laplace \((a,b)\)**

Returns the kurtosis coefficient of a Laplace\((a,b)\) random variable, with \(b > 0\). To make use of this function, write first `load(distrib)`.
Function \texttt{random} \texttt{laplace}(a,b)

Returns a Laplace\((a,b)\) random variate, with \(b > 0\). Calling \texttt{random} \texttt{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 \texttt{load(distrib)}.

Function \texttt{pdf} \texttt{cauchy}(x,a,b)

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 \texttt{load(distrib)}.

Function \texttt{cdf} \texttt{cauchy}(x,a,b)

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 \texttt{load(distrib)}.

Function \texttt{quantile} \texttt{cauchy}(q,a,b)

Returns the \(q\)-quantile of a Cauchy\((a,b)\) random variable, with \(b > 0\); in other words, this is the inverse of \texttt{cdf} \texttt{cauchy}. Argument \(q\) must be an element of \([0,1]\). To make use of this function, write first \texttt{load(distrib)}.

Function \texttt{random} \texttt{cauchy}(a,b)

Returns a Cauchy\((a,b)\) random variate, with \(b > 0\). Calling \texttt{random} \texttt{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 \texttt{load(distrib)}.

Function \texttt{pdf} \texttt{gumbel}(x,a,b)

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 \texttt{load(distrib)}.

Function \texttt{cdf} \texttt{gumbel}(x,a,b)

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 \texttt{load(distrib)}.

Function \texttt{quantile} \texttt{gumbel}(q,a,b)

Returns the \(q\)-quantile of a Gumbel\((a,b)\) random variable, with \(b > 0\); in other words, this is the inverse of \texttt{cdf} \texttt{gumbel}. Argument \(q\) must be an element of \([0,1]\). To make use of this function, write first \texttt{load(distrib)}.

Function \texttt{mean} \texttt{gumbel}(a,b)

Returns the mean of a Gumbel\((a,b)\) random variable, with \(b > 0\).

\begin{verbatim}
(%i1) load (distrib)$
(%i2) assume(b>0)$ mean_gumbel(a,b);
(%o3) \%gamma b + a
\end{verbatim}

where symbol \%gamma stands for the Euler-Mascheroni constant. See also \%gamma.
var_gumbel \((a,b)\)

Returns the variance of a Gumbel\((a,b)\) random variable, with \(b > 0\). To make use of this function, write first \texttt{load(distrib)}.

std_gumbel \((a,b)\)

Returns the standard deviation of a Gumbel\((a,b)\) random variable, with \(b > 0\). To make use of this function, write first \texttt{load(distrib)}.

skewness_gumbel \((a,b)\)

Returns the skewness coefficient of a Gumbel\((a,b)\) random variable, with \(b > 0\).

\[
\text{(i1)} \quad \texttt{load (distrib)}$
\]
\[
\text{(i2)} \quad \texttt{assume(b>0)}$
\]
\[
\texttt{skewness_gumbel(a,b);} \\
12 \sqrt{6} \texttt{ zeta(3)}
\]
\[
\text{(o3)} \quad \frac{3}{\pi}
\]

\[
\text{(i4)} \quad \texttt{numer:true}$
\]
\[
\texttt{skewness_gumbel(a,b);} \\
1.139547099404649
\]

where \texttt{zeta} stands for the Riemann’s zeta function.

kurtosis_gumbel \((a,b)\)

Returns the kurtosis coefficient of a Gumbel\((a,b)\) random variable, with \(b > 0\). To make use of this function, write first \texttt{load(distrib)}.

random_gumbel \((a,b)\)

random_gumbel \((a,b,n)\)

Returns a Gumbel\((a,b)\) random variate, with \(b > 0\). Calling \texttt{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 \texttt{load(distrib)}.

50.3 Functions and Variables for discrete distributions

pdf_binomial \((x,n,p)\)

Returns the value at \(x\) of the probability function of a Binomial\((n,p)\) random variable, with \(0 < p < 1\) and \(n\) a positive integer. To make use of this function, write first \texttt{load(distrib)}.

\[
\text{(i1)} \quad \texttt{load (distrib)}$
\]
\[
\text{(i2)} \quad \texttt{cdf_binomial(5,7,1/6);} \\
7775
\]

\[
\text{(o2)} \quad ----
\]
quantile_binomial (q,n,p)  
    Function  
    Returns the q-quantile of a Binomial(n,p) random variable, with 0 < p < 1 and n a positive integer; in other words, this is the inverse of cdf_binomial. Argument q must be an element of [0, 1]. To make use of this function, write first load(distrib).

mean_binomial (n,p)  
    Function  
    Returns the mean of a Binomial(n,p) random variable, with 0 < p < 1 and n a positive integer. To make use of this function, write first load(distrib).

var_binomial (n,p)  
    Function  
    Returns the variance of a Binomial(n,p) random variable, with 0 < p < 1 and n a positive integer. To make use of this function, write first load(distrib).

std_binomial (n,p)  
    Function  
    Returns the standard deviation of a Binomial(n,p) random variable, with 0 < p < 1 and n a positive integer. To make use of this function, write first load(distrib).

skewness_binomial (n,p)  
    Function  
    Returns the skewness coefficient of a Binomial(n,p) random variable, with 0 < p < 1 and n a positive integer. To make use of this function, write first load(distrib).

kurtosis_binomial (n,p)  
    Function  
    Returns the kurtosis coefficient of a Binomial(n,p) random variable, with 0 < p < 1 and n a positive integer. To make use of this function, write first load(distrib).

random_binomial (n,p)  
    Function  
    random_binomial (n,p,m)  
    Function  
    Returns a Binomial(n,p) random variate, with 0 < p < 1 and n a positive integer. Calling random_binomial with a third argument m, a random sample of size m will be simulated.
    To make use of this function, write first load(distrib).

dpdf_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.
Chapter 50: distrib

(%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 \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{load(distrib)}.

random_poisson \((m)\)

Function

Returns a \(Poisson(m)\) random variate, with \(m > 0\). Calling \texttt{random_poisson} with a second argument \(n\), a random sample of size \(n\) will be simulated.


To make use of this function, write first \texttt{load(distrib)}.

pdf_bernoulli \((x,p)\)

Function

Returns the value at \(x\) of the probability function of a \(Bernoulli(p)\) random variable, with \(0 < p < 1\).

The \(Bernoulli(p)\) random variable is equivalent to the \(Binomial(1,p)\), therefore when Maxima has not enough information to get the result, a noun form based on the binomial probability function is returned.
(%i1) load (distrib)$
(%i2) pdf_bernoulli(1,p);
(%o2)    pdf_binomial(1, 1, p)
(%i3) assume(0<p,p<1)$ pdf_bernoulli(1,p);
(%o4)       p

cdf_bernoulli (x,p) 
Function
Returns the value at x of the distribution function of a Bernoulli(p) random variable, with 0 < p < 1. To make use of this function, write first load(distrib).

quantile_bernoulli (q,p) 
Function
Returns the q-quantile of a Bernoulli(p) random variable, with 0 < p < 1; in other words, this is the inverse of cdf_bernoulli. Argument q must be an element of [0,1]. To make use of this function, write first load(distrib).

mean_bernoulli (p) 
Function
Returns the mean of a Bernoulli(p) random variable, with 0 < p < 1.
The Bernoulli(p) random variable is equivalent to the Binomial(1,p), therefore when Maxima has not enough information to get the result, a noun form based on the binomial mean is returned.

(%i1) load (distrib)$
(%i2) mean_bernoulli(p);
(%o2) mean_binomial(1, p)
(%i3) assume(0<p,p<1)$ mean_bernoulli(p);
(%o4)       p

var_bernoulli (p) 
Function
Returns the variance of a Bernoulli(p) random variable, with 0 < p < 1.
The Bernoulli(p) random variable is equivalent to the Binomial(1,p), therefore when Maxima has not enough information to get the result, a noun form based on the binomial variance is returned.

(%i1) load (distrib)$
(%i2) var_bernoulli(p);
(%o2) var_binomial(1, p)
(%i3) assume(0<p,p<1)$ var_bernoulli(p);
(%o4)     (1 - p) p

std_bernoulli (p) 
Function
Returns the standard deviation of a Bernoulli(p) random variable, with 0 < p < 1.
The Bernoulli(p) random variable is equivalent to the Binomial(1,p), therefore when Maxima has not enough information to get the result, a noun form based on the binomial standard deviation is returned.

(%i1) load (distrib)$
(%i2) std_bernoulli(p);
(%o2) std_binomial(1, p)
(%i3) assume(0<p,p<1)$ std_bernoulli(p);
(%o4) sqrt(1 - p) sqrt(p)
skewness_bernoulli \( (p) \)  
Returns the skewness coefficient of a \( Bernoulli(p) \) random variable, with \( 0 < p < 1 \).
The \( Bernoulli(p) \) random variable is equivalent to the \( Binomial(1, p) \), therefore when Maxima has not enough information to get the result, a noun form based on the binomial skewness coefficient is returned.

\[
\begin{align*}
(\%i1) \ & \text{load} \ (\text{distrib})$
\%
(\%i2) \ & \text{skewness_bernoulli}(p);
\%
(\%o2) \ & \text{skewness_binomial}(1, p)
\%
(\%i3) \ & \text{assume} (0<p, p<1)$ \text{skewness_bernoulli}(p);
\%
(\%o3) \ & \frac{1 - 2 \ p}{\sqrt{1 - p} \ \sqrt{p}}
\%
\end{align*}
\]

kurtosis_bernoulli \( (p) \)  
Returns the kurtosis coefficient of a \( Bernoulli(p) \) random variable, with \( 0 < p < 1 \).
The \( Bernoulli(p) \) random variable is equivalent to the \( Binomial(1, p) \), therefore when Maxima has not enough information to get the result, a noun form based on the binomial kurtosis coefficient is returned.

\[
\begin{align*}
(\%i1) \ & \text{load} \ (\text{distrib})$
\%
(\%i2) \ & \text{kurtosis_bernoulli}(p);
\%
(\%o2) \ & \text{kurtosis_binomial}(1, p)
\%
(\%i3) \ & \text{assume} (0<p, p<1)$ \text{kurtosis_bernoulli}(p);
\%
(\%o3) \ & \frac{1 - 6 \ (1 - p) \ p}{(1 - p) \ p}
\%
\end{align*}
\]

random_bernoulli \( (p) \)  
random_bernoulli \( (p,n) \)  
Returns a \( Bernoulli(p) \) random variate, with \( 0 < p < 1 \). Calling \text{random_bernoulli} with a second argument \( n \), a random sample of size \( n \) will be simulated.

This is a direct application of the \text{random} built-in Maxima function.

See also \text{random}. To make use of this function, write first \text{load(distrib)}.

pdf_geometric \( (x,p) \)  
Returns the value at \( x \) of the probability function of a \( Geometric(p) \) random variable, with \( 0 < p < 1 \). To make use of this function, write first \text{load(distrib)}.

cdf_geometric \( (x,p) \)  
Returns the value at \( x \) of the distribution function of a \( Geometric(p) \) random variable, with \( 0 < p < 1 \). To make use of this function, write first \text{load(distrib)}.

quantile_geometric \( (q,p) \)  
Returns the \( q \)-quantile of a \( Geometric(p) \) random variable, with \( 0 < p < 1 \); in other words, this is the inverse of \text{cdf_geometric}. Argument \( q \) must be an element of \([0, 1]\).
To make use of this function, write first \text{load(distrib)}.
**mean_geometric** ($p$) Function
Returns the mean of a $Geometric(p)$ random variable, with $0 < p < 1$. To make use of this function, write first `load(distrib)`.

**var_geometric** ($p$) Function
Returns the variance of a $Geometric(p)$ random variable, with $0 < p < 1$. To make use of this function, write first `load(distrib)`.

**std_geometric** ($p$) Function
Returns the standard deviation of a $Geometric(p)$ random variable, with $0 < p < 1$. To make use of this function, write first `load(distrib)`.

**skewness_geometric** ($p$) Function
Returns the skewness coefficient of a $Geometric(p)$ random variable, with $0 < p < 1$. To make use of this function, write first `load(distrib)`.

**kurtosis_geometric** ($p$) Function
Returns the kurtosis coefficient of a geometric random variable $Geo(p)$, with $0 < p < 1$. To make use of this function, write first `load(distrib)`.

**random_geometric** ($p$) Function
**random_geometric** ($p, n$) Function
Returns a $Geometric(p)$ random variate, with $0 < p < 1$. Calling `random_geometric` with a second argument $n$, a random sample of size $n$ will be simulated. The algorithm is based on simulation of Bernoulli trials. To make use of this function, write first `load(distrib)`.

**pdf_discrete_uniform** ($x, n$) Function
Returns the value at $x$ of the probability function of a $DiscreteUniform(n)$ random variable, with $n$ a strictly positive integer. To make use of this function, write first `load(distrib)`.

**cdf_discrete_uniform** ($x, n$) Function
Returns the value at $x$ of the distribution function of a $DiscreteUniform(n)$ random variable, with $n$ a strictly positive integer. To make use of this function, write first `load(distrib)`.

**quantile_discrete_uniform** ($q, n$) Function
Returns the $q$-quantile of a $DiscreteUniform(n)$ random variable, with $n$ a strictly positive integer; in other words, this is the inverse of `cdf_discrete_uniform`. Argument $q$ must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

**mean_discrete_uniform** ($n$) Function
Returns the mean of a $DiscreteUniform(n)$ random variable, with $n$ a strictly positive integer. To make use of this function, write first `load(distrib)`.
var_discrete_uniform (n)
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)
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)
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)
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)
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)
Returns the value at x of the probability function of a Hypergeometric(n1,n2,n) random variable, with n1, n2 and n non negative integers and n <= n1 + n2. To make use of this function, write first load(distrib).

cdf_hypergeometric (x,n1,n2,n)
Returns the value at x of the distribution function of a Hypergeometric(n1,n2,n) random variable, with n1, n2 and n non negative integers and n <= n1 + n2. To make use of this function, write first load(distrib).

quantile_hypergeometric (q,n1,n2,n)
Returns the q-quantile of a Hypergeometric(n1,n2,n) random variable, with n1, n2 and n non negative integers and n <= n1 + n2; in other words, this is the inverse of cdf_hypergeometric. Argument q must be an element of [0,1]. To make use of this function, write first load(distrib).

mean_hypergeometric (n1,n2,n)
Returns the mean of a discrete uniform random variable Hyp(n1,n2,n), with n1, n2 and n non negative integers and n <= n1 + n2. To make use of this function, write first load(distrib).
var_hypergeometric (n1,n2,n)
Returns the variance of a hypergeometric random variable Hyp(n1,n2,n), with n1, n2 and n non negative integers and n <= n1 + n2. To make use of this function, write first load(distrib).

std_hypergeometric (n1,n2,n)
Returns the standard deviation of a Hypergeometric(n1,n2,n) random variable, with n1, n2 and n non negative integers and n <= n1 + n2. To make use of this function, write first load(distrib).

skewness_hypergeometric (n1,n2,n)
Returns the skewness coefficient of a Hypergeometric(n1,n2,n) random variable, with n1, n2 and n non negative integers and n <= n1 + n2. To make use of this function, write first load(distrib).

kurtosis_hypergeometric (n1,n2,n)
Returns the kurtosis coefficient of a Hypergeometric(n1,n2,n) random variable, with n1, n2 and n non negative integers and n <= n1 + n2. To make use of this function, write first load(distrib).

random_hypergeometric (n1,n2,n)
random_hypergeometric (n1,n2,n,m)
Returns a Hypergeometric(n1,n2,n) random variate, with n1, n2 and n non negative integers and n <= n1 + n2. Calling random_hypergeometric with a fourth argument m, a random sample of size m will be simulated. Algorithm described in Kachitvichyanukul, V., Schmeiser, B.W. (1985) Computer generation of hypergeometric random variates. Journal of Statistical Computation and Simulation 22, 127-145. To make use of this function, write first load(distrib).

pdf_negative_binomial (x,n,p)
Returns the value at x of the probability function of a NegativeBinomial(n,p) random variable, with 0 < p < 1 and n a positive integer. To make use of this function, write first load(distrib).

cdf_negative_binomial (x,n,p)
Returns the value at x of the distribution function of a NegativeBinomial(n,p) random variable, with 0 < p < 1 and n a positive integer.
quantile_negative_binomial \((q,n,p)\)  
Returns the \(q\)-quantile of a \(\text{NegativeBinomial}(n,p)\) random variable, with \(0 < p < 1\) and \(n\) a positive integer; in other words, this is the inverse of \(\text{cdf_negative_binomial}\). Argument \(q\) must be an element of \([0,1]\). To make use of this function, write first \texttt{load(distrib)}.

mean_negative_binomial \((n,p)\)  
Returns the mean of a \(\text{NegativeBinomial}(n,p)\) random variable, with \(0 < p < 1\) and \(n\) a positive integer. To make use of this function, write first \texttt{load(distrib)}.

var_negative_binomial \((n,p)\)  
Returns the variance of a \(\text{NegativeBinomial}(n,p)\) random variable, with \(0 < p < 1\) and \(n\) a positive integer. To make use of this function, write first \texttt{load(distrib)}.

std_negative_binomial \((n,p)\)  
Returns the standard deviation of a \(\text{NegativeBinomial}(n,p)\) random variable, with \(0 < p < 1\) and \(n\) a positive integer. To make use of this function, write first \texttt{load(distrib)}.

skewness_negative_binomial \((n,p)\)  
Returns the skewness coefficient of a \(\text{NegativeBinomial}(n,p)\) random variable, with \(0 < p < 1\) and \(n\) a positive integer. To make use of this function, write first \texttt{load(distrib)}.

kurtosis_negative_binomial \((n,p)\)  
Returns the kurtosis coefficient of a \(\text{NegativeBinomial}(n,p)\) random variable, with \(0 < p < 1\) and \(n\) a positive integer. To make use of this function, write first \texttt{load(distrib)}.

random_negative_binomial \((n,p)\)  
random_negative_binomial \((n,p,m)\)  
Returns a \(\text{NegativeBinomial}(n,p)\) random variate, with \(0 < p < 1\) and \(n\) a positive integer. Calling \texttt{random_negative_binomial} with a third argument \(m\), a random sample of size \(m\) will be simulated.  
To make use of this function, write first \texttt{load(distrib)}.
Chapter 51: draw

51 draw

51.1 Introduction to draw

`draw` is a Maxima-Gnuplot interface.

There are three main functions to be used at Maxima level: `draw2d`, `draw3d` and `draw`. Follow this link for more elaborated examples of this package: http://riotorto.users.sourceforge.net/gnuplot

You need Gnuplot 4.2 or newer to run this program.

51.2 Functions and Variables for draw

51.2.1 Scenes

`gr2d` (graphic option, ..., graphic object, ...)

Scene constructor
Function `gr2d` builds an object describing a 2D scene. Arguments are `graphic options`, `graphic objects`, or lists containing both graphic options and objects. This scene is interpreted sequentially: `graphic options` affect those `graphic objects` placed on its right. Some `graphic options` affect the global 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`.

To make use of this object, write first `load(draw)`.

`gr3d` (graphic option, ..., graphic object, ...)

Scene constructor
Function `gr3d` builds an object describing a 3D scene. Arguments are `graphic options`, `graphic objects`, or lists containing both graphic options and objects. This scene is interpreted sequentially: `graphic options` affect those `graphic objects` placed on its right. Some `graphic options` affect the global 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`.

To make use of this object, write first `load(draw)`.
51.2.2 Functions

\textbf{draw} \ (gr2d, ..., gr3d, ..., options, ...)

Function

Plots a series of scenes; its arguments are \texttt{gr2d} and/or \texttt{gr3d} objects, together with some options, or lists of scenes and options. By default, the scenes are put together in one column.

Function \texttt{draw} accepts the following global options: \texttt{terminal}, \texttt{columns}, \texttt{dimensions}, \texttt{file\_name} and \texttt{delay}.

Functions \texttt{draw2d} and \texttt{draw3d} are short cuts to be used when only one scene is required, in two or three dimensions, respectively.

See also \texttt{gr2d} and \texttt{gr3d}.

To make use of this function, write first \texttt{load(draw)}.

Example:

\begin{verbatim}
(%i1) load(draw)$
(%i2) scene1: gr2d(title="Ellipse",
   nticks=30,
   parametric(2*cos(t),5*sin(t),t,0,2*%pi))$
(%i3) scene2: gr2d(title="Triangle",
   polygon([4,5,7],[6,4,2]))$
(%i4) draw(scene1, scene2, columns = 2)$
\end{verbatim}

The two draw sentences are equivalent:

\begin{verbatim}
(%i1) load(draw)$
(%i2) draw(gr3d(explicit(x^2+y^2,x,-1,1,y,-1,1)));
(%o2) [gr3d(explicit)]
(%i3) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1));
(%o3) [gr3d(explicit)]
\end{verbatim}

An animated gif file:

\begin{verbatim}
(%i1) load(draw)$
(%i2) draw(
   delay = 100,
   file_name = "zzz",
   terminal = 'animated_gif,
   gr2d(explicit(x^2,x,-1,1)),
   gr2d(explicit(x^3,x,-1,1)),
   gr2d(explicit(x^4,x,-1,1)));
\end{verbatim}

See also \texttt{gr2d}, \texttt{gr3d}, \texttt{draw2d} and \texttt{draw3d}.

\textbf{draw2d} \ (option, graphic\_object, ...)

Function

This function is a short cut for \texttt{draw(gr2d(options, ..., graphic\_object, ...))}.

It can be used to plot a unique scene in 2d.

To make use of this function, write first \texttt{load(draw)}.

See also \texttt{draw} and \texttt{gr2d}.
draw3d (option, graphic_object, ...) Function  
This function is a short cut for draw(gr3d(options, ..., graphic_object, ...)).
It can be used to plot a unique scene in 3d.
To make use of this function, write first load(draw).
See also draw and gr3d.

draw_file (graphic_option, ..., graphic_object, ...) Function  
Saves the current plot into a file. Accepted graphics options are: terminal, dimensions, file_name and background_color.
Example:
```
(%i1) load(draw)$
(%i2) /* screen plot */
    draw(gr3d(explicit(x^2+y^2,x,-1,1,y,-1,1)))$
(%i3) /* same plot in eps format */
    draw_file(terminal = eps,
        dimensions = [5,5])$
```

multiplot_mode (term) Function  
This function enables Maxima to work in one-window multiplot mode with terminal term; accepted arguments for this function are screen, wxt, aquaterm and none.
When multiplot mode is enabled, each call to draw sends a new plot to the same window, without erasing the previous ones. To disable the multiplot mode, write multiplot_mode(none).
When multiplot mode is enabled, global option terminal is blocked and you have to disable this working mode before changing to another terminal.
This feature does not work in Windows platforms.
Example:
```
(%i1) load(draw)$
(%i2) set_draw_defaults(
    xrange = [-1,1],
    yrange = [-1,1],
    grid = true,
    title = "Step by step plot" )$
(%i3) multiplot_mode(screen)$
(%i4) draw2d(color=blue, explicit(x^2,x,-1,1))$
(%i5) draw2d(color=red, explicit(x^3,x,-1,1))$
(%i6) draw2d(color=brown, explicit(x^4,x,-1,1))$
(%i7) multiplot_mode(none)$
```

set_draw_defaults (graphic_option, ..., graphic_object, ...) Function  
Sets user graphics options. This function is useful for plotting a sequence of graphics with common graphics options. Calling this function without arguments removes user defaults.
Example:
(%i1) load(draw)$
(%i2) set_draw_defaults(
    xrange = [-10,10],
    yrange = [-2, 2],
    color = blue,
    grid = true)$
(%i3) /* plot with user defaults */
    draw2d(explicit(((1+x)**2/(1+x*x))-1,x,-10,10))$
(%i4) set_draw_defaults()$
(%i5) /* plot with standard defaults */
    draw2d(explicit(((1+x)**2/(1+x*x))-1,x,-10,10))$

To make use of this function, write first load(draw).

51.2.3 Graphics options

adapt_depth

Graphic option

Default value: 10

adapt_depth is the maximum number of splittings used by the adaptive plotting routine.

This option is relevant only for 2d explicit functions.

axis_3d

Graphic option

Default value: true

If axis_3d is true, the x, y and z axis are shown in 3d scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

(%i1) load(draw)$
(%i2) draw3d(axis_3d = false,
            explicit(sin(x^2+y^2),x,-2,2,y,-2,2) )$

See also axis_bottom, axis_left, axis_top, and axis_right for axis in 2d.

axis_bottom

Graphic option

Default value: true

If axis_bottom is true, the bottom axis is shown in 2d scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

(%i1) load(draw)$
(%i2) draw2d(axis_bottom = false,
            explicit(x^3,x,-1,1))$

See also axis_left, axis_top, axis_right, and axis_3d.
**axis_left**  
Graphic option

Default value: **true**

If **axis_left** is **true**, the left axis is shown in 2d scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(axis_left = false,
             explicit(x^3,x,-1,1))$
```

See also **axis_bottom**, **axis_top**, **axis_right**, and **axis_3d**.

**axis_right**  
Graphic option

Default value: **true**

If **axis_right** is **true**, the right axis is shown in 2d scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(axis_right = false,
             explicit(x^3,x,-1,1))$
```

See also **axis_bottom**, **axis_left**, **axis_top**, and **axis_3d**.

**axis_top**  
Graphic option

Default value: **true**

If **axis_top** is **true**, the top axis is shown in 2d scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(axis_top = false,
             explicit(x^3,x,-1,1))$
```

See also **axis_bottom**, **axis_left**, **axis_right**, and **axis_3d**.

**background_color**  
Graphic option

Default value: **white**

Sets the background color for terminals **gif**, **png**, **jpg**, and **gif**. Default background color is white.

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:
• \texttt{gr2d}: polygon, rectangle, and ellipse.

Example:

\begin{verbatim}
(%i1) load(draw)$
(%i2) draw2d(color = brown,
       line_width = 8,
       polygon([[3,2],[7,2],[5,5]]),
       border = false,
       fill_color = blue,
       polygon([[5,2],[9,2],[7,5]]))$
\end{verbatim}

\textbf{cbrange} \hspace{1cm} \textit{Graphic option}

Default value: \texttt{auto}

If \texttt{cbrange} is \texttt{auto}, the range for the values which are colored when \texttt{enhanced3d} is not \texttt{false} is computed automatically. Values outside of the color range use color of the nearest extreme.

When \texttt{enhanced3d} or \texttt{colorbox} is \texttt{false}, option \texttt{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 \texttt{cbrange=[-2,3]}.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

\begin{verbatim}
(%i1) load(draw)$
(%i2) draw3d(
       enhanced3d = true,
       color = green,
       cbrange = [-3,10],
       cbrange = explicit(x^2+y^2, x,-2,2,y,-2,2))$
\end{verbatim}

See also \texttt{enhanced3d}, \texttt{colorbox} and \texttt{cbtics}.

\textbf{cbtics} \hspace{1cm} \textit{Graphic option}

Default value: \texttt{auto}

This graphic option controls the way tic marks are drawn on the colorbox when option \texttt{enhanced3d} is not \texttt{false}.

When \texttt{enhanced3d} or \texttt{colorbox} is \texttt{false}, option \texttt{cbtics} has no effect.

See \texttt{xtics} for a complete description.

Example:

\begin{verbatim}
(%i1) load(draw)$
(%i2) draw3d(
       enhanced3d = true,
       color = green,
       cbrange = explicit(x^2+y^2, x,-2,2,y,-2,2))$
\end{verbatim}

See also \texttt{enhanced3d}, \texttt{colorbox} and \texttt{cbrange}.
Chapter 51: draw

color

Default value: blue

color specifies the color for plotting lines, points, borders of polygons and labels.

Colors can be given as names or in hexadecimal rgb code.

Available color names are: white, black, gray0, grey0, gray10, grey10, gray20, grey20, gray30, grey30, gray40, grey40, gray50, grey50, gray60, grey60, gray70, grey70, gray80, grey80, gray90, grey90, gray100, grey100, gray, grey, light_gray, light_grey, dark_gray, dark_grey, red, light_red, dark_red, yellow, light_yellow, dark_yellow, green, light_green, dark_green, spring_green, forest_green, sea_green, blue, light_blue, dark_blue, midnight_blue, navy, medium_blue, royalblue, skyblue, cyan, light_cyan, dark_cyan, magenta, light_magenta, dark_magenta, turquoise, light_turquoise, dark_turquoise, pink, light_pink, dark_pink, coral, light_coral, orange_red, salmon, light_salmon, dark_salmon, aquamarine, khaki, dark_khaki, goldenrod, light_goldenrod, dark_goldenrod, gold, beige, brown, orange, dark_orange, violet, dark_violet, plum and purple.

Cromatic components in hexadecimal code are introduced in the form \"#rrggbb\".

Example:

(%i1) load(draw)$
(%i2) draw2d(explicit(x^2,x,-1,1), /* default is black */
    color = red,
    explicit(0.5 + x^2,x,-1,1),
    color = blue,
    explicit(1 + x^2,x,-1,1),
    color = light_blue,
    explicit(1.5 + x^2,x,-1,1),
    color = \"#23ab0f\",
    label(["This is a label",0,1.2]) )$

See also fill_color.

colorbox

Default value: true

If colorbox is true, a color scale without label is drawn together with image 2D objects, or coloured 3D objects. If colorbox is false, no color scale is shown. If colorbox is a string, a color scale with label is drawn.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

Color scale and images.

(%i1) load(draw)$
(%i2) im: apply('matrix,
makelist(makelist(random(200),i,1,30),i,1,30))$
(%i3) draw2d(image(im,0,0,30,30))$
(%i4) draw2d(colorbox = false, image(im,0,0,30,30))$

Color scale and 3D coloured object.
(%i1) load(draw)$
(%i2) draw3d(
   colorbox = "Magnitude",
   enhanced3d = true,
   explicit(x^2+y^2,x,-1,1,y,-1,1))$
See also palette.

columns

Graphic option

Default value: 1

columns is the number of columns in multiple plots.
Since this is a global graphics option, its position in the scene description does not
matter. It can be also used as an argument of function draw.
Example:

(%i1) load(draw)$
(%i2) scene1: gr2d(title="Ellipse",
   nticks=30,
   parametric(2*cos(t),5*sin(t),t,0,2*%pi))$
(%i3) scene2: gr2d(title="Triangle",
   polygon([4,5,7],[6,4,2]))$
(%i4) draw(scene1, scene2, columns = 2)$

contour

Graphic option

Default value: none

Option contour enables the user to select where to plot contour lines. Possible values are:
- none: no contour lines are plotted.
- base: contour lines are projected on the xy plane.
- surface: contour lines are plotted on the surface.
- both: two contour lines are plotted: on the xy plane and on the surface.
- map: contour lines are projected on the xy plane, and the view point is set just in the vertical.

Since this is a global graphics option, its position in the scene description does not
matter.
Example:

(%i1) load(draw)$
(%i2) draw3d(explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
    contour_levels = 15,
    contour = both,
    surface_hide = true) $

contour_levels

Graphic option

Default value: 5

This graphic option controls the way contours are drawn. contour_levels can be set to
a positive integer number, a list of three numbers or an arbitrary set of numbers:
• When option \texttt{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 \texttt{contour\_levels} is bounded to a list of length three of the form \([\text{lowest}, s, \text{highest}]\), contour lines are plotted from \texttt{lowest} to \texttt{highest} in steps of \(s\).

• When option \texttt{contour\_levels} is bounded to a set of numbers of the form \(\{n_1, n_2, \ldots\}\), contour lines are plotted at values \(n_1, n_2, \ldots\).

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.

\begin{verbatim}
(%i1) load(draw)$
(%i2) draw3d(color = green,
   explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
   contour_levels = 10,
   contour = both,
   surface_hide = true) $\end{verbatim}

From \(-8\) to \(8\) in steps of \(4\).

\begin{verbatim}
(%i1) load(draw)$
(%i2) draw3d(color = green,
   explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
   contour_levels = [-8,4,8],
   contour = both,
   surface_hide = true) $\end{verbatim}

Isolines at levels \(-7\), \(-6\), \(0.8\) and \(5\).

\begin{verbatim}
(%i1) load(draw)$
(%i2) draw3d(color = green,
   explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
   contour_levels = {-7, -6, 0.8, 5},
   contour = both,
   surface_hide = true) $\end{verbatim}

See also \texttt{contour}.

\textbf{data\_file\_name} \hspace{3cm} \textbf{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 \texttt{draw}.

See example in \texttt{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:

```maxima
(%i1) load(draw)$
(%i2) draw(
   delay = 100,
   file_name = "zzz",
   terminal = 'animated_gif,
   gr2d(explicit(x^2,x,-1,1)),
   gr2d(explicit(x^3,x,-1,1)),
   gr2d(explicit(x^4,x,-1,1)));

End of animation sequence
(%o2) [gr2d(explicit), gr2d(explicit), gr2d(explicit)]
```

Option `delay` is only active in animated gif's; it is ignored in any other case.

See also `terminal`, `dimensions`.

**dimensions**

Graphic option

Default value: `[600,500]`

Dimensions of the output terminal. Its value is a list formed by the width and the height. The meaning of the two numbers depends on the terminal you are working with.

With terminals `gif`, `animated_gif`, `png`, `jpg`, `svg`, `screen`, `wxt`, and `aquaterm`, the integers represent the number of points in each direction. If they are not integers, 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.

```maxima
(%i1) load(draw)$
(%i2) draw2d(
   dimensions = [300,300],
   terminal = 'png,
   explicit(x^4,x,-1,1))$
(%i3) draw2d(
   dimensions = [300,300],
   terminal = 'wxt,
   explicit(x^4,x,-1,1))$
```

Option `dimensions` applied to `eps` output. We want an `eps` file with A4 portrait dimensions.
Chapter 51: draw

(%i1) load(draw)$
(%i2) A4portrait: 100*[21, 29.7]$  
(%i3) draw3d(
    dimensions = A4portrait,  
    terminal = 'eps,  
    explicit(x^2-y^2,x,-2,2,y,-2,2))

enhanced3d

Graphic option

Default value: none

If enhanced3d is none, surfaces are not colored in 3D plots. In order to get a colored surface, a list must be assigned to option enhanced3d, where the first element is an expression and the rest are the names of the variables or parameters used in that expression. A list such \([f(x,y,z), x, y, z]\) means that point \([x,y,z]\) of the surface is assigned number \(f(x,y,z)\), which will be colored according to the actual palette. For those 3D graphic objects defined in terms of parameters, it is possible to define the color number in terms of the parameters, as in \([f(u), u]\), as in objects parametric and tube, or \([f(u,v), u, v]\), as in object parametric_surface. While all 3D objects admit the model based on absolute coordinates, \([f(x,y,z), x, y, z]\), only two of them, namely explicit and elevation_grid, accept also models defined on the \([x,y]\) coordinates, \([f(x,y), x, y]\). 3D graphic object implicit accepts only the \([f(x,y,z), x, y, z]\) model. Object points accepts also the \([f(x,y,z), x, y, z]\) model, but when points have a chronological nature, model \([f(k), k]\) is also valid, being \(k\) an ordering parameter.

When enhanced3d is assigned something different to none, options color and surface_hide are ignored.

The names of the variables defined in the lists may be different to those used in the definitions of the graphic objects.

In order to maintain back compatibility, enhanced3d = false is equivalent to enhanced3d = none, and enhanced3d = true is equivalent to enhanced3d = [z, x, y, z]. If an expression is given to enhanced3d, its variables must be the same used in the surface definition. This is not necessary when using lists.

See option palette to learn how palettes are specified.

Examples:

explicit object with coloring defined by the \([f(x,y,z), x, y, z]\) model.

(%i1) load(draw)$
(%i2) draw3d(
    enhanced3d = [x-z/10,x,y,z],
    palette = gray,
    explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3))$

explicit object with coloring defined by the \([f(x,y), x, y]\) model. The names of the variables defined in the lists may be different to those used in the definitions of the graphic objects; in this case, \(r\) corresponds to \(x\), and \(s\) to \(y\).

(%i1) load(draw)$
(%i2) draw3d(
    enhanced3d = [sin(r*s),r,s],
\texttt{explicit}(20*\exp(-x^2-y^2)-10,x,-3,3,y,-3,3))$

\textbf{parametric} object with coloring defined by the \([f(x,y,z), x, y, z]\) model.

\begin{verbatim}
(%i1) load(draw)$
(%i2) draw3d(
    nticks = 100,
    line_width = 2,
    enhanced3d = [if y>= 0 then 1 else 0, x, y, z],
    parametric(sin(u)^2,cos(u),u,u,0,4*%pi))$
\end{verbatim}

\textbf{parametric} object with coloring defined by the \([f(u), u]\) model. In this case, \((u-1)^2\) is a shortcut for \([(u-1)^2,u]\).

\begin{verbatim}
(%i1) load(draw)$
(%i2) draw3d(
    nticks = 60,
    line_width = 3,
    enhanced3d = (u-1)^2,
    parametric(cos(5*u)^2,sin(7*u),u-2,u,0,2))$
\end{verbatim}

\textbf{elevation_grid} object with coloring defined by the \([f(x,y), x, y]\) model.

\begin{verbatim}
(%i1) load(draw)$
(%i2) m: apply(
    matrix,
    makelist(makelist(cos(i^2/80-k/30),k,1,30),i,1,20))
(%i3) draw3d(
    enhanced3d = [cos(x*y*10),x,y],
    elevation_grid(m,-1,-1,2,2),
    xlabel = "x",
    ylabel = "y");
\end{verbatim}

\textbf{tube} object with coloring defined by the \([f(x,y,z), x, y, z]\) model.

\begin{verbatim}
(%i1) load(draw)$
(%i2) draw3d(
    enhanced3d = [cos(x-y),x,y,z],
    palette = gray,
    xu_grid = 50,
    tube(cos(a), a, 0, 1, a, 0, 4*%pi) )$
\end{verbatim}

\textbf{tube} object with coloring defined by the \([f(u), u]\) model. Here, \texttt{enhanced3d = -a} would be the shortcut for \texttt{enhanced3d = [-foo,foo]}.

\begin{verbatim}
(%i1) load(draw)$
(%i2) draw3d(
    tube_extremes = [open, closed],
    palette = [26,15,-2],
    enhanced3d = [-foo, foo],
    tube(a, a, a^2, 1, a, -2, 2) )$
\end{verbatim}

\textbf{implicit} and \textbf{points} objects with coloring defined by the \([f(x,y,z), x, y, z]\) model.

\begin{verbatim}
(%i1) load(draw)$
(%i2) draw3d(
    enhanced3d = [x-y,x,y,z],
\end{verbatim}
Chapter 51: draw

\[ \text{implicit}\left((x^2+y^2+z^2-1)\left(x^2+(y-1.5)^2+z^2-0.5\right)=0.015, \right. \\
\left. x,-1,1,y,-1.2,2.3,z,-1,1)\right) \]

(%i3) m: makelist([random(1.0),random(1.0),random(1.0)],k,1,2000)$
(%i4) draw3d(
    point_type = filled_circle,
    point_size = 2,
    enhanced3d = [u+v-w,u,v,w],
    points(m) )$

When points have a chronological nature, model \([f(k), k]\) is also valid, being \(k\) an ordering parameter.

(%i1) load(draw)$
(%i2) m:makelist([random(1.0), random(1.0), random(1.0)],k,1,5)$
(%i3) draw3d(
    enhanced3d = [sin(j), j],
    point_size = 3,
    point_type = filled_circle,
    points_joined = true,
    points(m) )$

\text{error_type} \quad \text{Graphic option}

Default value: \(y\)

Depending on its value, which can be \(x\), \(y\), or \(xy\), graphic object \text{errors} will draw points with horizontal, vertical, or both, error bars. When \text{error_type=boxes}, boxes will be drawn instead of crosses.

See also \text{errors}.

\text{file_name} \quad \text{Graphic option}

Default value: "maxima_out"

This is the name of the file where terminals \text{png}, \text{jpg}, \text{gif}, \text{eps}, \text{eps_color}, \text{pdf}, \text{pdfcairo} and \text{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 \text{draw}.

Example:

(%i1) load(draw)$
(%i2) draw2d(file_name = "myfile",
    explicit(x^2,x,-1,1),
    terminal = 'png)$

See also \text{terminal}, \text{dimensions}.

\text{fill_color} \quad \text{Graphic option}

Default value: "red"

\text{fill_color} specifies the color for filling polygons and 2d \text{explicit} functions.

See \text{color} to learn how colors are specified.
fill_density

Graphic option

Default value: 0

fill_density is a number between 0 and 1 that specifies the intensity of the fill_color in bars objects.

See bars for examples.

filled_func

Graphic option

Default value: false

Option filled_func controls how regions limited by functions should be filled. When filled_func is true, the region bounded by the function defined with object explicit and the bottom of the graphic window is filled with fill_color. When filled_func contains a function expression, then the region bounded by this function and the function defined with object explicit will be filled. By default, explicit functions are not filled.

This option affects only the 2d graphic object explicit.

Example:

Region bounded by an explicit object and the bottom of the graphic window.

(%i1) load(draw)$
(%i2) draw2d(fill_color = red,
       filled_func = true,
       explicit(sin(x),x,0,10) )$

Region bounded by an explicit object and the function defined by option filled_func. Note that the variable in filled_func must be the same as that used in explicit.

(%i1) load(draw)$
(%i2) draw2d(fill_color = grey,
       filled_func = sin(x),
       explicit(-sin(x),x,0,%pi));

See also fill_color and explicit.

font

Graphic option

Default value: "" (empty string)

This option can be used to set the font face to be used by the terminal. Only one font face and size can be used throughout the plot.

Since this is a global graphics option, its position in the scene description does not matter.

See also font_size.

Gnuplot doesn’t handle fonts by itself, it leaves this task to the support libraries of the different terminals, each one with its own philosophy about it. A brief summary follows:

- x11: Uses the normal x11 font server mechanism.

Example:
Chapter 51: draw

679

(%i1) load(draw)$
(%i2) draw2d(font = "Arial",
font_size = 20,
label(["Arial font, size 20",1,1]))$

• **windows**: The windows terminal doesn’t support changing of fonts from inside the plot. Once the plot has been generated, the font can be changed right-clicking on the menu of the graph window.

• **png, jpeg, gif**: The `libgd` library uses the font path stored in the environment variable `GDFONTPATH`; in this case, it is only necessary to set option `font` to the font’s name. It is also possible to give the complete path to the font file.

Examples:

Option `font` can be given the complete path to the font file:

```lisp
(%i1) load(draw)$
(%i2) path: "/usr/share/fonts/truetype/freefont/"$
(%i3) file: "FreeSerifBoldItalic.ttf"$
(%i4) draw2d(
    font = concat(path, file),
    font_size = 20,
    color = red,
    label(["FreeSerifBoldItalic font, size 20",1,1]),
    terminal = png)$
```

If environment variable `GDFONTPATH` is set to the path where font files are allocated, it is possible to set graphic option `font` to the name of the font.

```lisp
(%i1) load(draw)$
(%i2) draw2d(
    font = "FreeSerifBoldItalic",
    font_size = 20,
    color = red,
    label(["FreeSerifBoldItalic font, size 20",1,1]),
    terminal = png)$
```


Example:

```lisp
(%i1) load(draw)$
(%i2) draw2d(
    font = "Courier-Oblique",
    font_size = 15,
    label(["Courier-Oblique font, size 15",1,1]),
    terminal = eps)$
```

• **pdf**: Uses same fonts as Postscript.

• **pdfcairo**: Uses same fonts as wxt.

• **wxt**: The `pango` library finds fonts via the `fontconfig` utility.

• **aqua**: Default is "Times-Roman".
The gnuplot documentation is an important source of information about terminals and fonts.

**font_size**

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**

Default value: "maxout.gnuplot"

This is the name of the file with the necessary commands to be processed by Gnuplot.

Since this is a global graphics option, its position in the scene description does not matter. It can be also used as an argument of function `draw`.

Example:

```lisp
(%i1) load(draw)$
(%i2) draw2d(
    file_name = "my_file",
    gnuplot_file_name = "my_commands_for_gnuplot",
    data_file_name = "my_data_for_gnuplot",
    terminal = png,
    explicit(x^2,x,-1,1))$
```

See also `data_file_name`.

**grid**

Default value: false

If `grid` is true, a grid will be drawn on the xy plane.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```lisp
(%i1) load(draw)$
(%i2) draw2d(grid = true,
    explicit(exp(u),u,-2,2))$
```

**head_angle**

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:
Chapter 51: draw

(%i1) load(draw)$
(%i2) draw2d(xrange = [0,10],
   yrange = [0,9],
   head_length = 0.7,
   head_angle = 10,
   vector([1,1],[0,6]),
   head_angle = 20,
   vector([2,1],[0,6]),
   head_angle = 30,
   vector([3,1],[0,6]),
   head_angle = 40,
   vector([4,1],[0,6]),
   head_angle = 60,
   vector([5,1],[0,6]),
   head_angle = 90,
   vector([6,1],[0,6]),
   head_angle = 120,
   vector([7,1],[0,6]),
   head_angle = 160,
   vector([8,1],[0,6]),
   head_angle = 180,
   vector([9,1],[0,6]) )$

See also head_both, head_length, and head_type.

head_both

Graphic option

Default value: false

If head_both is true, vectors are plotted with two arrow heads. If false, only one arrow is plotted.

This option is relevant only for vector objects.

Example:

(%i1) load(draw)$
(%i2) draw2d(xrange = [0,8],
   yrange = [0,8],
   head_length = 0.7,
   vector([1,1],[6,0]),
   head_both = true,
   vector([1,7],[6,0]) )$

See also head_length, head_angle, and head_type.

head_length

Graphic option

Default value: 2

head_length indicates, in x-axis units, the length of arrow heads.

This option is relevant only for vector objects.

Example:

(%i1) load(draw)$
(%i2) draw2d(xrange = [0,12],


```
yrange = [0,8],
vector([0,1],[5,5]),
head_length = 1,
vector([2,1],[5,5]),
head_length = 0.5,
vector([4,1],[5,5]),
head_length = 0.25,
vector([6,1],[5,5]))$
```

See also head_both, head_angle, and head_type.

**head_type**

Graphic option

Default value: filled

`head_type` is used to specify how arrow heads are plotted. Possible values are:
- `filled` (closed and filled arrow heads),
- `empty` (closed but not filled arrow heads),
- `nofilled` (open arrow heads).

This option is relevant only for `vector` objects.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(xrange = [0,12],
yrange = [0,10],
head_length = 1,
vector([0,1],[5,5]), /* default type */
head_type = 'empty,
vector([3,1],[5,5]),
head_type = 'nofilled,
vector([6,1],[5,5]))$
```

See also head_both, head_angle, and head_length.

**ip_grid**

Graphic option

Default value: [50, 50]

`ip_grid` sets the grid for the first sampling in implicit plots.

This option is relevant only for `implicit` objects.

**ip_grid_in**

Graphic option

Default value: [5, 5]

`ip_grid_in` sets the grid for the second sampling in implicit plots.

This option is relevant only for `implicit` objects.

**key**

Graphic option

Default value: "" (empty string)

`key` is the name of a function in the legend. If `key` is an empty string, no key is assigned to the function.

This option affects the following graphic objects:
- `gr2d`: points, polygon, rectangle, ellipse, vector, explicit, implicit, parametric, and polar.
• gr3d: points, explicit, parametric, and parametric_surface.

Example:

```lisp
(%i1) load(draw)$
(%i2) draw2d(key = "Sinus",
            explicit(sin(x),x,0,10),
            key = "Cosinus",
            color = red,
            explicit(cos(x),x,0,10) )$
```

**label_alignment**

Graphic option

Default value: center

`label_alignment` is used to specify where to write labels with respect to the given coordinates. Possible values are: center, left, and right. This option is relevant only for label objects.

Example:

```lisp
(%i1) load(draw)$
(%i2) draw2d(xrange = [0,10],
            yrange = [0,10],
            points_joined = true,
            points([[5,0],[5,10]]),
            color = blue,
            label("Centered alignment (default)",5,2),
            label_alignment = 'left,
            label("Left alignment",5,5),
            label_alignment = 'right,
            label("Right alignment",5,8))$
```

See also `label_orientation`, and color.

**label_orientation**

Graphic option

Default value: horizontal

`label_orientation` is used to specify orientation of labels. Possible values are: horizontal, and vertical. This option is relevant only for label objects.

Example:

In this example, a dummy point is added to get an image. Package draw needs always data to draw an scene.

```lisp
(%i1) load(draw)$
(%i2) draw2d(xrange = [0,10],
            yrange = [0,10],
            point_size = 0,
            points([[5,5]]),
            color = navy,
            label("Horizontal orientation (default)",5,2),
            label_orientation = 'vertical,
            color = "#654321",
```
\begin{verbatim}
label(["Vertical orientation",1,5])

See also \texttt{label\_alignment} and \texttt{color}.

\textbf{line\_type} \hfill Graphic option

Default value: \texttt{solid}

line\_type indicates how lines are displayed; possible values are \texttt{solid} and \texttt{dots}.

This option affects the following graphic objects:

- \texttt{gr2d}: points, polygon, rectangle, ellipse, vector, explicit, implicit, parametric and polar.
- \texttt{gr3d}: points, explicit, parametric and \texttt{parametric\_surface}.

Example:
\begin{verbatim}
(%i1) load(draw)$
(%i2) draw2d(line\_type = dots,
    explicit(1 + x^2,x,-1,1),
    line\_type = solid, /* default */
    explicit(2 + x^2,x,-1,1))$
\end{verbatim}

See also \texttt{line\_width}.

\textbf{line\_width} \hfill 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:

- \texttt{gr2d}: points, polygon, rectangle, ellipse, vector, explicit, implicit, parametric and polar.
- \texttt{gr3d}: points and \texttt{parametric}.

Example:
\begin{verbatim}
(%i1) load(draw)$
(%i2) draw2d(explicit(x^2,x,-1,1), /* default width */
    line\_width = 5.5,
    explicit(1 + x^2,x,-1,1),
    line\_width = 10,
    explicit(2 + x^2,x,-1,1))$
\end{verbatim}

See also \texttt{line\_type}.

\textbf{logcb} \hfill Graphic option

Default value: \texttt{false}

If \texttt{logcb} is \texttt{true}, the tics in the colorbox will be drawn in the logarithmic scale.

When \texttt{enhanced3d} or \texttt{colorbox} is \texttt{false}, option \texttt{logcb} has no effect.

Since this is a global graphics option, its position in the scene description does not matter.

Example:
\end{verbatim}
(\%i1) load(draw)$
(\%i2) draw3d (  
  enhanced3d = true,  
  color = green,  
  logcb = true,  
  logz = true,  
  palette = [-15,24,-9],  
  explicit(exp(x^2-y^2), x,-2,2,y,-2,2))$

See also enhanced3d, colorbox and cbrange.

logx  
Graphic option  
Default value: false  
If logx is true, the x axis will be drawn in the logarithmic scale.  
Since this is a global graphics option, its position in the scene description does not matter.  
Example:  
(\%i1) load(draw)$
(\%i2) draw2d(explicit(log(x),x,0.01,5),  
          logx = true)$

See also logy and logz.

logy  
Graphic option  
Default value: false  
If logy is true, the y axis will be drawn in the logarithmic scale.  
Since this is a global graphics option, its position in the scene description does not matter.  
Example:  
(\%i1) load(draw)$
(\%i2) draw2d(logy = true,  
          explicit(exp(x),x,0,5))$

See also logx and logz.

logz  
Graphic option  
Default value: false  
If logz is true, the z axis will be drawn in the logarithmic scale.  
Since this is a global graphics option, its position in the scene description does not matter.  
Example:  
(\%i1) load(draw)$
(\%i2) draw3d(logz = true,  
          explicit(exp(u^2+v^2),u,-2,2,v,-2,2))$

See also logx and logy.
nticks

Default value: 29

In 2d, nticks gives the initial number of points used by the adaptive plotting routine for explicit objects. It is also the number of points that will be shown in parametric and polar curves.

This option affects the following graphic objects:

- gr2d: ellipse, explicit, parametric and polar.
- gr3d: parametric.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(transparent = true, ellipse(0,0,4,2,0,180), nticks = 5, ellipse(0,0,4,2,180,180) )$
```

palette

Default value: color

palette indicates how to map gray levels onto color components. It works together with option enhanced3d in 3D graphics, who associates every point of a surfaces to a real number or gray level. It also works with gray images. With palette, levels are transformed into colors.

There are two ways for defining these transformations.

First, palette can be a vector of length three with components ranging from -36 to +36; each value is an index for a formula mapping the levels onto red, green and blue colors, respectively:

0: 0  1: 0.5  2: 1
3: x  4: x^2  5: x^3
6: x^4  7: sqrt(x)  8: sqrt(sqrt(x))
9: sin(90x)  10: cos(90x)  11: |x-0.5|
12: (2x-1)^2  13: sin(180x)  14: |cos(180x)|
15: sin(360x)  16: cos(360x)  17: |sin(360x)|
18: |cos(360x)|  19: |sin(720x)|  20: |cos(720x)|
21: 3x  22: 3x-1  23: 3x-2
24: |3x-1|  25: |3x-2|  26: (3x-1)/2
27: (3x-2)/2  28: |(3x-1)/2|  29: |(3x-2)/2|
30: x/0.32-0.78125  31: 2*x-0.84  32: 4x;1-2x+1.84;x/0.08-11.5
33: |2*x - 0.5|  34: 2*x  35: 2*x - 0.5
36: 2*x - 1

negative numbers mean negative colour component. palette = gray and palette = color are short cuts for palette = [3,3,3] and palette = [7,5,15], respectively.

Second, palette can be a user defined lookup table. In this case, the format for building a lookup table of length n is palette=[color_1, color_2, ..., color_n], where color_i is a well formed color (see option color), such that color_1 is assigned to the lowest gray level and color_n to the highest. The rest of colors are interpolated.
Since this is a global graphics option, its position in the scene description does not matter.

Examples:
It works together with option enhanced3d in 3D graphics.

```
(%i1) load(draw)$
(%i2) draw3d(
   enhanced3d = [z-x+2*y,x,y,z],
   palette = [32, -8, 17],
   explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3))$
```

It also works with gray images.

```
(%i1) load(draw)$
(%i2) im: apply(
   'matrix,
   makelist(makelist(random(200),i,1,30),i,1,30))$
(%i3) /* palette = color, default */
   draw2d(image(im,0,0,30,30))$
(%i4) draw2d(palette = gray, image(im,0,0,30,30))$
(%i5) draw2d(palette = [15,20,-4],
   colorbox=false,
   image(im,0,0,30,30))$
```

palette can be a user defined lookup table. In this example, low values of x are colored in red, and higher values in yellow.

```
(%i1) load(draw)$
(%i2) draw3d(
   palette = [red, blue, yellow],
   enhanced3d = x,
   explicit(x^2+y^2,x,-1,1,y,-1,1)) $
```

See also colorbox and enhanced3d.

**point_size**

Graphic option

Default value: 1

point_size sets the size for plotted points. It must be a non negative number.

This option has no effect when graphic option point_type is set to dot.

This option affects the following graphic objects:

- gr2d: points.
- gr3d: points.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(points(makelist([random(20),random(50)],k,1,10)),
   point_size = 5,
   points(makelist(k,k,1,20),makelist(random(30),k,1,20)))$
```

**point_type**

Graphic option

Default value: 1
point_type indicates how isolated points are displayed; the value of this option can be any integer index greater or equal than -1, or the name of a point style: none (-1), dot (0), plus (1), multiply (2), asterisk (3), square (4), filled_square (5), circle (6), filled_circle (7), up_triangle (8), filled_up_triangle (9), down_triangle (10), filled_down_triangle (11), diamant (12) and filled_diamant (13).

This option affects the following graphic objects:

- gr2d: points.
- gr3d: points.

Example:

```maxima
(%i1) load(draw)$
(%i2) draw2d(xrange = [0,10],
yrange = [0,10],
point_size = 3,
point_type = diamant,
points([[1,1],[5,1],[9,1]]),
point_type = filled_down_triangle,
points([[1,2],[5,2],[9,2]]),
point_type = asterisk,
points([[1,3],[5,3],[9,3]]),
point_type = filled_diamant,
points([[1,4],[5,4],[9,4]]),
point_type = 5,
points([[1,5],[5,5],[9,5]]),
point_type = 6,
points([[1,6],[5,6],[9,6]]),
point_type = filled_circle,
points([[1,7],[5,7],[9,7]]),
point_type = 8,
points([[1,8],[5,8],[9,8]]),
point_type = filled_diamant,
points([[1,9],[5,9],[9,9]]) )$
```

pointsJoined

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:

```maxima
(%i1) load(draw)$
(%i2) draw2d(xrange = [0,10],
yrange = [0,4],
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]]))$
```
point_type = up_triangle,
color = blue,
points([[1,1],[5,1],[9,1]]),
points_joined = true,
point_type = square,
line_type = dots,
points([[1,2],[5,2],[9,2]]),
point_type = circle,
color = red,
line_width = 7,
points([[1,3],[5,3],[9,3]])

proportional_axes

Default value: none

When proportional_axes is equal to xy or xyz, a 2D or 3D scene will be drawn with axes proportional to their relative lengths.

Since this is a global graphics option, its position in the scene description does not matter.

This option works with Gnuplot version 4.2.6 or greater.

Examples:

Single 2D plot.

(%i1) load(draw)$
(%i2) draw2d(  
  ellipse(0,0,1,1,0,360),
  transparent=true,
  color = blue,
  line_width = 4,
  ellipse(0,0,2,1/2,0,360),
  proportional_axes = xy) $

Multiplot.

(%i1) load(draw)$
(%i2) draw(  
  terminal = wxt,
  gr2d(proportional_axes = xy,
       explicit(x^2,x,0,1)),
  gr2d(explicit(x^2,x,0,1),
       xrange = [0,1],
       yrange = [0,2],
       proportional_axes=xy),
  gr2d(explicit(x^2,x,0,1))

surface_hide

Default value: false

If surface_hide is true, hidden parts are not plotted in 3d surfaces.

Since this is a global graphics option, its position in the scene description does not matter.
Example:
(\%i1) load(draw)$
(\%i2) draw(columns=2,
   gr3d(explicit(exp(sin(x)+cos(x^2)),x,-3,3,y,-3,3)),
   gr3d(surface_hide = true,
      explicit(exp(sin(x)+cos(x^2)),x,-3,3,y,-3,3)) )$

\textbf{terminal}

Graphic option

Default value: screen

Selects the terminal to be used by Gnuplot; possible values are: screen (default),
png, pngcairo, jpg, eps, eps_color, pdf, pdfcairo, gif, animated_gif, wxt, svg,
and aquaterm.

Terminals screen, wxt and aquaterm can be also defined as a list with two elements:
the name of the terminal itself and a non negative integer number. In this form, multiple windows can be opened at the same time, each with its corresponding number.
This feature does not work in Windows platforms.

Since this is a global graphics option, its position in the scene description does not matter. It can be also used as an argument of function draw.

N.B. pdfcairo requires Gnuplot 4.3 or newer. pdf requires Gnuplot to be compiled with the option --enable-pdf and libpdf must be installed. The pdf library is available from: http://www.pdflib.com/en/download/pdflib-family/pdflib-lite/

Examples:
(\%i1) load(draw)$
(\%i2) /* screen terminal (default) */
   draw2d(explicit(x^2,x,-1,1))$
(\%i3) /* png file */
   draw2d(terminal = 'png,
          explicit(x^2,x,-1,1))$
(\%i4) /* jpg file */
   draw2d(terminal = 'jpg,
          dimensions = [300,300],
          explicit(x^2,x,-1,1))$
(\%i5) /* eps file */
   draw2d(file_name = "myfile",
          explicit(x^2,x,-1,1),
          terminal = 'eps)$
(\%i6) /* pdf file */
   draw2d(file_name = "mypdf",
          dimensions = 100*[12.0,8.0],
          explicit(x^2,x,-1,1),
          terminal = 'pdf)$
(\%i7) /* wxwidgets window */
   draw2d(explicit(x^2,x,-1,1),
          terminal = 'wxt)$

Multiple windows.
(\%i1) load(draw)$
Chapter 51: draw

(%i2) draw2d(explicit(x^5,x,-2,2), terminal=[screen, 3])$
(%i3) draw2d(explicit(x^2,x,-2,2), terminal=[screen, 0])$

An animated gif file.

(%i1) load(draw)$
(%i2) draw(
    delay = 100,
    file_name = "zzz",
    terminal = 'animated_gif,
    gr2d(explicit(x^2,x,-1,1)),
    gr2d(explicit(x^3,x,-1,1)),
    gr2d(explicit(x^4,x,-1,1)))

End of animation sequence
(%o2) [gr2d(explicit), gr2d(explicit), gr2d(explicit)]

Option delay is only active in animated gif's; it is ignored in any other case.

See also file_name, dimensions and delay.

title

Graphic option

Default value: "" (empty string)

Option title, a string, is the main title for the scene. By default, no title is written.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

(%i1) load(draw)$
(%i2) draw2d(explicit(exp(u),u,-2,2),
    title = "Exponential function")$

transform

Graphic option

Default value: none

If transform is none, the space is not transformed and graphic objects are drawn as defined. When a space transformation is desired, a list must be assigned to option transform. In case of a 2D scene, the list takes the form [f1(x,y), f2(x,y), x, y]. In case of a 3D scene, the list is of the form [f1(x,y,z), f2(x,y,z), f3(x,y,z), x, y, z].

The names of the variables defined in the lists may be different to those used in the definitions of the graphic objects.

Examples:

Rotation in 2D.

(%i1) load(draw)$
(%i2) th : %pi / 4$
(%i3) draw2d(
    color = "#e245f0",
    proportional_axes = 'xy,
    line_width = 8,
    triangle([3,2],[7,2],[5,5]),
    border = false,
fill_color = yellow,
transform = [cos(th)*x - sin(th)*y,
            sin(th)*x + cos(th)*y, x, y],
triangle([3,2],[7,2],[5,5]) )$

Translation in 3D.
(%i1) load(draw)$
(%i2) draw3d(
    color = "#a02c00",
    explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3),
    transform = [x+10,y+10,z+10,x,y,z],
    color = blue,
    explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3) )$

**transparent**

Default value: false

If transparent is true, interior regions of polygons are filled according to fill_color.

This option affects the following graphic objects:

- gr2d: polygon, rectangle, and ellipse.

Example:
(%i1) load(draw)$
(%i2) draw2d(polygon([[3,2],[7,2],[5,5]]),
             transparent = true,
             color = blue,
             polygon([[5,2],[9,2],[7,5]]) )$

**tube_extremes**

Default value: [open, open]

A list with two possible elements, open and closed, indicating whether the extremes of a graphic object tube remain open or must be closed. By default, both extremes are left open.

Example:
(%i1) load(draw)$
(%i2) draw3d(  
    tube_extremes = [open, closed],
    tube(0, 0, a, 1,
         a, 0, 8) )$

**unit_vectors**

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:
Chapter 51: draw

(%i1) load(draw)$
(%i2) draw2d(xrange = [-1,6],
yrange = [-1,6],
head_length = 0.1,
vector([0,0],[5,2]),
unit_vectors = true,
color = red,
vector([0,3],[5,2]))$

user_preamble
Graphic option
Default value: "" (empty string)
Expert Gnuplot users can make use of this option to fine tune Gnuplot’s behaviour by writing settings to be sent before the plot or splot command.
The value of this option must be a string or a list of strings (one per line).
Since this is a global graphics option, its position in the scene description does not matter.
Example:
The *dumb* terminal is not supported by package draw, but it is possible to set it by making use of option user_preamble,

(%i1) load(draw)$
(%i2) draw2d(explicit(exp(x)-1,x,-1,1),
parametric(cos(u),sin(u),u,0,2*%pi),
user_preamble="set terminal dumb")$

view
Graphic option
Default value: [60,30]
A pair of angles, measured in degrees, indicating the view direction in a 3D scene.
The first angle is the vertical rotation around the x axis, in the range [0,180]. The second one is the horizontal rotation around the z axis, in the range [0,360].
Since this is a global graphics option, its position in the scene description does not matter.
Example:

(%i1) load(draw)$
(%i2) draw3d(view = [170, 360],
explicit(sin(x^2+y^2),x,-2,2,y,-2,2) )$

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:

```maxima
(%i1) load(draw)$
(%i2) draw2d(explicit(x^3,x,-1,1),
   xaxis = true,
   xaxis_color = blue)$
```

See also `xaxis_width`, `xaxis_type` and `xaxis_color`.

**xaxis_color**

Graphic option

Default value: "black"

*xaxis_color* specifies the color for the x axis. See `color` to know how colors are defined.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```maxima
(%i1) load(draw)$
(%i2) draw2d(explicit(x^3,x,-1,1),
   xaxis = true,
   xaxis_color = red)$
```

See also `xaxis`, `xaxis_width` and `xaxis_type`.

**xaxis_secondary**

Graphic option

Default value: false

If *xaxis_secondary* is true, function values can be plotted with respect to the second x axis, which will be drawn on top of the scene.

Note that this is a local graphics option which only affects to 2d plots.

Example:

```maxima
(%i1) load(draw)$
(%i2) draw2d(key = "Bottom x-axis",
   explicit(x+1,x,1,2),
   color = red,
   key = "Above x-axis",
   xtics_secondary = true,
   xaxis_secondary = true,
   explicit(x^2,x,-1,1))$
```

See also `xrange_secondary`, `xtics_secondary`, `xtics_rotate_secondary`, `xtics_axis_secondary` and `xaxis_secondary`.

**xaxis_type**

Graphic option

Default value: dots

*xaxis_type* indicates how the x axis is displayed; possible values are *solid* and *dots*. 
Since this is a global graphics option, its position in the scene description does not matter.

Example:

```lisp
(%i1) load(draw)$
(%i2) draw2d(explicit(x^3,x,-1,1),
   xaxis = true,
   xaxis_type = solid)$
```

See also `xaxis`, `xaxis_width` and `xaxis_color`.

**xaxis_width**

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:

```lisp
(%i1) load(draw)$
(%i2) draw2d(explicit(x^3,x,-1,1),
   xaxis = true,
   xaxis_width = 3)$
```

See also `xaxis`, `xaxis_type` and `xaxis_color`.

**xlabel**

Default value: "" (empty string)

Option `xlabel`, a string, is the label for the x axis. By default, no label is written.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```lisp
(%i1) load(draw)$
(%i2) draw2d(xlabel = "Time",
   explicit(exp(u),u,-2,2),
   ylabel = "Population")$
```

See also `ylabel`, and `zlabel`.

**xrange**

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:

```lisp
(%i1) load(draw)$
(%i2) draw2d(xrange = [-3,5],
   explicit(x^2,x,-1,1))$
```

See also `yrange` and `zrange`. 
xrange_secondary  Graphic option

Default value: auto

If `xrange_secondary` is `auto`, the range for the second x axis is computed automatically.

If the user wants a specific interval for the second x axis, it must be given as a Maxima list, as in `xrange_secondary=[-2, 3]`.

Since this is a global graphics option, its position in the scene description does not matter.

See also `xrange`, `yrange`, `zrange` and `yrange_secondary`.

xtics  Graphic option

Default value: auto

This graphic option controls the way tic marks are drawn on the x axis.

- When option `xtics` is bounded to symbol `auto`, tic marks are drawn automatically.
- When option `xtics` is bounded to symbol `none`, tic marks are not drawn.
- When option `xtics` is bounded to a positive number, this is the distance between two consecutive tic marks.
- When option `xtics` is bounded to a list of length three of the form `[start, incr, end]`, tic marks are plotted from `start` to `end` at intervals of length `incr`.
- When option `xtics` is bounded to a set of numbers of the form `{n1, n2, ...}`, tic marks are plotted at values `n1, n2, ...`.
- When option `xtics` is bounded to a set of pairs of the form `{"label1", n1}, {"label2", n2}, ...`, tic marks corresponding to values `n1, n2, ...` are labeled with "label1", "label2", ..., respectively.

Since this is a global graphics option, its position in the scene description does not matter.

Examples:

Disable tics.

```
(%i1) load(draw)$
(%i2) draw2d(xtics = 'none, explicit(x^3,x,-1,1) )$
```

Tics every 1/4 units.

```
(%i1) load(draw)$
(%i2) draw2d(xtics = 1/4, explicit(x^3,x,-1,1) )$
```

Tics from -3/4 to 3/4 in steps of 1/8.

```
(%i1) load(draw)$
(%i2) draw2d(xtics = [-3/4,1/8,3/4], explicit(x^3,x,-1,1) )$
```

Tics at points -1/2, -1/4 and 3/4.
Chapter 51: draw

(%i1) load(draw)$
(%i2) draw2d(xtics = {-1/2,-1/4,3/4},
    explicit(x^3,x,-1,1) )$

Labeled tics.

(%i1) load(draw)$
(%i2) draw2d(xtics = {["High",0.75],["Medium",0],["Low",-0.75]},
    explicit(x^3,x,-1,1) )$

See also ytics, and ztics.

**xtics_axis**

Graphic option

Default value: `false`

If `xtics_axis` is `true`, tic marks and their labels are plotted just along the x axis, if it is `false` tics are plotted on the border.

Since this is a global graphics option, its position in the scene description does not matter.

**xtics_rotate**

Graphic option

Default value: `false`

If `xtics_rotate` is `true`, tic marks on the x axis are rotated 90 degrees.

Since this is a global graphics option, its position in the scene description does not matter.

**xtics_rotate_secondary**

Graphic option

Default value: `false`

If `xtics_rotate_secondary` is `true`, tic marks on the secondary x axis are rotated 90 degrees.

Since this is a global graphics option, its position in the scene description does not matter.

**xtics_secondary**

Graphic option

Default value: `auto`

This graphic option controls the way tic marks are drawn on the second x axis.

See `xtics` for a complete description.

**xtics_secondary_axis**

Graphic option

Default value: `false`

If `xtics_secondary_axis` is `true`, tic marks and their labels are plotted just along the secondary x axis, if it is `false` tics are plotted on the border.

Since this is a global graphics option, its position in the scene description does not matter.
**xu_grid**  
Graphic option  
Default value: 30  
*xu_grid* is the number of coordinates of the first variable (*x* in explicit and *u* in parametric 3d surfaces) to build the grid of sample points.  
This option affects the following graphic objects:  
- *gr3d*: explicit and *parametric_surface*.  
Example:  
```
(%i1) load(draw)
(%i2) draw3d(xu_grid = 10,
yv_grid = 50,
        explicit(x^2+y^2,x,-3,3,y,-3,3) )$
```
See also *yv_grid*.

**xy_file**  
Graphic option  
Default value: "" (empty string)  
*xy_file* is the name of the file where the coordinates will be saved after clicking with the mouse button and hitting the 'x' key. By default, no coordinates are saved.  
Since this is a global graphics option, its position in the scene description does not matter.

**xyplane**  
Graphic option  
Default value: *false*  
Allocates the xy-plane in 3D scenes. When *xyplane* is *false*, the xy-plane is placed automatically; when it is a real number, the xy-plane intersects the z-axis at this level. This option has no effect in 2D scenes.  
Since this is a global graphics option, its position in the scene description does not matter.  
Example:  
```
(%i1) load(draw)
(%i2) draw3d(xyplane = %e-2,
        explicit(x^2+y^2,x,-1,1,y,-1,1))$
```

**y_voxel**  
Graphic option  
Default value: 10  
*y_voxel* is the number of voxels in the y direction to be used by the *marching cubes algorithm* implemented by the 3d *implicit* object. It is also used by graphic object *region*.

**yaxis**  
Graphic option  
Default value: *false*  
If *yaxis* is *true*, the *y* axis is drawn.  
Since this is a global graphics option, its position in the scene description does not matter.  
Example:
Chapter 51: draw

(%i1) load(draw)$
(%i2) draw2d(explicit(x^3,x,-1,1),
   yaxis = true,
   yaxis_color = blue)$

See also yaxis_width, yaxis_type and yaxis_color.

**yaxis_color**

Graphic option

Default value: "black"

*yaxis_color* specifies the color for the *y* axis. See *color* to know how colors are defined.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

(%i1) load(draw)$
(%i2) draw2d(explicit(x^3,x,-1,1),
   yaxis = true,
   yaxis_color = red)$

See also yaxis, yaxis_width and yaxis_type.

**yaxis_secondary**

Graphic option

Default value: false

If *yaxis_secondary* is true, function values can be plotted with respect to the second *y* axis, which will be drawn on the right side of the scene.

Note that this is a local graphics option which only affects to 2d plots.

Example:

(%i1) load(draw)$
(%i2) draw2d(
   explicit(sin(x),x,0,10),
   yaxis_secondary = true,
   ytics_secondary = true,
   color = blue,
   explicit(100*sin(x+0.1)+2,x,0,10));

See also yrange_secondary, ytics_secondary, ytics_rotate_secondary and ytics_axis_secondary.

**yaxis_type**

Graphic option

Default value: dots

*yaxis_type* indicates how the *y* axis is displayed; possible values are *solid* and *dots*.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

(%i1) load(draw)$
(%i2) draw2d(explicit(x^3,x,-1,1),
   yaxis = true,
   yaxis_type = solid)$

See also yaxis, yaxis_width and yaxis_color.
Graphic option

**yaxis_width**

Default value: 1

*yaxis_width* is the width of the y axis. Its value must be a positive number.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(explicit(x^3,x,-1,1),
              yaxis = true,
              yaxis_width = 3)$
```

See also *yaxis*, *yaxis_type* and *yaxis_color*.

Graphic option

**ylabel**

Default value: "" (empty string)

Option *ylabel*, a string, is the label for the y axis. By default, no label is written.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(xlabel = "Time",
            ylabel = "Population",
            explicit(exp(u),u,-2,2) )$
```

See also *xlabel* and *zlabel*.

Graphic option

**yrange**

Default value: auto

If *yrange* is auto, the range for the y coordinate is computed automatically.

If the user wants a specific interval for y, it must be given as a Maxima list, as in *yrange=[-2, 3]*.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(yrange = [-2,3],
            explicit(x^2,x,-1,1),
            xrange = [-3,3])$
```

See also *xrange*, *yrange_secondary* and *zrange*.

Graphic option

**yrange_secondary**

Default value: auto

If *yrange_secondary* is auto, the range for the second y axis is computed automatically.

If the user wants a specific interval for the second y axis, it must be given as a Maxima list, as in *yrange_secondary=[-2, 3]*.
Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(
    explicit(sin(x),x,0,10),
    yaxis_secondary = true,
    ytics_secondary = true,
    yrange = [-3, 3],
    yrange_secondary = [-20, 20],
    color = blue,
    explicit(100*sin(x+0.1)+2,x,0,10)) $
```

See also `xrange`, `yrange` and `zrange`.

**ytics**  
Graphic option  
Default value: `auto`  
This graphic option controls the way tic marks are drawn on the y axis.  
See `xtics` for a complete description.

**ytics_axis**  
Graphic option  
Default value: `false`  
If `ytics_axis` is `true`, tic marks and their labels are plotted just along the y axis, if it is `false` tics are plotted on the border.  
Since this is a global graphics option, its position in the scene description does not matter.

**ytics_rotate**  
Graphic option  
Default value: `false`  
If `ytics_rotate` is `true`, tic marks on the y axis are rotated 90 degrees.  
Since this is a global graphics option, its position in the scene description does not matter.

**ytics_rotate_secondary**  
Graphic option  
Default value: `false`  
If `ytics_rotate_secondary` is `true`, tic marks on the secondary y axis are rotated 90 degrees.  
Since this is a global graphics option, its position in the scene description does not matter.

**ytics_secondary**  
Graphic option  
Default value: `auto`  
This graphic option controls the way tic marks are drawn on the second y axis.  
See `xtics` for a complete description.
**ytic_secondary_axis**

Graphic option

Default value: `false`

If `ytic_secondary_axis` is `true`, tic marks and their labels are plotted just along the secondary `y` axis, if it is `false` tics are plotted on the border.

Since this is a global graphics option, its position in the scene description does not matter.

**yv_grid**

Graphic option

Default value: 30

`yv_grid` is the number of coordinates of the second variable (\(y\) in explicit and \(v\) in parametric 3d surfaces) to build the grid of sample points.

This option affects the following graphic objects:

- `gr3d`: `explicit` and `parametric_surface`.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(xu_grid = 10,
           yv_grid = 50,
           explicit(x^2+y^2,x,-3,3,y,-3,3) )$
```

See also `xu_grid`.

**z_voxel**

Graphic option

Default value: 10

`z_voxel` is the number of voxels in the `z` direction to be used by the `marching cubes algorithm` implemented by the 3d `implicit` object.

**zaxis**

Graphic option

Default value: `false`

If `zaxis` is `true`, the `z` axis is drawn in 3D plots. This option has no effect in 2D scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
           zaxis = true,
           zaxis_type = solid,
           zaxis_color = blue)$
```

See also `zaxis_width`, `zaxis_type` and `zaxis_color`.

**zaxis_color**

Graphic option

Default value: "black"

`zaxis_color` specifies the color for the `z` axis. See `color` to know how colors are defined. This option has no effect in 2D scenes.
Since this is a global graphics option, its position in the scene description does not matter.
Example:

```lisp
(%i1) load(draw)$
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
   zaxis = true,
   zaxis_type = solid,
   zaxis_color = red)$
```

See also zaxis, zaxis_width and zaxis_type.

**zaxis_type**

Graphic option

Default value: dots

zaxis_type indicates how the z axis is displayed; possible values are solid and dots. This option has no effect in 2D scenes.

Since this is a global graphics option, its position in the scene description does not matter.
Example:

```lisp
(%i1) load(draw)$
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
   zaxis = true,
   zaxis_type = solid)$
```

See also zaxis, zaxis_width and zaxis_color.

**zaxis_width**

Graphic option

Default value: 1

zaxis_width is the width of the z axis. Its value must be a positive number. This option has no effect in 2D scenes.

Since this is a global graphics option, its position in the scene description does not matter.
Example:

```lisp
(%i1) load(draw)$
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
   zaxis = true,
   zaxis_type = solid,
   zaxis_width = 3)$
```

See also zaxis, zaxis_type and zaxis_color.

**zlabel**

Graphic option

Default value: "" (empty string)

Option zlabel, a string, is the label for the z axis. By default, no label is written.

Since this is a global graphics option, its position in the scene description does not matter.
Example:
(%i1) load(draw)$
(%i2) draw3d(zlabel = "Z variable",
    ylabel = "Y variable",
    explicit(sin(x^2+y^2),x,-2,2,y,-2,2),
    xlabel = "X variable")$

See also xlabel, and ylabel.

**zrange**

Default value: auto

If `zrange` is auto, the range for the z coordinate is computed automatically.

If the user wants a specific interval for z, it must be given as a Maxima list, as in `zrange=[-2, 3]`.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

(%i1) load(draw)$
(%i2) draw3d(yrange = [-3,3],
    zrange = [-2,5],
    explicit(x^2+y^2,x,-1,1,y,-1,1),
    xrange = [-3,3])$

See also `xrange` and `yrange`.

**ztics**

Default value: auto

This graphic option controls the way tic marks are drawn on the z axis.

See `xtics` for a complete description.

**ztics_axis**

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**

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.
51.2.4 Graphics objects

bars ([x1,h1,w1], [x2,h2,w2, ...])  Graphic object

Draws vertical bars in 2D.

2D bars ([x1,h1,w1], [x2,h2,w2, ...]) draws bars centered at values x1, x2, ... with heights h1, h2, ... and widths w1, w2, ...

This object is affected by the following graphic options: key, fill_color, fill_density and line_width.

Example:

(%i1) load(draw)$
(%i2) draw2d(
    key = "Group A",
    fill_color = blue,
    fill_density = 0.2,
    bars([0.8,5,0.4], [1.8,7,0.4], [2.8,-4,0.4]),
    key = "Group B",
    fill_color = red,
    fill_density = 0.6,
    line_width = 4,
    bars([1.2,4,0.4], [2.2,-2,0.4], [3.2,5,0.4]),
    xaxis = true);

cylindrical (radius,z,minz,maxz,azi,minazi,maxazi)  Graphic object

Draws 3D functions defined in cylindrical coordinates.

3D cylindrical (radius,z,minz,maxz,azi,minazi,maxazi) plots function radius(z,azi) defined in cylindrical coordinates, with variable z taking values from minz to maxz and azimuth azi taking values from minazi to maxazi.

This object is affected by the following graphic options: xu_grid, yv_grid, line_type, key and color.

Example:

(%i1) load(draw)$
(%i2) draw3d(cylindrical(1,z,-2,2,az,0,2*%pi))$

elevation_grid (mat,x0,y0,width,height)  Graphic object

Draws matrix mat in 3D space. z values are taken from mat, the abscissas range from x0 to x0 + width and ordinates from y0 to y0 + height. Element a(1,1) is projected on point (x0, y0 + height), a(1,n) on (x0 + width, y0 + height), a(m,1) on (x0, y0), and a(m,n) on (x0 + width, y0).

This object is affected by the following graphic options: line_type, line_width, key, enhanced3d, and color.

In older versions of Maxima, elevation_grid was called mesh. See also mesh.

Example:
(%i1) load(draw)$
(%i2) m: apply(
    matrix,
    makelist(makelist(random(10.0), k, 1, 30), i, 1, 20))$
(%i3) draw3d(
    color = blue,
    elevation_grid(m, 0, 0, 3, 2),
    xlabel = "x",
    ylabel = "y",
    surface_hide = true);

ellipse (xc, yc, a, b, ang1, ang2)
Draws ellipses and circles in 2D.

2D

ellipse (xc, yc, a, b, ang1, ang2) plots an ellipse centered at [xc, yc] with horizontal and vertical semi axis a and b, respectively, starting at angle ang1 with an amplitude equal to angle ang2.

This object is affected by the following graphic options: nticks, transparent, fill_color, border, line_width, line_type, key and color.

Example:
(%i1) load(draw)$
(%i2) draw2d(transparent = false,
    fill_color = red,
    color = gray30,
    transparent = false,
    line_width = 5,
    ellipse(0, 6, 3, 2, 270, -270),
    /* center (x,y), a, b, start & end in degrees */
    transparent = true,
    color = blue,
    line_width = 3,
    ellipse(2.5, 6, 2, 3, 30, -90),
    xrange = [-3, 6],
    yrange = [2, 9])$

errors ([x1,x2,...], [y1,y2,...])
Draws points with error bars, horizontally, vertically or both, depending on the value of option error_type.

2D

If error_type=x, arguments to errors must be of the form [x,y,xdelta] or [x,y,xlow,xhigh]. If error_type=y, arguments must be of the form [x,y,ydelta] or [x,y,ylow,yhigh]. If error_type=xy or error_type=boxes, arguments to errors must be of the form [x,y,xdelta,ydelta] or [x,y,xlow,xhigh,ylow,yhigh].

See also error_type.
This object is affected by the following graphic options: `error_type`, `points_joined`, `line_width`, `key`, `line_type`, `color`, `fill_density`, `xaxis_secondary`, and `yaxis_secondary`.

Option `fill_density` is only relevant when `error_type=boxes`.

Examples:

Horizontal error bars.

```
(%i1) load(draw)$
(%i2) draw2d(
    error_type = y,
    errors([[1,2,1], [3,5,3], [10,3,1], [17,6,2]]) )$
```

Vertical and horizontal error bars.

```
(%i1) load(draw)$
(%i2) draw2d(
    error_type = xy,
    points_joined = true,
    color = blue,
    errors([[1,2,1,2], [3,5,2,1], [10,3,1,1], [17,6,1/2,2]]) );
```

### `explicit (fcn, var, minval, maxval)`

Graphic object

### `explicit (fcn, var1, minval1, maxval1, var2, minval2, maxval2)`

Graphic object

Draws explicit functions in 2D and 3D.

**2D**

`explicit(fcn, var, minval, maxval)` plots explicit function `fcn`, with variable `var` taking values from `minval` to `maxval`.

This object is affected by the following graphic options: `nticks`, `adapt_depth`, `line_width`, `line_type`, `key`, `filled_func`, `fill_color` and `color`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(line_width = 3,
               color = blue,
               explicit(x^2,x,-3,3) )$
(%i3) draw2d(fill_color = brown,
               filled_func = true,
               explicit(x^2,x,-3,3) )$
```

**3D**

`explicit(fcn, var1, minval1, maxval1, var2, minval2, maxval2)` plots 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: `xu_grid`, `yv_grid`, `line_type`, `line_width`, `key`, `enhanced3d`, and `color`.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(key = "Gauss",
               color = "#a02c00",
               explicit(x^2,y,0,1, x, -2,2) )$
```
explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3),
yv_grid = 10,
color = blue,
key = "Plane",
explicit(x+y,x,-5,5,y,-5,5),
surface_hide = true)$

See also filled_func for filled functions.

**image** \((im,x0,y0,width,height)\)  
Graphic object

Renders images in 2D.

**2D**

`image (im,x0,y0,width,height)` plots image `im` in the rectangular region from vertex \((x0,y0)\) to \((x0+width,y0+height)\) on the real plane. Argument `im` must be a matrix of real numbers, a matrix of vectors of length three or a `picture` object.

If `im` is a matrix of real numbers or a `levels picture` object, pixel values are interpreted according to graphic option `palette`, which is a vector of length three with components ranging from -36 to +36; each value is an index for a formula mapping the levels onto red, green and blue colors, respectively:

<table>
<thead>
<tr>
<th>Value</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>(x)</td>
</tr>
<tr>
<td>4</td>
<td>(x^2)</td>
</tr>
<tr>
<td>5</td>
<td>(x^3)</td>
</tr>
<tr>
<td>6</td>
<td>(x^4)</td>
</tr>
<tr>
<td>7</td>
<td>(\sqrt{x})</td>
</tr>
<tr>
<td>8</td>
<td>(\sqrt{\sqrt{x}})</td>
</tr>
<tr>
<td>9</td>
<td>(\sin(90x))</td>
</tr>
<tr>
<td>10</td>
<td>(\cos(90x))</td>
</tr>
<tr>
<td>11</td>
<td>(</td>
</tr>
<tr>
<td>12</td>
<td>((2x-1)^2)</td>
</tr>
<tr>
<td>13</td>
<td>(\sin(180x))</td>
</tr>
<tr>
<td>14</td>
<td>(</td>
</tr>
<tr>
<td>15</td>
<td>(\sin(360x))</td>
</tr>
<tr>
<td>16</td>
<td>(\cos(360x))</td>
</tr>
<tr>
<td>17</td>
<td>(</td>
</tr>
<tr>
<td>18</td>
<td>(</td>
</tr>
<tr>
<td>19</td>
<td>(</td>
</tr>
<tr>
<td>20</td>
<td>(</td>
</tr>
<tr>
<td>21</td>
<td>(3x)</td>
</tr>
<tr>
<td>22</td>
<td>(3x-1)</td>
</tr>
<tr>
<td>23</td>
<td>(3x-2)</td>
</tr>
<tr>
<td>24</td>
<td>(</td>
</tr>
<tr>
<td>25</td>
<td>(</td>
</tr>
<tr>
<td>26</td>
<td>(</td>
</tr>
<tr>
<td>27</td>
<td>(</td>
</tr>
<tr>
<td>30</td>
<td>(x/0.32-0.78125)</td>
</tr>
<tr>
<td>31</td>
<td>(2x-0.84)</td>
</tr>
<tr>
<td>32</td>
<td>(4x;1;-2x+1.84;x/0.08-11.5)</td>
</tr>
<tr>
<td>33</td>
<td>(</td>
</tr>
<tr>
<td>34</td>
<td>(2x)</td>
</tr>
<tr>
<td>35</td>
<td>(2x - 0.5)</td>
</tr>
<tr>
<td>36</td>
<td>(2x - 1)</td>
</tr>
</tbody>
</table>

Negative numbers mean negative colour component.

`palette = gray` and `palette = color` are short cuts for `palette = [3,3,3]` and `palette = [7,5,15]`, respectively.

If `im` is a matrix of vectors of length three or an `rgb picture` object, they are interpreted as red, green and blue color components.

Examples:

If `im` is a matrix of real numbers, pixel values are interpreted according to graphic option `palette`.

(%i1) load(draw)$
(%i2) im: apply('matrix,
makelist(makelist(random(200),i,1,30),i,1,30))$
(%i3) /* palette = color, default */
    draw2d(image(im,0,0,30,30))$
(**draw2d**(palette = gray, image(im,0,0,30,30))
**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.

```lisp
(%i1) load(draw)$
(%i2) im: apply('matrix,
  makelist(makelist([random(300), random(300), random(300)],i,1,30),i,1,30))$
(%i3) draw2d(image(im,0,0,30,30))$
```

Package draw automatically loads package picture. In this example, a level picture object is built by hand and then rendered.

```lisp
(%i1) load(draw)$
(%i2) im: make_level_picture([45,87,2,134,204,16],3,2);
(%o2) picture(level, 3, 2, {Array: #(45 87 2 134 204 16)})
(%i3) /* default color palette */
  draw2d(image(im,0,0,30,30))$
(%i4) /* gray palette */
  draw2d(palette = gray,
    image(im,0,0,30,30))$
```

An xpm file is read and then rendered.

```lisp
(%i1) load(draw)$
(%i2) im: read_xpm("myfile.xpm")$
(%i3) draw2d(image(im,0,0,10,7))$
```

See also make_level_picture, make_rgb_picture and read_xpm.


**implicit**  \((fcn,x,xmin,xmax,y,ymin,ymax)\)  

\[ \text{Graphic object} \]

\[ \text{Draws implicit functions in 2D and 3D.} \]

**2D**

\[ \text{implicit}(fcn,x,xmin,xmax,y,ymin,ymax) \] 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:

```lisp
(%i1) load(draw)$
(%i2) draw2d(terminal = eps,
```
grid = true,  
line_type = solid,  
key = "y^2=x^3-2*x+1",  
implicit(y^2=x^3-2*x+1, x, -4,4, y, -4,4),  
line_type = dots,  
key = "x^3+y^3 = 3*x*y^2-x-1",  
implicit(x^3+y^3 = 3*x*y^2-x-1, x,-4,4, y,-4,4),  
title = "Two implicit functions" )$

3D implicit (fcn, x, xmin, xmax, y, ymin, ymax, z, zmin, zmax) plots the implicit surface defined by fcn, with variable x taking values from xmin to xmax, variable y taking values from ymin to ymax and variable z taking values from zmin to zmax. This object implements the marching cubes algorithm.

This object is affected by the following graphic options: x_voxel, y_voxel, z_voxel, line_width, line_type, key and color.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(
   color=blue,
   implicit((x^2+y^2+z^2-1)*(x^2+(y-1.5)^2+z^2-0.5)=0.015,  
   x,-1,1,y,-1.2,2.3,z,-1,1),  
   surface_hide=true;)
```

**label** ([string,x,y]...)  
**label** ([string,x,y,z]...)  

Graphic object

Graphic object

Writes labels in 2D and 3D.

Colored labels work only with Gnuplot 4.3. This is a known bug in package draw.

This object is affected by the following graphic options: label_alignment, label_orientation and color.

**2D**

label([string,x,y]) writes the string at point [x,y].

Example:

```
(%i1) load(draw)$
(%i2) draw2d(yrange = [0.1,1.4],  
   color = red,  
   label(["Label in red",0,0.3]),  
   color = "#000ff",  
   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:
Chapter 51: draw

(%i1) load(draw)$
(%i2) draw3d(explicit(exp(sin(x)+cos(x^2)),x,-3,3,y,-3,3),
    color = red,
    label(["UP 1",-2,0,3], ["UP 2",1.5,0,4]),
    color = blue,
    label(["DOWN 1",2,0,-3]) )$

mesh (row_1,row_2,...) Graphic object
Draws a quadrangular mesh in 3D.

3D
Argument row_j is a list of n 3D points of the form [[x_i1,y_i1,z_i1], ..., [x_i_n,y_i_n,z_i_n]], 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, enhanced3d, and transform.

Examples:
A simple example.

(%i1) load(draw)$
(%i2) draw3d(
    mesh([[1,1,3], [7,3,1], [12,-2,4], [15,0,5]],
    [[2,7,8], [4,3,1], [10,5,8], [12,7,1]],
    [[-2,11,10],[6,9,5],[6,15,1],[20,15,2]])) $

Plotting a triangle in 3D.

(%i1) load(draw)$
(%i2) draw3d(
    line_width = 2,
    mesh([[1,0,0],[0,1,0]],
    [[0,0,1],[0,0,1]])) $

Two quadrilaterals.

(%i1) load(draw)$
(%i2) draw3d(
    surface_hide = true,
    line_width = 3,
    color = red,
    mesh([[0,0,0],[0,1,0]],
    [[2,0,2],[2,2,2]]),
    color = blue,
    mesh([[0,0,2],[0,1,2]],
    [[2,0,4],[2,2,4]])) $

parametric (xfun,yfun,par,parmin,parmax) Graphic object
parametric (xfun,yfun,zfun,par,parmin,parmax) Graphic object
Draws parametric functions in 2D and 3D.

This object is affected by the following graphic options: nticks, line_width, line_type, key, color and enhanced3d.
2D

**parametric ([xfun, yfun, par, parmin, parmax])** plots parametric function 

\([xfun, yfun]\), with parameter \(par\) taking values from \(parmin\) to \(parmax\).

Example:

```lisp
(%i1) load(draw)$
(%i2) draw2d(explicit(exp(x), x, -1, 3),
   color = red,
   key = "This is the parametric one!!",
   parametric(2*cos(rrr), rrr^2, rrr, 0, 2*%pi))$
```

3D

**parametric ([xfun, yfun, zfun, par, parmin, parmax])** plots parametric curve 

\([xfun, yfun, zfun]\), with parameter \(par\) taking values from \(parmin\) to \(parmax\).

Example:

```lisp
(%i1) load(draw)$
(%i2) draw3d(explicit(exp(sin(x)+cos(x^2)), x, -3, 3, y, -3, 3),
   color = royalblue,
   parametric(cos(5*u)^2, sin(7*u), u-2, u, 0, 2),
   color = turquoise,
   line_width = 2,
   parametric(t^2, sin(t), 2+t, t, 0, 2),
   surface_hide = true,
   title = "Surface & curves")$
```

**parametric_surface**

(\([xfun, yfun, zfun, par1, par1min, par1max, par2, par2min, par2max]\))

Draws parametric surfaces in 3D.

**3D**

**parametric_surface ([xfun, yfun, zfun, par1, par1min, par1max, par2, par2min, par2max])**

plots parametric surface \([xfun, yfun, zfun]\), with parameter \(par1\) taking values from \(par1min\) to \(par1max\) and parameter \(par2\) taking values from \(par2min\) to \(par2max\).

This object is affected by the following **graphic options**: \(xu\_grid\), \(yv\_grid\), \(line\_type\), \(line\_width\), \(key\), \(enhanced3d\), and \(color\).

Example:

```lisp
(%i1) load(draw)$
(%i2) draw3d(title = "Sea shell",
   xu_grid = 100,
   yv_grid = 25,
   view = [100, 20],
   surface_hide = true,
   parametric_surface(0.5*u*cos(u)*(cos(v)+1), 0.5*u*sin(u)*(cos(v)+1), u*sin(v) - ((u+3)/8*%pi)^2 - 20, u, 0, 13*%pi, v, -%pi, %pi))$
```
Chapter 51: draw

**points** (\[[x_1,y_1], [x_2,y_2],...\])

Graphic object

**points** (\[[x_1,x_2,...], [y_1,y_2,...]\])

Graphic object

**points** (\[[y_1,y_2,...]\])

Graphic object

**points** (\[[x_1,y_1,z_1], [x_2,y_2,z_2],...\])

Graphic object

**points** (\[[x_1,x_2,...], [y_1,y_2,...], [z_1,z_2,...]\])

Graphic object

**points** (matrix)

Graphic object

**points** (\[1d_y\_array\])

Graphic object

**points** (\[1d_x\_array, 1d_y\_array\])

Graphic object

**points** (\[1d_x\_array, 1d_y\_array, 1d_z\_array\])

Graphic object

**points** (\[2d\_xy\_array\])

Graphic object

**points** (\[2d\_xyz\_array\])

Graphic object

Draws points in 2D and 3D.

This object is affected by the following graphic options: **point_size**, **point_type**, **points_joined**, **line_width**, **key**, **line_type** and **color**. In 3D mode, it is also affected by **enhanced3d**.

### 2D

**points** (\[[x_1,y_1], [x_2,y_2],...\]) or **points** (\[[x_1,x_2,...], [y_1,y_2,...]\]) plots points \([x_1,y_1], [x_2,y_2],...\) etc. If abscissas are not given, they are set to consecutive positive integers, so that **points** (\[[y_1,y_2,...]\]) draws points \([1,y_1], [2,y_2],...\) etc. If **matrix** is a two-column or two-row matrix, **points** (**matrix**) draws the associated points. If **matrix** is a one-column or one-row matrix, abscissas are assigned automatically.

If **1d_y\_array** is a 1D lisp array of numbers, **points** (**1d_y\_array**) plots them setting abscissas to consecutive positive integers. **points** (**1d_x\_array, 1d_y\_array**) plots points with their coordinates taken from the two arrays passed as arguments. If **2d\_xy\_array** is a 2D array with two columns, or with two rows, **points** (**2d\_xy\_array**) plots the corresponding points on the plane.

Examples:

Two types of arguments for **points**, a list of pairs and two lists of separate coordinates.

(\%i1) load(draw)$
(\%i2) draw2d($
    key = "Small points",
    points(makelist([random(20),random(50)],k,1,10)),
    point_type = circle,
    point_size = 3,
    points_joined = true,
    key = "Great points",
    points(makelist(k,k,1,20),makelist(random(30),k,1,20)),
    point_type = filled_down_triangle,
    key = "Automatic abscissas",
    color = red,
    points([2,12,8]))$

Drawing impulses.

(\%i1) load(draw)$
(\%i2) draw2d($
    points_joined = impulses,
line_width = 2,
color = red,
points(makelist([[random(20),random(50)],k,1,10]))$

Array with ordinates.
(%i1) load(draw)$
(%i2) a: make_array (flonum, 100) $
(%i3) for i:0 thru 99 do a[i]: random(1.0) $
(%i4) draw2d(points(a)) $

Two arrays with separate coordinates.
(%i1) load(draw)$
(%i2) x: make_array (flonum, 100) $
(%i3) y: make_array (fixnum, 100) $
(%i4) for i:0 thru 99 do ( 
    x[i]: float(i/100),
    y[i]: random(10) ) $
(%i5) draw2d(points(x, y)) $

A two-column 2D array.
(%i1) load(draw)$
(%i2) xy: make_array(flonum, 100, 2) $
(%i3) for i:0 thru 99 do ( 
    xy[i, 0]: float(i/100),
    xy[i, 1]: random(10) ) $
(%i4) draw2d(points(xy)) $

Drawing an array filled with function read_array.
(%i1) load(draw)$
(%i2) a: make_array(flonum,100) $
(%i3) read_array (file_search ("pidigits.data"), a) $
(%i4) draw2d(points(a)) $

3D
points ([[x1,y1,z1], [x2,y2,z2],...]) or points ([[x1,x2,...], [y1,y2,...], [z1,z2,...]]) plots points [x1,y1,z1], [x2,y2,z2], etc. If matrix is a three-column or three-row matrix, points (matrix) draws the associated points.

When arguments are lisp arrays, points (1d_x_array, 1d_y_array, 1d_z_array) takes coordinates from the three 1D arrays. If 2d_xyz_array is a 2D array with three columns, or with three rows, points (2d_xyz_array) plots the corresponding points.

Examples:
One tridimensional sample,
(%i1) load(draw)$
(%i2) load (numericalio)$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) draw3d(title = "Daily average wind speeds",
    point_size = 2,
    points(args(submatrix (s2, 4, 5)))) $

Two tridimensional samples,
Chapter 51: draw

(%i1) load(draw)$
(%i2) load(numericalio)$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) draw3d(
   title = "Daily average wind speeds. Two data sets",
   point_size = 2,
   key = "Sample from stations 1, 2 and 3",
   points(args(submatrix (s2, 4, 5)));
   point_type = 4,
   key = "Sample from stations 1, 4 and 5",
   points(args(submatrix (s2, 2, 3))) )$

Unidimensional arrays,

(%i1) load(draw)$
(%i2) x: make_array (fixnum, 10)$
(%i3) y: make_array (fixnum, 10)$
(%i4) z: make_array (fixnum, 10)$
(%i5) for i:0 thru 9 do ( 
   x[i]: random(10),
   y[i]: random(10),
   z[i]: random(10) )$
(%i6) draw3d(points(x,y,z))$

Bidimensional colored array,

(%i1) load(draw)$
(%i2) xyz: make_array(fixnum, 10, 3)$
(%i3) for i:0 thru 9 do ( 
   xyz[i, 0]: random(10),
   xyz[i, 1]: random(10),
   xyz[i, 2]: random(10) )$
(%i4) draw3d(
   enhanced3d = true,
   points_joined = true,
   points(xyz))$

Color numbers explicitly specified by the user.

(%i1) load(draw)$
(%i2) pts: makelist([t,t^2,cos(t)], t, 0, 15)$
(%i3) col_num: makelist(k, k, 1, length(pts))$
(%i4) draw3d(
   enhanced3d = ['part(col_num,k),k],
   point_size = 3,
   point_type = filled_circle,
   points(pts))$

polar (radius,ang,minang,maxang)  

Graphic object

Draws 2D functions defined in polar coordinates.

2D

polar (radius,ang, minang, maxang) plots function radius(ang) defined in polar coordinates, with variable ang taking values from minang to maxang.
This object is affected by the following graphic options: nticks, line_width, line_type, key and color.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(user_preamble = "set grid polar",
                nticks = 200,
                xrange = [-5,5],
                yrange = [-5,5],
                color = blue,
                line_width = 3,
                title = "Hyperbolic Spiral",
                polar(10/theta,theta,1,10*%pi) )$
```

**polygon** ([[[x1,y1], [x2,y2],...]])

Graphic object

2D

**polygon** ([[x1,y1], [x2,y2],...]) or **polygon** ([[x1,x2,...], [y1,y2,...]]): plots on the plane a polygon with vertices [x1,y1], [x2,y2], etc..

This object is affected by the following graphic options: transparent, fill_color, border, line_width, key, line_type and color.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(color = "#e245f0",
                line_width = 8,
                polygon([[3,2],[7,2],[5,5]]),
                border = false,
                fill_color = yellow,
                polygon([[5,2],[9,2],[7,5]]) )$
```

**quadrilateral** (point_1, point_2, point_3, point_4)

Graphic object

2D

**quadrilateral** ([[x1,y1], [x2,y2], [x3,y3], [x4,y4]]) draws a quadrilateral with vertices [x1,y1], [x2,y2], [x3,y3], and [x4,y4].

This object is affected by the following graphic options: transparent, fill_color, border, line_width, key, xaxis_secondary, yaxis_secondary, line_type, transform and color.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(
    quadrilateral([[1,1],[2,2],[3,-1],[2,-2]]))$
```

3D

**quadrilateral** ([[x1,y1,z1], [x2,y2,z2], [x3,y3,z3], [x4,y4,z4]]) draws a quadrilateral with vertices [x1,y1,z1], [x2,y2,z2], [x3,y3,z3], and [x4,y4,z4].
Chapter 51: draw

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:

```latex
(%i1) load(draw)$
(%i2) draw2d(fill_color = red,
        line_width = 6,
        line_type = dots,
        transparent = false,
        fill_color = blue,
        rectangle([-2,-2],[8,-1]), /* opposite vertices */
        transparent = true,
        line_type = solid,
        line_width = 1,
        rectangle([9,4],[2,-1.5]),
        xrange = [-3,10],
        yrange = [-3,4.5])$
```

**region** (expr, var1,minval1,maxval1,var2,minval2,maxval2)

Graphic object

Plots a region on the plane defined by inequalities.

2D expr is an expression formed by inequalities and boolean operators and, or, and not. The region is bounded by the rectangle defined by [minval1,maxval1] and [minval2,maxval2].

This object is affected by the following graphic options: fill_color, key, x_voxel, and y_voxel.

Example:

```latex
(%i1) load(draw)$
(%i2) draw2d(
        x_voxel = 30,
        y_voxel = 30,
        region(x^2+y^2<1 and x^2+y^2 > 1/2,
               x, -1.5, 1.5, y, -1.5, 1.5));
```

**spherical** (radius,azi,iminazi,imaxazi,zen,iminzen,imaxzen)

Graphic object

Draws 3D functions defined in spherical coordinates.

3D

spherical (radius,azi,iminazi,imaxazi,zen,iminzen,imaxzen) plots function radius(azi,zen) defined in spherical coordinates, with azimuth azi taking values from minazi to maxazi and zenith zen taking values from minzen to maxzen.
This object is affected by the following graphic options: xu_grid, yv_grid, line_type, key and color.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(spherical(1,a,0,2*%pi,z,0,4*%pi))$
```

triangle (point_1, point_2, point_3)  
Draws a triangle.

2D

triangle ([x1,y1], [x2,y2], [x3,y3]) draws a triangle with vertices [x1,y1], [x2,y2], and [x3,y3].

This object is affected by the following graphic options: transparent, fill_color, border, line_width, key, xaxis_secondary, yaxis_secondary, line_type, transform, and color.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(
   triangle([1,1],[2,2],[3,-1]))
```

3D

triangle ([x1,y1,z1], [x2,y2,z2], [x3,y3,z3]) draws a triangle with vertices [x1,y1,z1], [x2,y2,z2], and [x3,y3,z3].

This object is affected by the following graphic options: line_type, line_width, color, key, enhanced3d, and transform.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(
   enhanced3d = true,
   xu_grid = 50,
   tube(cos(a), a, 0, cos(a/10)^2,
   a, 0, 4*%pi ) )$
```

tube (xfun, yfun, zfun, rfun, p, pmin, pmax)  
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, enhanced3d, color, and tube_extremes.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(
   enhanced3d = true,
   xu_grid = 50,
   tube(cos(a), a, 0, cos(a/10)^2,
   a, 0, 4*%pi ) )$
```

vector ([x,y], [dx,dy])  
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:

```lisp
(%i1) load(draw)$
(%i2) draw2d(xrange = [0,12],
   yrange = [0,10],
   head_length = 1,
   vector([0,1],[5,5]), /* default type */
   head_type = 'empty,
   vector([3,1],[5,5]),
   head_both = true,
   head_type = 'nofilled,
   line_type = dots,
   vector([6,1],[5,5]))$
```

3D

vector([x,y,z], [dx,dy,dz]) plots vector [dx,dy,dz] with origin in [x,y,z].

Example:

```lisp
(%i1) load(draw)$
(%i2) draw3d(color = cyan,
   vector([0,0,0],[1,1,1]/sqrt(3)),
   vector([0,0,0],[1,-1,0]/sqrt(2)),
   vector([0,0,0],[1,1,-2]/sqrt(6)) )$
```

51.3 Functions and Variables for pictures

- **get_pixel** (pic,x,y) Function
  Returns pixel from picture. Coordinates x and y range from 0 to width-1 and height-1, respectively.

- **make_level_picture** (data) Function
- **make_level_picture** (data,width,height) Function
  Returns a levels picture object. make_level_picture (data) builds the picture object from matrix data. make_level_picture (data,width,height) builds the object from a list of numbers; in this case, both the width and the height must be given.

  The returned picture object contains the following four parts:
  1. symbol level
  2. image width
  3. image height
  4. an integer array with pixel data ranging from 0 to 255. Argument data must contain only numbers ranged from 0 to 255; negative numbers are substituted by 0, and those which are greater than 255 are set to 255.
Example:
Level picture from matrix.
(%i1) load(draw)$
(%i2) make_level_picture(matrix([3,2,5],[7,-9,3000]));
(%o2) picture(level, 3, 2, {Array: #(3 2 5 7 0 255)})

Level picture from numeric list.
(%i1) load(draw)$
(%i2) make_level_picture([-2,0,54,\%pi],2,2);
(%o2) picture(level, 2, 2, {Array: #(0 0 54 3)})

**make_rgb_picture** (redlevel,greenlevel,bluelevel) Function
Returns an rgb-coloured picture object. All three arguments must be levels picture; with red, green and blue levels.
The returned picture object contains the following four parts:
1. symbol rgb
2. image width
3. image height
4. an integer array of length 3*width*height with pixel data ranging from 0 to 255. Each pixel is represented by three consecutive numbers (red, green, blue).

Example:
(%i1) load(draw)$
(%i2) red: make_level_picture(matrix([3,2],[7,260]));
(%o2) picture(level, 2, 2, {Array: #(3 2 7 255)})
(%i3) green: make_level_picture(matrix([54,23],[73,-9]));
(%o3) picture(level, 2, 2, {Array: #(54 23 73 0)})
(%i4) blue: make_level_picture(matrix([123,82],[45,32.5698]));
(%o4) picture(level, 2, 2, {Array: #(123 82 45 33)})
(%i5) make_rgb_picture(red,green,blue);
(%o5) picture(rgb, 2, 2,
       {Array: #(3 54 123 2 23 82 7 73 45 255 0 33)})

**negative_picture** (pic) Function
Returns the negative of a (level or rgb) picture.

**picture_equalp** (x,y) Function
Returns true in case of equal pictures, and false otherwise.

**picturep** (x) Function
Returns true if the argument is a well formed image, and false otherwise.

**read_xpm** (xpm_file) Function
Reads a file in xpm and returns a picture object.

**rgb2level** (pic) Function
Transforms an rgb picture into a level one by averaging the red, green and blue channels.
**take_channel** (im, color)

Function

If argument color is red, green or blue, function take_channel returns the corresponding color channel of picture im. Example:

```latex
\begin{verbatim}
(%i1) load(draw)$
(%i2) red: make_level_picture(matrix([3,2],[7,260]));
(%o2) picture(level, 2, 2, {Array: #(3 2 7 255)})
(%i3) green: make_level_picture(matrix([54,23],[73,-9]));
(%o3) picture(level, 2, 2, {Array: #(54 23 73 0)})
(%i4) blue: make_level_picture(matrix([123,82],[45,32.5698]));
(%o4) picture(level, 2, 2, {Array: #(123 82 45 33)})
(%i5) make_rgb_picture(red, green, blue);
(%o5) picture(rgb, 2, 2, {Array: #(3 54 123 2 23 82 7 73 45 255 0 33)})
(%i6) take_channel(%,'green); /* simple quote!!! */
(%o6) picture(level, 2, 2, {Array: #(54 23 73 0)})
\end{verbatim}
```

### 51.4 Functions and Variables for worldmap

This package automatically loads package draw.

#### 51.4.1 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`.

```latex
\begin{verbatim}
(%i11) load(worldmap)$
(%i12) europeanBorders:
    region_boundaries(-31.81,74.92,49.84,32.06)$
(%i13) numbered_boundaries(europeanBorders)$
\end{verbatim}
```

**make_poly_continent** (continent_name)

Function

Makes the necessary polygons to draw a colored continent or a list of countries.

Example:
(%i1) load(worldmap)$
(%i2) /* A continent */
    make_poly_continent(Africa)$
(%i3) apply(draw2d, %)$
(%i4) /* A list of countries */
    make_poly_continent([Germany, Denmark, Poland])$
(%i5) apply(draw2d, %)$

make_poly_country (country_name)
Function
Makes the necessary polygons to draw a colored country. If islands exist, one country
can be defined with more than just one polygon.
Example:
(%i1) load(worldmap)$
(%i2) make_poly_country(India)$
(%i3) apply(draw2d, %)$

make_polygon (nlist)
Function
Returns a polygon object from boundary indices. Argument nlist is a list of compo-
nents 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];
(%i5) (Array:
    #(88.750549 27.14727 88.806351 27.25305 88.901367 27.282221
     88.917877 27.321039))
(%i4) boundaries_array[1143];
(%o4) (Array:
    #(88.750549 27.14727 88.806351 27.25305 88.901367 27.282221
     88.917877 27.321039))
Chapter 51: draw

51.4.2 Graphic objects

**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:

```maxima
(%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]))$
```

Auxiliary package `worldmap` sets global variable `boundaries_array` to real world boundaries in (longitude, latitude) coordinates. These data are in the public domain and come from `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`.

```maxima
(%i1) load(worldmap)$
(%i2) c1: gr2d(geomap(Canada,United_States,
   Mexico,Cuba))$
(%i3) c2: gr2d(geomap(Africa))$
```
Chapter 51: draw

(\%14) c3: gr2d(geomap(Oceania,China,Japan))$
(\%15) c4: gr2d(geomap(France,Portugal,Spain,
Morocco,Western_Sahara))$
(\%16) 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.

(\%11) load(worldmap)$
(\%12) 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))$
(\%13) apply(draw2d, mymap)$

3D

geomap (numlist) projects map boundaries on the sphere of radius 1 centered at (0,0,0). It is possible to change the sphere or the projection type by using geomap (numlist,3Dprojection).

Available 3D projections:

- [spherical_projection,x,y,z,r]: projects map boundaries on the sphere of radius r centered at (x,y,z).

(\%11) load(worldmap)$
(\%12) 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).

(\%11) load(worldmap)$
(\%12) 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.

(\%11) load(worldmap)$
(\%12) draw3d(geomap(World_coastlines,
    [conic_projection,0,0,0,0,1,90]))$

See also http://www.telefonica.net/web2/biomates/maxima/gpdraw/geomap for more elaborated examples.
52 drawdf

52.1 Introduction to drawdf

The function \texttt{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 \texttt{load(drawdf)}. Drawdf is built upon the \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{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, \texttt{drawdf} will implicitly admit \( x=t \), and \( G(x, y)=1 \), transforming the non-autonomous equation into a system of two autonomous equations.

52.2 Functions and Variables for drawdf

52.2.1 Functions

\texttt{drawdf} \ (dydx, \ldots\texttt{options and objects}...)

\texttt{drawdf} \ (dvdu, [u,v], \ldots\texttt{options and objects}...)

\texttt{drawdf} \ (dvdu, [u,umin,umax], [v,vmin,vmax], \ldots\texttt{options and objects}...)

\texttt{drawdf} \ ([dxdt,dydt], \ldots\texttt{options and objects}...)

\texttt{drawdf} \ ([dudt,dvdt], [u,v], \ldots\texttt{options and objects}...)

\texttt{drawdf} \ ([dudt,dvdt], [u,umin,umax], [v,vmin,vmax], \ldots\texttt{options and objects}...)

Function \texttt{drawdf} draws a 2D direction field with optional solution curves and other graphics using the \texttt{draw} package.
The first argument specifies the derivative(s), and must be either an expression or a list of two expressions. \( dy/dx \), \( dx/dt \) and \( dy/dt \) are expressions that depend on \( x \) and \( y \). \( dv/du \), \( du/dt \) and \( dv/dt \) 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 \texttt{drawdf} is a superset of those supported by \texttt{draw2d} and \texttt{gr2d} from the \texttt{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 \texttt{drawdf} include: \texttt{solns_at}, \texttt{points_at}, \texttt{saddles_at}, \texttt{soln_at}, \texttt{point_at}, and \texttt{saddle_at}.

The additional graphic options supported by \texttt{drawdf} include: \texttt{field_degree}, \texttt{soln_arrows}, \texttt{field_arrows}, \texttt{field_grid}, \texttt{field_color}, \texttt{show_field}, \texttt{tstep}, \texttt{nsteps}, \texttt{duration}, \texttt{direction}, \texttt{field_tstep}, \texttt{field_nsteps}, and \texttt{field_duration}.

Commonly used graphic objects inherited from the \texttt{draw} package include: \texttt{explicit}, \texttt{implicit}, \texttt{parametric}, \texttt{polygon}, \texttt{points}, \texttt{vector}, \texttt{label}, and all others supported by \texttt{draw2d} and \texttt{gr2d}.

Commonly used graphic options inherited from the \texttt{draw} package include: \texttt{points_joined}, \texttt{color}, \texttt{point_type}, \texttt{point_size}, \texttt{line_width}, \texttt{line_type}, \texttt{key}, \texttt{title}, \texttt{xlabel}, \texttt{ylabel}, \texttt{user_preamble}, \texttt{terminal}, \texttt{dimensions}, \texttt{file_name}, and all others supported by \texttt{draw2d} and \texttt{gr2d}.

See also \texttt{draw2d}.

Users of wxMaxima or Imaxima may optionally use \texttt{wxdrawdf}, which is identical to \texttt{drawdf} except that the graphics are drawn within the notebook using \texttt{wxdraw}.

To make use of this function, write first \texttt{load(drawdf)}.

Examples:

\begin{verbatim}
(%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])$
\end{verbatim}

For backward compatibility, \texttt{drawdf} accepts most of the parameters supported by \texttt{plotdf}.

\begin{verbatim}
(%i5) drawdf(2*cos(t)-1+y, [t,y], [t,-5,10], [y,-4,9],
    [trajectory_at,0,0])$
\end{verbatim}

\texttt{soln_at} and \texttt{solns_at} draw solution curves passing through the specified points, using a slightly enhanced 4th-order Runge Kutta numerical integrator.

\begin{verbatim}
(%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))$
\end{verbatim}
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)$

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))$
53 dynamics

53.1 Introduction to dynamics

The additional package dynamics includes several functions to create various graphical representations of discrete dynamical systems and fractals, and an implementation of the Runge-Kutta 4th-order numerical method for solving systems of differential equations.

To use the functions in this package you must first load it with load("dynamics").

Changes introduced in Maxima 5.12

Starting with Maxima 5.12, the dynamics package now uses the function plot2d to do the graphs. The commands that produce graphics (with the exception of julia and mandelbrot) now accept any options of plot2d, including the option to change among the various graphical interfaces, using different plot styles and colors, and representing one or both axes in a logarithmic scale. The old options domain, pointsize, xcenter, xradius, ycenter, yradius, xaxislabel and yaxislabel are not accepted in this new version.

All programs will now accept any variables names, and not just x and y as in the older versions. Two required parameters have changes in two of the programs: evolution2d now requires a list naming explicitly the two independent variables, and the horizontal range for orbits no longer requires a step size; the range should only specify the variable name, and the minimum and maximum values; the number of steps can now be changed with the option nticks.

53.2 Functions and Variables for dynamics

chaosgame ([x1, y1]...[xm, ym], [x0, y0], b, n, ..., options, ...);

Implements the so-called chaos game: the initial point (x0, y0) is plotted and then one of the m points [x1, y1]...[xm, ym] will be selected at random. The next point plotted will be on the segment from the previous point plotted to the point chosen randomly, at a distance from the random point which will be b times that segment’s length. The procedure is repeated n times.

evolution (F, y0, n, ..., options, ...);

Draws n+1 points in a two-dimensional graph, where the horizontal coordinates of the points are the integers 0, 1, 2, ..., n, and the vertical coordinates are the corresponding values y(n) of the sequence defined by the recurrence relation

\[ y_{n+1} = F(y_n) \]

With initial value y(0) equal to y0. F must be an expression that depends only on one variable (in the example, it depend on y, but any other variable can be used), y0 must be a real number and n must be a positive integer.
evolution2d ([F, G], [u, v], [u0, v0], 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
\[
\begin{align*}
    u_{n+1} &= F(u_n, v_n) \\
    v_{n+1} &= G(u_n, v_n)
\end{align*}
\]
With initial values u0 and v0. F and G must be two expressions that depend only on two variables, u and v, which must be named explicitly in a list.

ifs ([r1, ..., rm], [A1, ..., Am], [x1, y1], ..., [xm, ym], [x0, y0], n, ..., options, ...);  
Function
Implements the Iterated Function System method. This method is similar to the method described in the function chaosgame, but instead of shrinking the segment from the current point to the randomly chosen point, the 2 components of that segment will be multiplied by the 2 by 2 matrix Ai that corresponds to the point chosen randomly.
The random choice of one of the m attractive points can be made with a non-uniform probability distribution defined by the weights r1, ..., rm. Those weights are given in cumulative form; for instance if there are 3 points with probabilities 0.2, 0.5 and 0.3, the weights r1, r2 and r3 could be 2, 7 and 10.

julia (x, y, ...options...);  
Function
Creates a graphics file with the representation of the Julia set for the complex number (x + i y). The parameters x and y must be real. The file is created in the current directory or in the user’s directory, using the XPM graphics format. The program may take several seconds to run and after it is finished, a message will be printed with the name of the file created.
The points which do not belong to the Julia set are assigned different colors, according to the number of iterations it takes the sequence starting at that point to move out of the convergence circle of radius 2. The maximum number of iterations is set with the option levels; after that number of iterations, if the sequence is still inside the convergence circle, the point will be painted with the color defined by the option color.
All the colors used for the points that do not belong to the Julia set will have the same saturation and value, but with different hue angles distributed uniformly between hue and (hue + huerange).
options is an optional sequence of options. The list of accepted options is given in a section below.

mandelbrot (options);  
Function
Creates a graphics file with the representation of the Mandelbrot set. The file is created in the current directory or in the user’s directory, using the XPM graphics format. The program may take several seconds to run and after it is finished, a message will be printed with the name of the file created.
The points which do not belong to the Mandelbrot set are assigned different colors, according to the number of iterations it takes the sequence generated with that point to move out of the convergence circle of radius 2. The maximum number of iterations is set with the option \textit{levels}; after that number of iterations, if the sequence is still inside the convergence circle, the point will be painted with the color defined by the option \textit{color}.

All the colors used for the points that do not belong to the Mandelbrot set will have the same \textit{saturation} and \textit{value}, but with different hue angles distributed uniformly between \textit{hue} and \textit{(hue + huerange)}.

\textit{options} is an optional sequence of options. The list of accepted options is given in a section below.

\textbf{orbits} ($F$, $y0$, $n1$, $n2$, [\textit{x}, \textit{x0}, \textit{xf}, \textit{xstep}], \ldots\textit{options}\ldots);$

\textbf{Function}

Draws the orbits diagram for a family of one-dimensional discrete dynamical systems, with one parameter \textit{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 $y0$, as in the case of the function \textbf{evolution}, but in this case that function will also depend on a parameter \textit{x} that will take values in the interval from \textit{x0} to \textit{xf} with increments of \textit{xstep}. Each value used for the parameter \textit{x} is shown on the horizontal axis. The vertical axis will show the $n2$ values of the sequence $y(n1+1), \ldots, y(n1+n2+1)$ obtained after letting the sequence evolve $n1$ iterations.

\textbf{rk} ($ODE$, \textit{var}, \textit{initial}, \textit{domain})

\textbf{Function}

\textbf{rk} ([\textit{ODE1},\ldots,\textit{ODEm}],[\textit{v1},\ldots,\textit{vm}],[\textit{init1},\ldots,\textit{initm}], \textit{domain})

\textbf{Function}

The first form solves numerically one first-order ordinary differential equation, and the second form solves a system of $m$ of those equations, using the 4th order Runge-Kutta method. \textit{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 \textit{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 \textit{m} equations are going to be solved, there should be \textit{m} dependent variables \textit{v1}, \textit{v2}, \ldots, \textit{vm}. The initial values for those variables will be \textit{init1}, \textit{init2}, \ldots, \textit{initm}. There will still be just one independent variable defined by \textit{domain}, as in the previous case. \textit{ODE1}, \ldots, \textit{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 \textit{ODE1}, \ldots, \textit{ODEm} in the list in exactly the same order used for the dependent variables; for instance, the third element in the list will be interpreted as the derivative of the third dependent variable.
The program will try to integrate the equations from the initial value of the independent variable until its last value, using constant increments. If at some step one of the dependent variables takes an absolute value too large, the integration will be interrupted at that point. The result will be a list with as many elements as the number of iterations made. Each element in the results list is itself another list with \( m+1 \) elements: the value of the independent variable, followed by the values of the dependent variables corresponding to that point.

\[
staircase (F, y0, n, ...options...);
\]

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 \texttt{evolution}. A staircase diagram consists of a plot of the function \( F(y) \), together with the line \( G(y) = y \). A vertical segment is drawn from the point \((y0, y0)\) on that line until the point where it intersects the function \( F \). From that point a horizontal segment is drawn until it reaches the point \((y1, y1)\) on the line, and the procedure is repeated \( n \) times until the point \((yn, yn)\) is reached.

**Options**

Each option is a list of two or more items. The first item is the name of the option, and the remainder comprises the arguments for the option.

The options accepted by the functions \texttt{evolution}, \texttt{evolution2d}, \texttt{staircase}, \texttt{orbits}, \texttt{ifs} and \texttt{chaosgame} are the same as the options for \texttt{plot2d}. In addition to those options, \texttt{orbits} accepts and extra option \texttt{pixels} that sets up the maximum number of different points that will be represented in the vertical direction.

The following options are accepted by the functions \texttt{julia} and \texttt{mandelbrot}:

- \texttt{size} takes either one or two arguments. If only one argument is given, the width and height of the graphic file created will be equal to that value, in pixels. If two arguments are given, they will define the width and height. The default value is 400 pixels for both the width and height. If the two values are not equal, the set will appear distorted.
- \texttt{levels} defines the maximum number of iterations, which is also equal to the number of colors used for points not belonging to the set. The default value is 12; larger values mean much longer processing times.
- \texttt{huerange} defines the range of hue angles used for the hue of points not belonging to the set. The default value is 360, which means that the colors will expand all the range of hues. Values bigger than 360, will mean repeated ranges of the hue, and negative values can be used to make the hue angle decrease as the number of iterations increases.
- \texttt{hue} sets the hue, in degrees, of the first color used for the points which do not belong to the set. Its default value is 300 degrees, which corresponds to magenta; the values for other standard colors are 0 for red, 45 for orange, 60 for yellow, 120 for green, 180 for cyan and 240 for blue. See also option \texttt{huerange}.
- \texttt{saturation} sets the value of the saturation used for points not belonging to the set. It must be between 0 and 1. The default is 0.46.
• **value** sets the value of the colors used for points not belonging to the set. It must be between 0 and 1; the higher the value, the brighter the colors. The default is 0.96.

• **color** must be followed by three parameters that define the hue, saturation and value, for the color used to represent the points of the set. The default value is 0 for the three parameters, which corresponds to black. For an explanation of the range of allowed values, see options hue, saturation and value.

• **center** must be followed by two real parameters, which give the coordinates, on the complex plane, of the point in the center of the region shown. The default value is 0 for both coordinates (the origin).

• **radius** sets the radius of the biggest circle inside the square region that will be displayed. The default value is 2.

• **filename** gives the name of the file where the resulting graph will be saved. The extension .xpm will be added to that name. If the file already exists, it will be replaced by the file generated by the function. The default values are julia for the Julia set, and mandelbrot for the Mandelbrot set.

**Examples**

Graphical representation and staircase diagram for the sequence: 2, cos(2), cos(cos(2)),...

```latex
(%i1) load("dynamics")$
(%i2) evolution(cos(y), 2, 11);
(%i3) staircase(cos(y), 1, 11, [y, 0, 1.2]);
```

![Graphical representation and staircase diagram](image)
If your system is slow, you’ll have to reduce the number of iterations in the following examples. And if the dots appear too small in your monitor, you might want to try a different style, such as [style, [points, 0.8]].

Orbits diagram for the quadratic map, with a parameter $a$.

$$x_{n+1} = a + x_n^2$$

(%i4) orbits(x^2+a, 0, 50, 200, [a, -2, 0.25], [style, dots]);

To enlarge the region around the lower bifurcation near $x = -1.25$ use:

(%i5) orbits(x^2+a, 0, 100, 400, [a, -1, -1.53], [x, -1.6, -0.8], [style, dots]);
Evolution of a two-dimensional system that leads to a fractal:

(%i6) f: 0.6*x*(1+2*x)+0.8*y*(x-1)-y^2-0.9$

(%i7) g: 0.1*x*(1-6*x+4*y)+0.1*y*(1+9*y)-0.4$

(%i8) evolution2d([f,g], [x,y], [-0.5,0], 50000, [style,dots]);

And an enlargement of a small region in that fractal:

(%i9) evolution2d([f,g], [x,y], [-0.5,0], 300000, [x,-0.8,-0.6], [y,-0.9,0.1], [nticks,400], [style,dots]);
A plot of Sierpinsky's triangle, obtained with the chaos game:

(%i9) chaosgame([[0, 0], [1, 0], [0.5, sqrt(3)/2]], [0.1, 0.1], 1/2, 30000, [style, dots]);

Barnsley’s fern, obtained with an Iterated Function System:

(%i10) a1: matrix([0.85,0.04],[-0.04,0.85])$

(%i11) a2: matrix([0.2,-0.26],[0.23,0.22])$

(%i12) a3: matrix([-0.15,0.28],[0.26,0.24])$

(%i13) a4: matrix([0,0],[0,0.16])$

(%i14) p1: [0,1.6]$

(%i15) p2: [0,1.6]$

(%i16) p3: [0,0.44]$
To create a file named "dynamics9.xpm" with a graphical representation of the Mandelbrot set, with 12 colors, use:

```
mandelbrot([filename,"dynamics9"])$
```

and the Julia set for the number \((-0.55 \, \text{+} \, 0.6\) can be obtained with:

```
julia(-0.55, 0.6, [levels, 36], [center, 0, 0.6], [radius, 0.3], [hue, 240], [huerange, -180], [filename, "dynamics10"])$
```
the graph will be saved in the file *dynamics10.xpm* and will show the region from -0.3 to 0.3 in the x direction, and from 0.3 to 0.9 in the y direction. 36 colors will be used, starting with blue and ending with yellow.

![Image of the graph](image.png)

To solve numerically the differential equation

\[
\frac{dx}{dt} = t - x^2
\]

With initial value \(x(t=0) = 1\), in the interval of \(t\) from 0 to 8 and with increments of 0.1 for \(t\), use:

\[
(\%i20) \text{ results: rk(t-x^2,x,1,[t,0,8,0.1])}
\]

the results will be saved in the list *results*.

To solve numerically the system:

\[
\begin{align*}
\frac{dx}{dt} &= 4 - x^2 - 4y^2 \\
\frac{dy}{dt} &= y^2 - x^2 + 1
\end{align*}
\]

for \(t\) between 0 and 4, and with values of -1.25 and 0.75 for \(x\) and \(y\) at \(t=0\):

\[
(\%i21) \text{ sol: rk([4-x^2-4*y^2,y^2-x^2+1],[x,y],[-1.25,0.75],[t,0,4,0.02])}
\]
54 ezunits

54.1 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.

\[ (x^a) \times (y^b) \text{ is equal to } (x \times y)^{(a \times b)}. \]

\[ (x^a) + (y^a) \text{ is equal to } (x + y)^a. \]

\[ (x^a)^y \text{ is equal to } x^{a \times y} \text{ when } y \text{ is nondimensional.} \]

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.

### 54.2 Introduction to physical\_constants

`physical\_constants` is a collection of physical constants, copied from CODATA 2006 recommended values. \[1\] `load(physical\_constants)` loads this package, and loads `ezunits` also, if it is not already loaded.

A physical constant is represented as a symbol which has a property which is the constant value. The constant value is a dimensional quantity, as represented by `ezunits`. The function `constvalue` fetches the constant value; the constant value is not the ordinary value of the symbol, so symbols of physical constants persist in evaluated expressions until their values are fetched by `constvalue`.

`physical\_constants` includes some 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.

- `%c` speed of light in vacuum
- `%mu_0` magnetic constant
- `%e_0` electric constant
- `%Z_0` characteristic impedance of vacuum
- `%G` Newtonian constant of gravitation
- `%h` Planck constant
- `%h_bar` Planck constant
- `%m_P` Planck mass
- `%T_P` Planck temperature
- `%l_P` Planck length
- `%t_P` Planck time
- `%e` elementary charge
- `%e_0` magnetic flux quantum
- `%G_0` conductance quantum
- `%K_J` Josephson constant
- `%R_K` von Klitzing constant
- `%mu_B` Bohr magneton
- `%mu_N` nuclear magneton
%alpha  fine-structure constant
%R_inf  Rydberg constant
%a_0    Bohr radius
%E_h    Hartree energy
%ratio_h_me quantum of circulation
%m_e    electron mass
%N_A    Avogadro constant
%m_u    atomic mass constant
%F      Faraday constant
%R      molar gas constant
%k      Boltzmann constant
%V_m    molar volume of ideal gas
%n_0    Loschmidt constant
%ratio_S0_R Sackur-Tetrode constant (absolute entropy constant)
%sigma  Stefan-Boltzmann constant
%c_1    first radiation constant
%c_1L   first radiation constant for spectral radiance
%c_2    second radiation constant
%b      Wien displacement law constant
%b_prime Wien displacement law constant

References:

Examples:
The list of all symbols which have the physical_constant property.

(%i1) load (physical_constants)$
(%i2) propvars (physical_constant);
(%o2) [%c, %mu_0, %e_0, %Z_0, %G, %h, %h_bar, %m_P, %T_P, %l_P,
       %t_P, %e, %Phi_0, %G_0, %K_J, %R_K, %mu_B, %mu_N, %alpha,
       %R_inf, %a_0, %E_h, %ratio_h_me, %m_e, %N_A, %m_u, %F, %R, %k,
       %V_m, %n_0, %ratio_S0_R, %sigma, %c_1, %c_1L, %c_2, %b, %b_prime]

Properties of the physical constant %c.

(%i1) load (physical_constants)$
(%i2) constantp (%c);
(%o2) true
(%i3) get (%c, description);
speed of light in vacuum
\[c\]

The energy equivalent of 1 pound-mass. The symbol \(c\) persists until its value is fetched by constvalue.

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.

54.3 Functions and Variables for ezunits

Operator
Chapter 54: ezunits

\[(x'a) \ast (y'b) \text{ is equal to } (x \ast y)(a \ast b).\]
\[(x'a) + (y'a) \text{ is equal to } (x + y)'a.\]
\[(x'a)^y \text{ is equal to } x'^ya^y \text{ when } y \text{ is nondimensional.}\]

**ezunits** does not require that units in a sum have the same dimensions; such terms are not added together, and no error is reported.

**load(ezunits)** enables this operator.

**Examples:**

**SI (Systeme Internationale) units.**

```lisp
(%i1) load (ezunits)$
(%i2) foo : 10 ' m;
    10 ' m
(%o2)
(%i3) qty (foo);
    10
(%o3)
(%i4) units (foo);
    m
(%o4)
(%i5) dimensions (foo);
    length
```

"Customary" units.

```lisp
(%i1) load (ezunits)$
(%i2) bar : x ' acre;
    x ' acre
(%o2)
(%i3) dimensions (bar);
    2
(%o3)
(%i4) fundamental_units (bar);
    2
(%o4)
```

**Units ad hoc.**

```lisp
(%i1) load (ezunits)$
(%i2) baz : 3 ' sheep + 8 ' goat + 1 ' horse;
    8 ' goat + 3 ' sheep + 1 ' horse
(%o2)
(%i3) subst ([sheep = 3*goat, horse = 10*goat], baz);
    27 ' goat
(%o3)
(%i4) baz2 : 1000'gallon/fortnight;
    1000 ' -----
      gallon
(%o4)   1000 ' --------
      fortnight
(%i5) subst (fortnight = 14*day, baz2);
    500 ' gallon
    --- ' -------
      7 ' day
(%o5)
```

**Arithmetic operations on dimensional quantities.**

```lisp
(%i1) load (ezunits)$
(%i2) 100 ' kg + 200 ' kg;
    300 ' kg
(%o2)
(%i3) 100 ' m^-3 - 100 ' m^-3;
```

```
The unit conversion operator. An expression $a \cdot b \cdot c$ converts from unit $b$ to unit $c$. 

\texttt{ezunits} has built-in conversions for SI base units, SI derived units, and some non-SI units. Unit conversions not already known to \texttt{ezunits} can be declared. The unit conversions known to \texttt{ezunits} are specified by the global variable \texttt{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. \texttt{ezunits} does not attempt to simplify units by prefixes (milli-, centi-, deci-, etc) unless such conversion is explicitly indicated.

\texttt{load(ezunits)} enables this operator.

Examples:

The set of known unit conversions:

\begin{verbatim}
(%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,
\end{verbatim}
Chapter 54: ezunits

MPa = 1000000*Pa, ms = s/1000, MWb = 1000000*Wb, 
Mg = 1000*kg, Mm = 1000000*m, Mmol = 1000000000*mol, 
Ms = 1000000*s, ounce = pound_mass/16, 
oz = ounce, Ohm = s/J/C^2, 
pound_force = 32*ft*pound_mass/s^2, 
pound_mass = 200*pound_mass/441, psi = pound_force/inch^2, 
Pa = N/m^2, week = 604800*s, 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, GV = 1000000000*N, 
GS = 1000000000*S, GT = 1000000000*T, GW = 1000000000*W, 
GW = 1000000000*W, H = J/A^2, J = m*N, kA = 1000*A, 
kC = 1000*C, kF = 1000*F, kH = 1000*H, kJ = 1000*J, kK = 1000*K, 
kL = 1000*lbf, kW = 1000*W, 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  ---- ' m
1250
(\%(o2)
(\%(i3) \%, numer;
(\%(o3)
(\%(i4) 1 ' kg '' lbm;
441  ---- ' lbm
200
(\%(o4)
(\%(i5) \%, numer;
(\%(o5)
(\%(i6) 1 ' W '' Btu/hour;
720  Btu
211  hour
(\%(o6)
(\%(i7) \%, numer;
(\%(o7) 3.41232274881517 ' ----hour
(\%(i8) 100 ' degC '' degF;
(%o8) 212 ' degF
(%o9) -40 ' degF ' degC;
(%o10) 1 ' acre*ft ' m^-3;
(%o10) 60228605349 3
----------- ' m
48828125
(%i11) %, numer;
3
(%o11) 1233.48183754752 ' m

Coercing quantities in feet and meters to one or the other.

(%i1) load (ezunits)$
(%i2) 100 ' m + 100 ' ft;
(%o2) 100 ' m + 100 ' ft
(%i3) (100 ' m + 100 ' ft) ' ft;
  Computes conversions to base units; may take a moment.
163100
(%o3) ------ ' ft
381
(%i4) %, numer;
(%o4) 428.0839895013123 ' ft
(%i5) (100 ' m + 100 ' ft) ' m;
3262
(%o5) ---- ' m
25
(%i6) %, numer;
(%o6) 130.48 ' m

Dimensional analysis to find fundamental dimensions and fundamental units.

(%i1) load (ezunits)$
(%i2) foo : 1 ' acre * ft;
(%o2) 1 ' acre ft
(%i3) dimensions (foo);
3
(%o3) length
(%i4) fundamental_units (foo);
3
(%o4) m
(%i5) foo ' m^-3;
Computing conversions to base units; may take a moment.
60228605349 3
----------- ' m
48828125
(%i6) %, numer;
3
(%o6) 1233.48183754752 ' m

Declared unit conversions.

(%i11) load (ezunits)$
(%i12) declare_unit_conversion (MMBtu = 10^6*Btu, kW = 1000*W);
Chapter 54: ezunits

(%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

Function constvalue (x)
Function declare_constvalue (a, x)
Function remove_constvalue (a)

Returns the declared constant value of a symbol, or value of an expression with declared constant values substituted for symbols.

Constant values are declared by declare_constvalue. Note that constant values
as recognized by constvalue are separate from values declared by numerval and
recognized by constantp.

The physical_units package declares constant values for a number of physical constants.

remove_constvalue reverts the effect of declare_constvalue.

load(ezunits) loads these functions.

Examples:

Constant value of a physical constant.

(%i1) load (physical_constants)$
(%i2) constvalue (%G);

m
(%o2) 6.67428 ' ----- 2
kg s
(%i3) get ('%G, 'description);
Newtonian constant of gravitation

Declaring a new constant.

(%i1) load (ezunits)$
(%i2) declare_constvalue (FOO, 100 ' lbm / acre);

lbm
(%o2) 100 ' ---- acre
(%i3) FOO * (50 ' acre);
(%o3) 50 FOO ' acre
(%i4) constvalue (%);
(%o4) 5000 ' lbm
**units** \((x)\)  \hspace{1cm} \text{Function}

**declare_units** \((a, u)\)  \hspace{1cm} \text{Function}

Returns the units of a dimensional quantity \(x\), or returns 1 if \(x\) is nondimensional.

\(x\) may be a literal dimensional expression \(a' b\), a symbol with declared units via **declare_units**, or an expression containing either or both of those.

**declare_units** declares that **units**\((a)\) should return \(u\), where \(u\) is an expression.

**load(ezunits)** loads these functions.

Examples:

**units** applied to literal dimensional expressions.

\[
\begin{align*}
\text{(\%i1) load (ezunits)}$ & \\
\text{(\%i2) foo : 100 ' kg; } & 100 ' \text{ kg} \\
\text{(\%i3) bar : x ' m/s; } & \text{m} \quad \text{x ' - s} \\
\text{(\%i4) units (foo); } & \text{kg} \\
\text{(\%i5) units (bar); } & \text{m} - \text{s} \\
\text{(\%i6) units (foo * bar); } & \text{kg m} \quad \text{---- s} \\
\text{(\%i7) units (foo / bar); } & \text{kg s} \quad \text{---- m} \\
\text{(\%i8) units (foo^2); } & 2 \text{ kg} \\
\end{align*}
\]

**units** applied to symbols with declared units.

\[
\begin{align*}
\text{(\%i1) load (ezunits)}$ & \\
\text{(\%i2) units (aa); } & 1 \\
\text{(\%i3) declare_units (aa, J);} & J \\
\text{(\%i4) units (aa);} & J \\
\text{(\%i5) units (aa^2);} & 2 \\
\text{(\%i6) foo : 100 ' kg; } & 100 ' \text{ kg} \\
\text{(\%i7) units (aa * foo);} & \text{J}
\end{align*}
\]
Chapter 54: ezunits

Function \( \text{qty}(x) \)

Function \( \text{declare\_qty}(a, x) \)

\( \text{qty} \) returns the nondimensional part of a dimensional quantity \( x \), or returns \( x \) if \( x \) is nondimensional. \( x \) may be a literal dimensional expression \( a^b \), a symbol with declared quantity, or an expression containing either or both of those.

\( \text{declare\_qty} \) declares that \( \text{qty}(a) \) should return \( x \), where \( x \) is a nondimensional quantity.

\( \text{load(ezunits)} \) loads these functions.

Examples:

\( \text{qty} \) applied to literal dimensional expressions.

\( \%i1 \) \( \text{load (ezunits)} \)$
\( \%i2 \) \( \text{foo} : 100 \ ' \text{kg}; \)
\( \%o2 \) \( 100 \ ' \text{kg} \)
\( \%i3 \) \( \text{qty} (\text{foo}); \)
\( \%o3 \) \( 100 \)
\( \%i4 \) \( \text{bar} : v \ ' \text{m/s}; \)
\( \%o4 \) \( v \ ' \text{m} \)
\( \%i5 \) \( \text{foo} \ast \text{bar}; \)
\( \%o5 \) \( 100 \ v \ ' \text{m/s} \)
\( \%i6 \) \( \text{qty} (\text{foo} \ast \text{bar}); \)
\( \%o6 \) \( 100 \ v \)

\( \text{qty} \) applied to symbols with declared quantity.

\( \%i1 \) \( \text{load (ezunits)} \)$
\( \%i2 \) \( \text{declare\_qty} (aa, xx); \)
\( \%o2 \) \( xx \)
\( \%i3 \) \( \text{qty} (aa); \)
\( \%o3 \) \( xx \)
\( \%i4 \) \( \text{qty} (aa^2); \)
\( \%o4 \) \( xx \)
\( \%i5 \) \( \text{foo} : 100 \ ' \text{kg}; \)
\( \%o5 \) \( 100 \ ' \text{kg} \)
\( \%i6 \) \( \text{qty} (aa \ast \text{foo}); \)
\( \%o6 \) \( 100 \ xx \)

Function \( \text{unitp}(x) \)

Returns \text{true} if \( x \) is a literal dimensional expression, a symbol declared dimensional, or an expression in which the main operator is declared dimensional. \( \text{unitp} \) returns \text{false} otherwise.

\( \text{load(ezunits)} \) loads this function.
Examples:

\texttt{unitp} applied to a literal dimensional expression.
\begin{verbatim}
(%i1) load (ezunits)$
(%i2) unitp (100 ' kg);
(%o2) true
\end{verbatim}

\texttt{unitp} applied to a symbol declared dimensional.
\begin{verbatim}
(%i1) load (ezunits)$
(%i2) unitp (foo);
(%o2) false
(%i3) declare (foo, dimensional);
(%o3) done
(%i4) unitp (foo);
(%o4) true
\end{verbatim}

\texttt{unitp} applied to an expression in which the main operator is declared dimensional.
\begin{verbatim}
(%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
\end{verbatim}

\texttt{declare_unit_conversion} \((u = v, \ldots)\)

\textbf{Function}

Appends equations \(u = v, \ldots\) to the list of unit conversions known to the unit conversion operator \(^\prime\). \(u\) 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.

\texttt{known_unit_conversions} is the list of known unit conversions.

\texttt{load(ezunits)} loads this function.

Examples:

Unit conversions expressed by equations of multiplicative terms.
\begin{verbatim}
(%i1) load (ezunits)$
(%i2) declare_unit_conversion (nautical_mile = 1852 * m, fortnight = 14 * day);
(%o2) done
(%i3) 100 ' nautical_mile / fortnight '' m/s;
\end{verbatim}

Computing conversions to base units; may take a moment.
\begin{verbatim}
463 m
--- ' -
3024 s
\end{verbatim}

Unit conversions expressed by equations of literal dimensional expressions.
\begin{verbatim}
(%i1) load (ezunits)$
(%i2) declare_unit_conversion (1 ' fluid_ounce = 2 ' tablespoon);
\end{verbatim}
declare_dimensions \((a_1, d_1, \ldots, a_n, d_n)\)

The `declare_dimensions` function declares \(a_1\), \(\ldots\), \(a_n\) to have dimensions \(d_1\), \(\ldots\), \(d_n\), respectively.

Each \(a_k\) is a symbol or a list of symbols. If it is a list, then every symbol in \(a_k\) is declared to have dimension \(d_k\).

`remove_dimensions` reverts the effect of `declare_dimensions`.

`load(ezunits)` loads these functions.

Examples:

\[
\begin{align*}
(\%i1) & \text{ load (ezunits) } \\
(\%i2) & \text{ declare_dimensions ([x, y, z], length, [t, u], time)}; \\
(\%o2) & \text{ done} \\
(\%i3) & \text{ dimensions (y^2/u)}; \\
(\%o3) & \text{ length} \\
& \text{ -------} \\
& \text{ time} \\
(\%i4) & \text{ fundamental_units (y^2/u)}; \\
& 0 \text{ errors, 0 warnings} \\
(\%o4) & 2 \\
& \text{ m} \\
& \text{ --} \\
& \text{ s}
\end{align*}
\]
([%o2]) [length, mass, time, current, temperature, quantity]
(%i3) declare_fundamental_dimensions (money, cattle, happiness);
(%o3) done
(%i4) fundamental_dimensions;
(%o4) [length, mass, time, current, temperature, quantity, money, cattle, happiness]
(%i5) remove_fundamental_dimensions (cattle, happiness);
(%o5) done
(%i6) fundamental_dimensions;
(%o6) [length, mass, time, current, temperature, quantity, money]

declare_fundamental_units (u_1, d_1, ..., u_n, d_n)
Function

remove_fundamental_units (u_1, ..., u_n)
Function

declare_fundamental_units declares u_1, ..., u_n to have dimensions d_1, ..., d_n, respectively. All arguments must be symbols.

After calling declare_fundamental_units, dimensions(u_k) returns d_k for each argument u_1, ..., u_n, and fundamental_units(d_k) returns u_k for each argument d_1, ..., d_n.

remove_fundamental_units reverts the effect of declare_fundamental_units.

load(ezunits) loads these functions.

Examples:

(%i1) load (ezunits) $
(%i2) declare_fundamental_dimensions (money, cattle, happiness);
(%o2) done
(%i3) declare_fundamental_units (dollar, money, goat, cattle, smile, happiness);
(%o3) [dollar, goat, smile]
(%i4) dimensions (100 ' dollar/goat/km^2);
     money
     --------------
     2
     cattle length
(%i5) dimensions (x ' smile/kg);
     happiness
     ----------
     mass
(%i6) fundamental_units (money*cattle/happiness);
0 errors, 0 warnings
dollar goat
       ------------
       smile

dimensions (x)
Function
dimensions_as_list (x)
Function
dimensions returns the dimensions of the dimensional quantity x as an expression comprising products and powers of base dimensions.
dimensions_as_list returns the dimensions of the dimensional quantity x as a list, in which each element is an integer which indicates the power of the corresponding base dimension in the dimensions of x.
load(ezunits) loads these functions.

Examples:

(%i1) load (ezunits)$
(%i2) dimensions (1000 ' kg*m^2/s^3);

  2
---
length mass

  3
time
(%o2)

(%i3) declare_units (foo, acre*ft/hour);
acre ft

  3
hour
(%o3)

(%i4) dimensions (foo);

  3
length
time
(%o4)

(%i1) load (ezunits)$
(%i2) fundamental_dimensions;

  [length, mass, time, charge, temperature, quantity]
(%o2)

(%i3) dimensions_as_list (1000 ' kg*m^2/s^3);

  2
[2, 1, - 3, 0, 0, 0]
(%o3)

(%i4) declare_units (foo, acre*ft/hour);
acre ft

  3
hour
(%o4)

(%i5) dimensions_as_list (foo);

  3,
  0,
  - 1,
  0,
  0,
  0
(%o5)

fundamental_units (x) Function

fundamental_units() Function

fundamental_units(x) returns the units associated with the fundamental dimensions of x, as determined by dimensions(x).

x may be a literal dimensional expression a'b, a symbol with declared units via declare_units, or an expression containing either or both of those.

fundamental_units() returns the list of all known fundamental units, as declared by declare_fundamental_units.

load(ezunits) loads this function.

Examples:

(%i1) load (ezunits)$
(%i2) fundamental_units ();

  [m, kg, s, A, K, mol]
(%o2)

(%i3) fundamental_units (100 ' mile/hour);

  m
(%o3)

  s
(%i4) declare_units (aa, g/foot^2);
  g
(%o4) ----
  2
foot
(%i5) fundamental_units (aa);
  kg
(%o5) --
  2
  m

dimensionless (L)

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)
  2
  \%\%e \%m_e
  [----------------------, ----]
  \%c \%e_0 \%h_bar \%m_P

natural_unit (expr, [v_1, ..., v_n])

Function
Finds exponents e_1, ..., e_n such that \text{dimension}(expr) = \text{dimension}(v_1^{e_1} \cdot \cdot \cdot v_n^{e_n}).
load(ezunits) loads this function.
Examples:
Chapter 55: f90

55 f90

55.1 Functions and Variables for f90

\textbf{f90} (\textit{expr}_1, ..., \textit{expr}_n) \hspace{1cm} \textit{Function}

Prints one or more expressions \textit{expr}_1, ..., \textit{expr}_n as a Fortran 90 program. Output is printed to the standard output.

\textit{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.

\textbf{load(f90)} loads this function.

Examples:

\begin{verbatim}
(%i1) load (f90)$
(%i2) foo : expand ((xxx + yyy + 7)^4);
   4 3 3 2 2 2
   |   |   |   |
   / 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**2 + 6*xxx**2 + 84*xxx*yyy**2 +
+ 294*yyy**2 + 4*xxx***3 + 84*xxx**2 + 588*xxx*yyy + 1372*yyy +
+ 4*xxx***3 + 294*xxx**2 + 1372*xxx + 2401

(%o3) false
\end{verbatim}

Multiple expressions. Capture standard output into a file via the \textbf{with_stdout} function.

\begin{verbatim}
(%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
\end{verbatim}
56 finance

56.1 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.

56.2 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:

```wolfram
(%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:

```wolfram
(%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:

```wolfram
(%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:

```wolfram
(%i1) load(finance)$
(%i2) graph_flow([-5000,-3000,800,1300,1500,2000])$
```
annuity_pv \((rate,PV,num)\)
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:

\[
\begin{align*}
(\%i1) & \text{load(finance)}$
(\%i2) & \text{annuity_pv(0.12,5000,10)}; \\
(\%o2) & 884.9208207992202
\end{align*}
\]

annuity_fv \((rate,FV,num)\)
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:

\[
\begin{align*}
(\%i1) & \text{load(finance)}$
(\%i2) & \text{annuity_fv(0.12,65000,10)}; \\
(\%o2) & 3703.970670389863
\end{align*}
\]

geo_annuity_pv \((rate,\text{growing rate},PV,num)\)
We can calculate the annuity knowing the present value (like an amount), in a growing periodic payment. \(rate\) is the interest rate, \(\text{growing rate}\) is the growing rate, \(PV\) is the present value and \(num\) is the number of periods.
Example:

\[
\begin{align*}
(\%i1) & \text{load(finance)}$
(\%i2) & \text{geo_annuity_pv(0.14,0.05,5000,10)}; \\
(\%o2) & 802.6888176505123
\end{align*}
\]

geo_annuity_fv \((rate,\text{growing rate},FV,num)\)
We can calculate the annuity knowing the desired value (future value), in a growing periodic payment. \(rate\) is the interest rate, \(\text{growing rate}\) is the growing rate, \(FV\) is the future value and \(num\) is the number of periods.
Example:

\[
\begin{align*}
(\%i1) & \text{load(finance)}$
(\%i2) & \text{geo_annuity_fv(0.14,0.05,5000,10)}; \\
(\%o2) & 216.5203395312695
\end{align*}
\]

amortization \((rate,\text{ammount},num)\)
Amortization table determined by a specific rate. \(rate\) is the interest rate, \(\text{ammount}\) is the ammount value, and \(num\) is the number of periods.
Example:

\[
\begin{align*}
(\%i1) & \text{load(finance)}$
(\%i2) & \text{amortization(0.05,56000,12)}$
\begin{tabular}{llllll}
"n" & "Balance" & "Interest" & "Amortization" & "Payment"
\hline
0.000 & 56000.000 & 0.000 & 0.000 & 0.000 \\
1.000 & 52481.777 & 2800.000 & 3518.223 & 6318.223 \\
\end{tabular}
\end{align*}
\]
arit_amortization \( (\text{rate, increment, ammount, num}) \)

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. \textit{rate} is the interest rate, \textit{increment} is the increment, \textit{ammount} is the ammount value, and \textit{num} is the number of periods.

Example:

```lisp
(%i1) load(finance)$
(%i2) arit_amortization(0.05,1000,56000,12)$
```

```

<table>
<thead>
<tr>
<th>n</th>
<th>Balance</th>
<th>Interest</th>
<th>Amortization</th>
<th>Payment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>56000.00</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>1.00</td>
<td>57403.68</td>
<td>2800.000</td>
<td>-1403.68</td>
<td>1396.32</td>
</tr>
<tr>
<td>2.00</td>
<td>57877.54</td>
<td>2870.184</td>
<td>-473.86</td>
<td>2396.32</td>
</tr>
<tr>
<td>3.00</td>
<td>57375.09</td>
<td>2893.877</td>
<td>502.444</td>
<td>3396.32</td>
</tr>
<tr>
<td>4.00</td>
<td>55847.53</td>
<td>2868.755</td>
<td>1527.567</td>
<td>4396.32</td>
</tr>
<tr>
<td>5.00</td>
<td>53243.58</td>
<td>2792.377</td>
<td>2603.945</td>
<td>5396.32</td>
</tr>
<tr>
<td>6.00</td>
<td>49509.44</td>
<td>2662.179</td>
<td>6166.892</td>
<td>6396.32</td>
</tr>
<tr>
<td>7.00</td>
<td>44588.59</td>
<td>2475.472</td>
<td>4920.849</td>
<td>7396.32</td>
</tr>
<tr>
<td>8.00</td>
<td>38421.70</td>
<td>2229.430</td>
<td>6166.892</td>
<td>8396.32</td>
</tr>
<tr>
<td>9.00</td>
<td>30946.46</td>
<td>1921.085</td>
<td>7475.236</td>
<td>9396.32</td>
</tr>
<tr>
<td>10.00</td>
<td>22097.47</td>
<td>1547.323</td>
<td>10291.448</td>
<td>10396.32</td>
</tr>
<tr>
<td>11.00</td>
<td>11806.02</td>
<td>1104.873</td>
<td>11806.020</td>
<td>11396.32</td>
</tr>
<tr>
<td>12.00</td>
<td>0.0000</td>
<td>590.301</td>
<td>6017.355</td>
<td>6318.223</td>
</tr>
</tbody>
</table>
```

geo_amortization \( (\text{rate, growing_rate, ammount, num}) \)

The amortization table determined by rate, ammount, and number of periods can be found by geo_amortization. Notice that the payment is not constant, it presents a geometric growing, \textit{growing_rate} is then the quotient between two consecutive rows in the "Payment" column. \textit{rate} is the interest rate, \textit{ammount} is the ammount value, and \textit{num} is the number of periods.

Example:

```lisp
(%i1) load(finance)$
(%i2) geo_amortization(0.05,0.03,56000,12)$
```

```

<table>
<thead>
<tr>
<th>n</th>
<th>Balance</th>
<th>Interest</th>
<th>Amortization</th>
<th>Payment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>56000.00</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>1.00</td>
<td>57403.68</td>
<td>2800.000</td>
<td>-1403.68</td>
<td>1396.32</td>
</tr>
<tr>
<td>2.00</td>
<td>57877.54</td>
<td>2870.184</td>
<td>-473.86</td>
<td>2396.32</td>
</tr>
<tr>
<td>3.00</td>
<td>57375.09</td>
<td>2893.877</td>
<td>502.444</td>
<td>3396.32</td>
</tr>
<tr>
<td>4.00</td>
<td>55847.53</td>
<td>2868.755</td>
<td>1527.567</td>
<td>4396.32</td>
</tr>
<tr>
<td>5.00</td>
<td>53243.58</td>
<td>2792.377</td>
<td>2603.945</td>
<td>5396.32</td>
</tr>
<tr>
<td>6.00</td>
<td>49509.44</td>
<td>2662.179</td>
<td>6166.892</td>
<td>6396.32</td>
</tr>
<tr>
<td>7.00</td>
<td>44588.59</td>
<td>2475.472</td>
<td>4920.849</td>
<td>7396.32</td>
</tr>
<tr>
<td>8.00</td>
<td>38421.70</td>
<td>2229.430</td>
<td>6166.892</td>
<td>8396.32</td>
</tr>
<tr>
<td>9.00</td>
<td>30946.46</td>
<td>1921.085</td>
<td>7475.236</td>
<td>9396.32</td>
</tr>
<tr>
<td>10.00</td>
<td>22097.47</td>
<td>1547.323</td>
<td>10291.448</td>
<td>10396.32</td>
</tr>
<tr>
<td>11.00</td>
<td>11806.02</td>
<td>1104.873</td>
<td>11806.020</td>
<td>11396.32</td>
</tr>
<tr>
<td>12.00</td>
<td>0.0000</td>
<td>590.301</td>
<td>6017.355</td>
<td>6318.223</td>
</tr>
</tbody>
</table>
```
saving \((rate, ammount, num)\)

The table that represents the values in a constant and periodic saving can be found by `saving`. `ammount` represents the desired quantity and `num` the number of periods to save.

Example:

```
(%i1) load(finance)$
(%i2) saving(0.15, 12000, 15)$
```

```
"n"  "Balance"  "Interest"  "Payment"
0.000  0.000     0.000       0.000
1.000  252.205   0.000       252.205
2.000  542.240   37.831      252.205
3.000  875.781   81.336      252.205
4.000  1259.352  131.367     252.205
5.000  1700.460  188.903     252.205
6.000  2207.733  255.069     252.205
7.000  2791.098  331.160     252.205
8.000  3461.967  418.665     252.205
9.000  4233.467  519.295     252.205
10.000 5120.692  635.020     252.205
11.000 6141.000  768.104     252.205
12.000 7314.355  921.150     252.205
13.000 8663.713 1097.153     252.205
14.000 10215.474 1299.557    252.205
15.000 12000.000 1532.321    252.205
```

npv \((rate, val)\)

Calculates the present value of a value series to evaluate the viability in a project. `flowValues` es una lista con los valores para cada periodo.

Example:

```
(%i1) load(finance)$
(%i2) npv(0.25, [100, 500, 323, 124, 300]);
```

```
(%o2) 714.4703999999999
```
Function \texttt{irr}(\textit{val},IO)

IRR (Internal Rate of Return) is the value of rate which makes Net Present Value zero. \textit{flowValues} los valores para cada periodo (para periodos mayores a 0) y \textit{I0} el valor para el periodo cero.

Example:

\begin{verbatim}
(%i1) load(finance)$
(%i2) res:irr([-5000,0,800,1300,1500,2000],0)$
(%i3) rhs(res[1][1]);
(%o3) .03009250374237132
\end{verbatim}

Function \texttt{benefit\_cost}(\textit{rate},\textit{input},\textit{output})

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. \textit{rate} is the interest rate, \textit{input} is a list of input values, and \textit{output} is a list of output values.

Example:

\begin{verbatim}
(%i1) load(finance)$
(%i2) benefit_cost(0.24,[0,300,500,150],[100,320,0,180]);
(%o2) 1.427249324905784
\end{verbatim}
57 fractals

57.1 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

57.2 Definitions for IFS fractals

Some fractals can be generated by iterative applications of contractive affine transformations in a random way; see


We define a list with several contractive affine transformations, and we randomly select the transformation in a recursive way. The probability of the choice of a transformation must be related with the contraction ratio.

You can change the transformations and find another fractal

sierpinskiale (n)

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,sierpinskiale(n)], [style,dots])$

Function

Treefale (n)

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])$
fernfale \((n)\)  
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: 
\[
\text{(\%i1) load(fractals)}$
\text{(\%i2) n: 10000}$
\text{(\%i3) plot2d([discrete,fernfale(n)], [style,dots])}$

57.3 Definitions for complex fractals 

mandelbrot_set \((x, y)\)  
Mandelbrot set. 
Example: 
This program is time consuming because it must make a lot of operations; the com-
puting time is also related with the number of grid points. 
\[
\text{(\%i1) load(fractals)}$
\text{(\%i2) plot3d (mandelbrot_set, [x, -2.5, 1], [y, -1.5, 1.5],}
\text{ [gnuplot_preamble, "set view map"],}
\text{ [gnuplot_pm3d, true],}
\text{ [grid, 150, 150])}$

julia_set \((x, y)\)  
Julia sets. 
This program is time consuming because it must make a lot of operations; the com-
puting time is also related with the number of grid points. 
Example: 
\[
\text{(\%i1) load(fractals)}$
\text{(\%i2) plot3d (julia_set, [x, -2, 1], [y, -1.5, 1.5],}
\text{ [gnuplot_preamble, "set view map"],}
\text{ [gnuplot_pm3d, true],}
\text{ [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 
\(-.745+\%i*.113002, -.39054-\%i*.58679, -.15652+\%i*.03225, -.194+\%i*.6557\) 
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.

### 57.4 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)])$
```

### 57.5 Definitions for Peano maps

Continuous curves that cover an area. Warning: the number of points exponentially grows with n.

**hilbertmap** (nn) Function

Hilbert map. Argument nn must be small (5, for example). Maxima can crash if nn is 7 or greater.

Example:
```
(%i1) load(fractals)$
(%i2) plot2d([discrete, hilbertmap(6)])$
```

**sierpinskimap** (nn) Function

Sierpinski map. Argument nn must be small (5, for example). Maxima can crash if nn is 7 or greater.

Example:
```
(%i1) load(fractals)$
(%i2) plot2d([discrete, sierpinskimap(6)])$
```
58  ggf

58.1 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**  
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")`.  

Chapter 59: graphs

59 graphs

59.1 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).

59.2 Functions and Variables for graphs

59.2.1 Building graphs

create_graph (v_list, e_list) Function
create_graph (n, e_list) Function
create_graph (v_list, e_list, directed) Function

Creates a new graph on the set of vertices $v_list$ and with edges $e_list$.

$v_list$ is a list of vertices ([v1, v2, ..., vn]) or a list of vertices together with vertex labels ([[v1,l1], [v2,l2], ..., [vn,ln]]).

$n$ is the number of vertices. Vertices will be identified by integers from 0 to n-1.

$e_list$ is a list of edges ([e1, e2, ..., em]) or a list of edges together with edge-weights ([[e1, w1], ..., [em, wm]]).

If directed is not false, a directed graph will be returned.

Example 1: create a cycle on 3 vertices:

```maxima
(%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 ()
    Returns the Clebsch graph.

complement_graph (g)
    Returns the complement of the graph g.

complete_bipartite_graph (n, m)
    Returns the complete bipartite graph on n+m vertices.

complete_graph (n)
    Returns the complete graph on n vertices.

cycle_digraph (n)
    Returns the directed cycle on n vertices.

cycle_graph (n)
    Returns the cycle on n vertices.

cuboctahedron_graph (n)
    Returns the cuboctahedron graph.

cube_graph (n)
    Returns the n-dimensional cube.

dodecahedron_graph ()
    Returns the dodecahedron graph.

empty_graph (n)
    Returns the empty graph on n vertices.

flower_snark (n)
    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)
    Returns the graph represented by its adjacency matrix A.

frucht_graph ()
    Returns the Frucht graph.
**graph_product** \((g1, g1)\)

Returns the direct product of graphs \(g1\) and \(g2\).

Example:

```lisp
(%i1) load (graphs)$
(%i2) grid : graph_product(path_graph(3), path_graph(4))$
(%i3) draw_graph(grid)$
```

**graph_union** \((g1, g1)\)

Returns the union (sum) of graphs \(g1\) and \(g2\).

**grid_graph** \((n, m)\)

Returns the \(n \times m\) grid.

**great_rhombicosidodecahedron_graph** ()

Returns the great rhombicosidodecahedron graph.

**great_rhombicuboctahedron_graph** ()

Returns the great rhombicuboctahedron graph.

**grotzch_graph** ()

Returns the Grotzch graph.

**heawood_graph** ()

Returns the Heawood graph.

**icosahedron_graph** ()

Returns the icosahedron graph.

**icosidodecahedron_graph** ()

Returns the icosidodecahedron graph.

**induced_subgraph** \((V, g)\)

Returns the graph induced on the subset \(V\) of vertices of the graph \(g\).

Example:

```lisp
(%i11) load (graphs)$
(%i12) p : petersen_graph()$
(%i13) V : [0,1,2,3,4]$  
(%i14) g : induced_subgraph(V, p)$
(%i15) 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** \((g)\)  
Returns the line graph of the graph \(g\).

**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 \(f(a,b)=true\).  
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()); \(\%o3\)  
\%(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);$  
\%(i5) in_neighbors(18, g);  
\{8, 12, 16, 20\}

**mycielski_graph** \((g)\)  
Returns the mycielskian graph of the graph \(g\).

**new_graph** ()  
Returns the graph with no vertices and no edges.

**path_digraph** \((n)\)  
Returns the directed path on \(n\) vertices.

**path_graph** \((n)\)  
Returns the path on \(n\) vertices.

**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** \((a, b, p)\)  
Returns a random bipartite graph on \(a+b\) vertices. Each edge is present with probability \(p\).
Function \texttt{random} \texttt{digraph} \texttt{(n, p)}

Returns a random directed graph on \( n \) vertices. Each arc is present with probability \( p \).

Function \texttt{random} \texttt{regular} \texttt{graph} \texttt{(n)}

Function \texttt{random} \texttt{regular} \texttt{graph} \texttt{(n, d)}

Returns a random \( d \)-regular graph on \( n \) vertices. The default value for \( d \) is \( d=3 \).

Function \texttt{random} \texttt{graph} \texttt{(n, p)}

Returns a random graph on \( n \) vertices. Each edge is present with probability \( p \).

Function \texttt{random} \texttt{graph1} \texttt{(n, m)}

Returns a random graph on \( n \) vertices and random \( m \) edges.

Function \texttt{random} \texttt{network} \texttt{(n, p, w)}

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 \([\text{network}, \text{source}, \text{sink}]\).

Example:

\begin{verbatim}
(%i1) load (graphs)$
(%i2) [net, s, t] : random_network(50, 0.2, 10.0); 
    [DIGRAPH, 50, 51]
(%i3) max_flow(net, s, t)$
(%i4) first(%);
    27.65981397932507
\end{verbatim}

Function \texttt{random} \texttt{tournament} \texttt{(n)}

Returns a random tournament on \( n \) vertices.

Function \texttt{random} \texttt{tree} \texttt{(n)}

Returns a random tree on \( n \) vertices.

Function \texttt{small} \texttt{rhombicosidodecahedron} \texttt{graph} \texttt{()}

Returns the small rhombicosidodecahedron graph.

Function \texttt{small} \texttt{rhombicuboctahedron} \texttt{graph} \texttt{()}

Returns the small rhombicuboctahedron graph.

Function \texttt{snub} \texttt{cube} \texttt{graph} \texttt{()}

Returns the snub cube graph.

Function \texttt{snub} \texttt{dodecahedron} \texttt{graph} \texttt{()}

Returns the snub dodecahedron graph.

Function \texttt{truncated} \texttt{cube} \texttt{graph} \texttt{()}

Returns the truncated cube graph.
Function \texttt{truncated\_dodecahedron\_graph ()}
\hspace{1em} Returns the truncated dodecahedron graph.

Function \texttt{truncated\_icosahedron\_graph ()}
\hspace{1em} Returns the truncated icosahedron graph.

Function \texttt{truncated\_tetrahedron\_graph ()}
\hspace{1em} Returns the truncated tetrahedron graph.

Function \texttt{tutte\_graph ()}
\hspace{1em} Returns the Tutte graph.

Function \texttt{underlying\_graph (g)}
\hspace{1em} Returns the underlying graph of the directed graph \(g\).

Function \texttt{wheel\_graph (n)}
\hspace{1em} Returns the wheel graph on \(n+1\) vertices.

\subsection*{59.2.2 Graph properties}

Function \texttt{adjacency\_matrix (gr)}
\hspace{1em} Returns the adjacency matrix of the graph \(gr\).
\begin{verbatim}
Example:
  (%i1) load (graphs)$
  (%i2) c5 : cycle_graph(4)$
  (%i3) adjacency_matrix(c5);
\end{verbatim}
\begin{verbatim}
   [ 0 1 0 1 ]
   [         ]
   [ 1 0 1 0 ]
  [         ]
   [ 0 1 0 1 ]
  [         ]
   [ 1 0 1 0 ]
\end{verbatim}

Function \texttt{average\_degree (gr)}
\hspace{1em} Returns the average degree of vertices in the graph \(gr\).
\begin{verbatim}
Example:
  (%i1) load (graphs)$
  (%i2) average_degree(grotzch_graph());
\end{verbatim}
\begin{verbatim}
   40
   --
   11
\end{verbatim}

Function \texttt{biconnected\_components (gr)}
\hspace{1em} Returns the (vertex sets of) 2-connected components of the graph \(gr\).
\begin{verbatim}
Example:
\end{verbatim}
(%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)
Returns a bipartition of the vertices of the graph gr or an empty list if gr is not
bipartite.
Example:
(%i1) load (graphs)$
(%i2) h : heawood_graph()$
(%i3) [A, B]:bipartition(h);
(%o3) [[8, 12, 6, 10, 0, 2, 4], [13, 5, 11, 7, 9, 1, 3]]
(%i4) draw_graph(h, show_vertices=A, program=circular)$

chromatic_index (gr)
Returns the chromatic index of the graph gr.
Example:
(%i1) load (graphs)$
(%i2) p : petersen_graph()$
(%i3) chromatic_index(p);
(%o3) 4

chromatic_number (gr)
Returns the chromatic number of the graph gr.
Example:
(%i1) load (graphs)$
(%i2) chromatic_number(cycle_graph(5));
(%o2) 3
(%i3) chromatic_number(cycle_graph(6));
(%o3) 2

clear_edge_weight (e, gr)
Removes the weight of the edge e in the graph gr.
Example:
(%i1) load (graphs)$
(%i2) g : create_graph(3, [[[0, 1], 1.5], [[1, 2], 1.3]])$
(%i3) get_edge_weight([0, 1], g);
(%o3) 1.5
(%i4) clear_edge_weight([0, 1], g)$
(%i5) get_edge_weight([0, 1], g);
(%o5) 1
clear_vertex_label \((v, gr)\)  
Removes the label of the vertex \(v\) in the graph \(gr\).
Example:
\[
\begin{align*}
(\%i1) & \quad \text{load (graphs)}$
(\%i2) & \quad g : \text{create_graph([[0,"Zero"], [1, "One"]], [[0,1]])}$
(\%i3) & \quad \text{get_vertex_label(0, g);}
(\%o3) & \quad \text{Zero}
(\%i4) & \quad \text{clear_vertex_label(0, g);}
(\%o4) & \quad \text{done}
(\%i5) & \quad \text{get_vertex_label(0, g);}
(\%o5) & \quad \text{false}
\end{align*}
\]

connected_components \((gr)\)  
Returns the (vertex sets of) connected components of the graph \(gr\).
Example:
\[
\begin{align*}
(\%i1) & \quad \text{load (graphs)}$
(\%i2) & \quad g : \text{graph_union(cycle_graph(5), path_graph(4))}$
(\%i3) & \quad \text{connected_components(g);}
(\%o3) & \quad [[1, 2, 3, 4, 0], [8, 7, 6, 5]]
\end{align*}
\]

diameter \((gr)\)  
Returns the diameter of the graph \(gr\).
Example:
\[
\begin{align*}
(\%i1) & \quad \text{load (graphs)}$
(\%i2) & \quad \text{diameter(dodecahedron_graph());}
(\%o2) & \quad 5
\end{align*}
\]

degree_sequence \((gr)\)  
Returns the list of vertex degrees of the graph \(gr\).
Example:
edge_connectivity (gr)
Returns the edge-connectivity of the graph gr.
See also min_edge_cut.

degrees (gr)
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]]

g_value (e, gr)
g_value (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)
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)
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) (x + 2)
**graph_center** (gr)

Returns the center of the graph gr.

Example:

```lisp
(%i1) load (graphs)$
(%i2) g : grid_graph(5,5)$
(%i3) graph_center(g);  [12]
```

**graph_eigenvalues** (gr)

Returns the eigenvalues of the graph gr. The function returns eigenvalues in the same format as maxima eigenvalue function.

Example:

```lisp
(%i1) load (graphs)$
(%i2) p : petersen_graph()$
(%i3) graph_eigenvalues(p);
(%o3)  [[3, - 2, 1], [1, 4, 5]]
```

**graph_periphery** (gr)

Returns the periphery of the graph gr.

Example:

```lisp
(%i1) load (graphs)$
(%i2) g : grid_graph(5,5)$
(%i3) graph_periphery(g);  [24, 20, 4, 0]
```

**graph_size** (gr)

Returns the number of edges in the graph gr.

Example:

```lisp
(%i1) load (graphs)$
(%i2) p : petersen_graph()$
(%i3) graph_size(p);  15
```

**graph_order** (gr)

Returns the number of vertices in the graph gr.

Example:

```lisp
(%i1) load (graphs)$
(%i2) p : petersen_graph()$
(%i3) graph_order(p);  10
```

**girth** (gr)

Returns the length of the shortest cycle in gr.

Example:
Function hamilton_cycle (gr)
Returns the Hamilton cycle of the graph gr or an empty list if gr is not hamiltonian.
Example:
(%i1) load (graphs)$
(%i2) g : heawood_graph()$
(%i3) girth(g);
(%o3) 6

Function hamilton_path (gr)
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]

Function isomorphism (gr1, gr2)
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]

Function in_neighbors (v, gr)
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) []

Function is_biconnected (gr)
Returns true if gr is 2-connected and false otherwise.
Example:
Chapter 59: graphs

(%i1) load (graphs)$
(%i2) is_biconnected(cycle_graph(5));
   true
(%i3) is_biconnected(path_graph(5));
   false

is_bipartite (gr)
  Returns true if gr is bipartite (2-colorable) and false otherwise.
Example:
(%i1) load (graphs)$
(%i2) is_bipartite(petersen_graph());
   false
(%i3) is_bipartite(heawood_graph());
   true

is_connected (gr)
  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)));
   false

is_digraph (gr)
  Returns true if gr is a directed graph and false otherwise.
Example:
(%i1) load (graphs)$
(%i2) is_digraph(path_graph(5));
   false
(%i3) is_digraph(path_digraph(5));
   true

is_edge_in_graph (e, gr)
  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);
   true
(%i4) is_edge_in_graph([3,2], c4);
   true
(%i5) is_edge_in_graph([2,4], c4);
   false
(%i6) is_edge_in_graph([3,2], cycle_digraph(4));
   false
Function

**is_graph** (gr)

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
```

Function

**is_graph_or_digraph** (gr)

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
```

Function

**is_isomorphic** (gr1, gr2)

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
```

Function

**is_planar** (gr)

Returns true if gr is a planar graph and false otherwise.

The algorithm used is the Demoucron’s algorithm, which is a quadratic time algo-

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
```

Function

**is_sconnected** (gr)

Returns true if the directed graph gr is strongly connected and false otherwise.

Example:
(\%i1) \text{load (graphs)}$
(\%i2) \text{is\_sconnected(cycle\_digraph(5));}
(\%o2) \text{true}
(\%i3) \text{is\_sconnected(path\_digraph(5));}
(\%o3) \text{false}

\text{is\_vertex\_in\_graph (v, gr)} \quad \text{Function}
\begin{align*}
\text{Returns true if } v \text{ is a vertex in the graph } g \text{ and false otherwise.}
\end{align*}
\text{Example:}
(\%i1) \text{load (graphs)}$
(\%i2) \text{c4 : cycle\_graph(4)}$
(\%i3) \text{is\_vertex\_in\_graph(0, c4);}
(\%o3) \text{true}
(\%i4) \text{is\_vertex\_in\_graph(6, c4);}
(\%o4) \text{false}

\text{is\_tree (gr)} \quad \text{Function}
\begin{align*}
\text{Returns true if } gr \text{ is a tree and false otherwise.}
\end{align*}
\text{Example:}
(\%i1) \text{load (graphs)}$
(\%i2) \text{is\_tree(random\_tree(4));}
(\%o2) \text{true}
(\%i3) \text{is\_tree(graph\_union(random\_tree(4), random\_tree(5)));}
(\%o3) \text{false}

\text{laplacian\_matrix (gr)} \quad \text{Function}
\begin{align*}
\text{Returns the laplacian matrix of the graph } gr.
\end{align*}
\text{Example:}
(\%i1) \text{load (graphs)}$
(\%i2) \text{laplacian\_matrix(cycle\_graph(5));}
(\%o2) \begin{bmatrix}
2 & -1 & 0 & 0 & -1 \\
-1 & 2 & -1 & 0 & 0 \\
0 & -1 & 2 & -1 & 0 \\
-1 & 0 & 0 & -1 & 2
\end{bmatrix}

\text{max\_clique (gr)} \quad \text{Function}
\begin{align*}
\text{Returns a maximum clique of the graph } gr.
\end{align*}
\text{Example:}
(\%i1) \text{load (graphs)}$
(\%i2) \text{g : random\_graph(100, 0.5)}$
(\%i3) \text{max\_clique(g);}
(\%o3) \begin{bmatrix}
6, 12, 31, 36, 52, 59, 62, 63, 80
\end{bmatrix}
Function

**max_degree** *(gr)*

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]
(%o4) vertex_degree(95, g); 2
```

Function

**max_flow** *(net, s, t)*

Returns a maximum flow through the network *net* with the source *s* and the sink *t*.

The function returns the value of the maximal flow and a list representing the weights of the arcs in the optimal flow.

Example:

```
(%i1) load (graphs)$
(%i2) net : create_graph(
    [1,2,3,4,5,6],
    [[[1,2], 1.0],
     [[1,3], 0.3],
     [[2,4], 0.2],
     [[2,5], 0.3],
     [[3,4], 0.1],
     [[3,5], 0.1],
     [[4,6], 1.0],
     [[5,6], 1.0]],
    directed=true)$
(%i3) [flow_value, flow] : max_flow(net, 1, 6);
(%o3) [0.7, [[[1, 2], 0.5], [[1, 3], 0.2], [[2, 4], 0.2],
     [[2, 5], 0.3], [[3, 4], 0.1], [[3, 5], 0.1], [[4, 6], 0.3],
     [[5, 6], 0.4]]
(%i4) fl : 0$
(%i5) for u in out_neighbors(1, net)
   do fl : fl + assoc([1, u], flow)$
(%o5) fl;
(%o6) 0.7
```

Function

**max_independent_set** *(gr)*

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)**

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)**

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)**

Returns the minimum edge cut in the graph \( gr \).

See also `edge_connectivity`.

**min_vertex_cover (gr)**

Returns the minimum vertex cover of the graph \( gr \).

**min_vertex_cut (gr)**

Returns the minimum vertex cut in the graph \( gr \).

See also `vertex_connectivity`.

**minimum_spanning_tree (gr)**

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)**

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** \((gr)\)

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);
     4
(%o3) 4
(%i4) odd_girth(g);
     7
(%o4) 7
```

**out_neighbors** \((v, gr)\)

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);
    [1]
(%o3) [1]
(%i4) out_neighbors(2, p);
    []
(%o4) []
```

**planar_embedding** \((gr)\)

Returns the list of facial walks in a planar embedding of \(gr\) and \texttt{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** \((gr)\)

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 : digraph(5)$
```
Chapter 59: graphs

(%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)
Returns the radius of the graph gr.
Example:
(%i1) load (graphs)$
(%i2) radius(dodecahedron_graph());
(%o2) 5

set_edge_weight (e, w, gr)
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)
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)
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 \((u, v, gr)\)

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 \((gr)\)

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 \((dag)\)

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 \((g)\)

Returns the vertex connectivity of the graph \(g\).

See also min_vertex_cut.
vertex_degree \((v, gr)\) \hspace{1cm} \text{Function}

Returns the degree of the vertex \(v\) in the graph \(gr\).

vertex_distance \((u, v, gr)\) \hspace{1cm} \text{Function}

Returns the length of the shortest path between \(u\) and \(v\) in the (directed) graph \(gr\).

Example:

\begin{verbatim}
(%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]
\end{verbatim}

vertex_eccentricity \((v, gr)\) \hspace{1cm} \text{Function}

Returns the eccentricity of the vertex \(v\) in the graph \(gr\).

Example:

\begin{verbatim}
(%i1) load (graphs)$
(%i2) g:cycle_graph(7)$
(%i3) vertex_eccentricity(0, g); \\
   (%o3) 3
\end{verbatim}

vertex_in_degree \((v, gr)\) \hspace{1cm} \text{Function}

Returns the in-degree of the vertex \(v\) in the directed graph \(gr\).

Example:

\begin{verbatim}
(%i1) load (graphs)$
(%i2) p5 : path_digraph(5)$
(%i3) print_graph(p5)$
   Digraph on 5 vertices with 4 arcs.
   Adjacencies:
   4 : 3
   3 : 4
   2 : 3
   1 : 2
   0 : 1
(%i4) vertex_in_degree(4, p5); \\
   (%o4) 1
(%i5) in_neighbors(4, p5); \\
   (%o5) [3]
\end{verbatim}

vertex_out_degree \((v, gr)\) \hspace{1cm} \text{Function}

Returns the out-degree of the vertex \(v\) in the directed graph \(gr\).

Example:

\begin{verbatim}
(%i1) load (graphs)$
(%i2) t : random_tournament(10)$
(%i3) vertex_out_degree(0, t); \\
   (%o3) 2
(%i4) out_neighbors(0, t); \\
   (%o4) [7, 1]
\end{verbatim}
Function \texttt{vertices} \((\text{gr})\)

Returns the list of vertices in the graph \(\text{gr}\).

Example:

```maxima
(%i1) load (graphs)$
(%i2) vertices(complete_graph(4));
(%o2) [3, 2, 1, 0]
```

Function \texttt{vertex\_coloring} \((\text{gr})\)

Returns an optimal coloring of the vertices of the graph \(\text{gr}\).

The function returns the chromatic number and a list representing the coloring of the vertices of \(\text{gr}\).

Example:

```maxima
(%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]]]
```

Function \texttt{wiener\_index} \((\text{gr})\)

Returns the Wiener index of the graph \(\text{gr}\).

Example:

```maxima
(%i2) wiener_index(dodecahedron_graph());
(%o2) 500
```

### 59.2.3 Modifying graphs

Function \texttt{add\_edge} \((e, \text{gr})\)

Adds the edge \(e\) to the graph \(\text{gr}\).

Example:

```maxima
(%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]
```

Function \texttt{add\_edges} \((e\_list, \text{gr})\)

Adds all edges in the list \(e\_list\) to the graph \(\text{gr}\).

Example:

```maxima
(%i1) load (graphs)$
(%i2) g : empty_graph(3)$
(%i3) add_edges([[0,1],[1,2]], g)$
```
add_vertex \ (v, \ gr) \\
\text{Function} \\
\text{Adds the vertex} \ v \ \text{to the graph} \ gr. \\
\text{Example:} \\
(%i1) \ \text{load} \ (\text{graphs})$
(%i2) \ g : \ \text{path}\_\text{graph}(2)$
(%i3) \ \text{add}\_\text{vertex}(2, \ g)$
(%i4) \ \text{print}\_\text{graph}(g)$
\text{Graph on 3 vertices with 1 edges.} \\
\text{Adjacencies:} \\
2 : 1 \\
1 : 0 \\
0 : 1 \\
add\_\text{vertices} \ (v\_\text{list}, \ gr) \\
\text{Function} \\
\text{Adds all vertices in the list} \ v\_\text{list} \ \text{to the graph} \ gr. \\
\text{Example:} \\
(%i1) \ \text{load} \ (\text{graphs})$
(%i2) \ g : \ \text{empty}\_\text{graph}(4)$
(%i3) \ \text{connect}\_\text{vertices}(0, \ [1,2,3], \ g)$
(%i4) \ \text{print}\_\text{graph}(g)$
\text{Graph on 4 vertices with 3 edges.} \\
\text{Adjacencies:} \\
3 : 0 \\
2 : 0 \\
1 : 0 \\
0 : 3 2 1 \\
contract\_\text{edge} \ (e, \ gr) \\
\text{Function} \\
\text{Contracts the edge} \ e \ \text{in the graph} \ gr. \\
\text{Example:} \\
(%i1) \ \text{load} \ (\text{graphs})$
(%i2) \ g : \ \text{create}\_\text{graph}( \\
8, \ \text{[[0,3],[1,3],[2,3],[3,4],[4,5],[4,6],[4,7]]} \ )$
(%i3) \ \text{print}\_\text{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)  
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)  
Removes the vertex v from the graph gr.

59.2.4 Reading and writing to files

dimacs_export (gr, fl)  
Function
Exports the graph into the file fl in the DIMACS format.

dimacs_export (gr, fl, comment1, ..., commentn)  
Function
Exports the graph into the file fl in the DIMACS format. Optional comments will be added to the top of the file.

dimacs_import (fl)  
Function
Returns the graph from file fl in the DIMACS format.
59.2.5 Visualization

\textbf{draw\_graph} \hspace{0.5cm} \textbf{graph}

Draws the graph using the \texttt{draw} package.

The algorithm used to position vertices is specified by the optional argument \texttt{program}. The default value is \texttt{program=spring\_embedding}. \texttt{draw\_graph} can also use the graphviz programs for positioning vertices, but graphviz must be installed separately.

Example 1:

\begin{verbatim}
(%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)$
\end{verbatim}

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, 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
Chapter 59: graphs

(%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);
   GRAPH
(%o2)
(%i3) draw_graph(g, redraw=true, program=planar_embedding);
   done
(%o3)

Example 6:
(%i1) load(graphs)$
(%i2) t: tutte_graph();
   GRAPH
(%o2)
(%i3) draw_graph(t, redraw=true, fixed_vertices=[1,2,3,4,5,6,7,8,9]);
   done
(%o3)

draw_graph_program
Option variable

The default value for the program used to position vertices in draw_graph program.

draw_graph option

show_id
Default value: false
draw_graph option

If true then ids of the vertices are displayed.

draw_graph option

show_label
Default value: false
draw_graph option

If true then labels of the vertices are displayed.

label_alignment
Default value: center
draw_graph option

Determines how to align the labels/ids of the vertices. Can be left, center or right.

draw_graph option

show_weight
Default value: false
draw_graph option

If true then weights of the edges are displayed.
**vertex_type**
- Default value: circle
- Defines how vertices are displayed. See the `point_type` option for the `draw` package for possible values.

**vertex_size**
- The size of vertices.

**vertex_color**
- The color used for displaying vertices.

**show_vertices**
- Default value: {} 
- Display selected vertices in the using a different color.

**show_vertex_type**
- Defines how vertices specified in `show_vertices` are displayed. See the `point_type` option for the `draw` package for possible values.

**show_vertex_size**
- The size of vertices in `show_vertices`.

**show_vertex_color**
- The color used for displaying vertices in the `show_vertices` list.

**vertex_partition**
- Default value: {} 
- A partition `[[v1,v2,...],...,[vk,...,vn]]` of the vertices of the graph. The vertices of each list in the partition will be drawn in a different color.

**vertex_coloring**
- Specifies coloring of the vertices. The coloring `col` must be specified in the format as returned by `vertex_coloring`.

**edge_color**
- The color used for displaying edges.

**edge_width**
- The width of edges.

**edge_type**
- Defines how edges are displayed. See the `line_type` option for the `draw` package.

**show_edges**
- Display edges specified in the list `e_list` using a different color.
show_edge_color  
The color used for displaying edges in the show_edges list.

draw_graph option

graph option

show_edge_width  
The width of edges in show_edges.

draw_graph option

show_edge_type  
Defines how edges in show_edges are displayed. See the line_type option for the draw package.

draw_graph option

edge_partition  
A partition \([e_1, e_2, \ldots], \ldots, [e_k, \ldots, e_m]\) of edges of the graph. The edges of each list in the partition will be drawn using a different color.

draw_graph option

edge_coloring  
The coloring of edges. The coloring must be specified in the format as returned by the function edge_coloring.

draw_graph option

redraw  
Default value: false

If true, vertex positions are recomputed even if the positions have been saved from a previous drawing of the graph.

draw_graph option

head_angle  
Default value: 15

The angle for the arrows displayed on arcs (in directed graphs).

draw_graph option

head_length  
Default value: 0.1

The length for the arrows displayed on arcs (in directed graphs).

draw_graph option

spring_embedding_depth  
Default value: 50

The number of iterations in the spring embedding graph drawing algorithm.

draw_graph option

terminal  
The terminal used for drawing (see the terminal option in the draw package).

draw_graph option

file_name  
The filename of the drawing if terminal is not screen.

draw_graph option

program  
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  
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)  
Converts a list v_list of vertices to a list of edges of the path defined by v_list.

vertices_to_cycle (v_list)  
Converts a list v_list of vertices to a list of edges of the cycle defined by v_list.
60 grobner

60.1 Introduction to grobner

grobner is a package for working with Groebner bases in Maxima. A tutorial on Groebner Bases can be found at http://www.geocities.com/CapeCanaveral/Hall/3131/

To use the following functions you must load the 'grobner.lisp' package.

load(grobner);

A demo can be started by
demo("grobner.demo");

or

batch("grobner.demo")

Some of the calculation in the demo will take a lot of time therefore the output 'grobner-demo.output' of the demo can be found in the same directory as the demo file.

60.1.1 Notes on the grobner package

The package was written by Marek Rychlik http://alamos.math.arizona.edu and is released 2002-05-24 under the terms of the General Public License(GPL) (see file 'grobner.lisp'. This documentation was extracted from the files 'README', 'grobner.lisp', 'grobner.demo', 'grobner-demo.output'

by Günter Nowak. Suggestions for improvement of the documentation can be discussed at the maxima-mailing-list maxima@math.utexas.edu. The code is a little bit out of date now. Modern implementation use the fast $F_4$ algorithm described in A new efficient algorithm for computing Gröbner bases (F4) Jean-Charles Faugère LIP6/CNRS Université Paris VI January 20, 1999

60.1.2 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
60.2 Functions and Variables for grobner

60.2.1 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.
Chapter 60: grobner

**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.

### 60.2.2 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`.

```maxima
(%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`.

```maxima
(%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`.

```maxima
(%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.

```maxima
(%i1) poly_primitive_part(35*y+21*x, [x, y, z]);
(%o1) 5 y + 3 x
```
**poly_normalize** *(poly, varlist)*  
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.

### 60.2.3 Other functions in grobner

**poly_expand** *(poly, varlist)*  
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.

```plaintext
(%i1) poly_expand((x-y)*(y+x),[x,y]); 2 2
   2
   x - y
(%o1) x - y
(%i2) poly_expand((y+x)^2,[x,y]); 2 2
   2
   y + 2 x y + x
(%o2) y + 2 x y + x
(%i3) poly_expand((y+x)^5,[x,y]); 5 4 2 3 3 2 4 5
   2
   y + 5 x y + 10 x y + 10 x y + 5 x y + x
(%o3) y + 5 x y + 10 x y + 10 x y + 5 x y + x
(%i4) poly_expand(-1-x*exp(y)+x^2/sqrt(y),[x]);
   y
   x
   - x %e + --------- - 1
   sqrt(y)
(%o4) - x %e + --------- - 1
   sqrt(y)
(%i5) poly_expand(-1-sin(x)^2+sin(x),[sin(x)]);
   2
   - sin(x) + sin(x) - 1
(%o5) - sin(x) + sin(x) - 1
```

**poly_expt** *(poly, number, varlist)*  
exponentitates `poly` by a positive integer `number`. If `number` is not a positive integer number an error will be raised.

```plaintext
(%i1) poly_expt(x-y,3,[x,y])-(x-y)^3,expand;
   0
(%o1) 0
```

**poly_content** *(poly, varlist)*  
poly_content extracts the GCD of its coefficients

```plaintext
(%i1) poly_content(35*y+21*x,[x,y]);
   7
(%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 \( c \ast poly \) 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:

\[
c \ast poly = \sum_{i=1}^{n} (a_i \ast polylist_i) + r
\]

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.

drain\_normal\_form (poly, polylist, varlist) Function
\( drain\_normal\_form \) finds the normal form of a polynomial \( poly \) with respect to a set of polynomials \( polylist \).

poly\_buchberger\_criterion (polylist, varlist) Function
Returns \texttt{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.

60.2.4 Standard postprocessing of Groebner Bases

The \( k \)-th elimination \textit{ideal} \( I_k \) of an ideal \( I \) over \( K[x_1, ..., x_1] \) is \( I \cap K[x_{k+1}, ..., x_n] \).
The \textit{colon ideal} \( I : J \) is the ideal \( \{ h \mid \forall w \in J : wh \in I \} \).
The ideal \( I : p^\infty \) is the ideal \( \{ h \mid \exists n \in N : p^n h \in I \} \).
The ideal \( I : J^\infty \) is the ideal \( \{ h \mid \exists n \in N, \exists p \in J : p^n h \in I \} \).
The \textit{radical ideal} \( \sqrt[I]{} \) is the ideal \( \{ h \mid \exists n \in N : h^n \in I \} \).

poly\_reduction (polylist, varlist) Function
\( poly\_reduction \) reduces a list of polynomials \( polylist \), so that each polynomial is fully reduced with respect to the other polynomials.

poly\_minimization (polylist, varlist) Function
Returns a sublist of the polynomial list \( polylist \) spanning the same monomial ideal as \( polylist \) but minimal, i.e. no leading monomial of a polynomial in the sublist divides the leading monomial of another polynomial.
Function  
\texttt{poly\_normalize\_list} (\texttt{polylist}, \texttt{varlist})
\texttt{poly\_normalize\_list} applies \texttt{poly\_normalize} to each polynomial in the list. That means it divides every polynomial in a list \texttt{polylist} by its leading coefficient.

Function  
\texttt{poly\_grobner} (\texttt{polylist}, \texttt{varlist})
Returns a Groebner basis of the ideal span by the polynomials \texttt{polylist}. Affected by the global flags.

Function  
\texttt{poly\_reduced\_grobner} (\texttt{polylist}, \texttt{varlist})
Returns a reduced Groebner basis of the ideal span by the polynomials \texttt{polylist}. Affected by the global flags.

Function  
\texttt{poly\_depends\_p} (\texttt{poly}, \texttt{var}, \texttt{varlist})
\texttt{poly\_depends} tests whether a polynomial depends on a variable \texttt{var}.

Function  
\texttt{poly\_elimination\_ideal} (\texttt{polylist}, \texttt{number}, \texttt{varlist})
\texttt{poly\_elimination\_ideal} returns the grobner basis of the \texttt{number}-th elimination ideal of an ideal specified as a list of generating polynomials (not necessarily Groebner basis).

Function  
\texttt{poly\_colon\_ideal} (\texttt{polylist1}, \texttt{polylist2}, \texttt{varlist})
Returns the reduced Groebner basis of the colon ideal \(I(\texttt{polylist1}): I(\texttt{polylist2})\) where \texttt{polylist1} and \texttt{polylist2} are two lists of polynomials.

Function  
\texttt{poly\_ideal\_intersection} (\texttt{polylist1}, \texttt{polylist2}, \texttt{varlist})
\texttt{poly\_ideal\_intersection} returns the intersection of two ideals.

Function  
\texttt{poly\_lcm} (\texttt{poly1}, \texttt{poly2}, \texttt{varlist})
Returns the lowest common multiple of \texttt{poly1} and \texttt{poly2}.

Function  
\texttt{poly\_gcd} (\texttt{poly1}, \texttt{poly2}, \texttt{varlist})
Returns the greatest common divisor of \texttt{poly1} and \texttt{poly2}.

Function  
\texttt{poly\_groebner\_equal} (\texttt{polylist1}, \texttt{polylist2}, \texttt{varlist})
\texttt{poly\_groebner\_equal} tests whether two Groebner Bases generate the same ideal. Returns \texttt{true} if two lists of polynomials \texttt{polylist1} and \texttt{polylist2}, assumed to be Groebner Bases, generate the same ideal, and \texttt{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 \texttt{false}.

\begin{verbatim}
(%i1) poly_grobner_equal([y+x,x-y],[x,y],[x,y]);
(%o1) false
\end{verbatim}
**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

\[ I(\text{polylist}) : poly^\infty \]

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

\[ I(\text{polylist}_1) : I(\text{polylist}_2)^\infty \]

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 is a list of n polynomials [poly1,...,polyn]. Returns the reduced Groebner basis of the ideal

\[ I(\text{polylist}) : poly_1^\infty : ... : poly_n^\infty \]

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

\[ I(\text{polylist}) : I(\text{polylist}_1)^\infty : ... : I(\text{polylist}_n)^\infty \]

**poly_saturation_extension** (poly, polylist, varlist1, varlist2)  
Function

poly_saturation_extension implements the famous Rabinowitz trick.

**poly_polysaturation_extension** (poly, polylist, varlist1, varlist2)  
Function
61 impdiff

61.1 Functions and Variables for impdiff

**implicit_derivative** \((f, \text{indvarlist}, \text{orderlist}, \text{depvar})\)

This subroutine computes implicit derivatives of multivariable functions. \(f\) is an array function, the indexes are the derivative degree in the \text{indvarlist} order; \text{indvarlist} is the independent variable list; \text{orderlist} is the order desired; and \text{depvar} is the dependent variable.

To use this function write first `load("impdiff").`
62 implicit_plot

62.1 Functions and Variables for implicit_plot

**implicit_plot** (\(expr, x\_range, y\_range\))

**Function**

Displays a plot of one or more expressions in implicit form. \(expr\) is the expression to be plotted, \(x\_range\) the range of the horizontal axis and \(y\_range\) the range of vertical axis. implicit_plot respects global setting for the gnuplot driver set by the \(set\_plot\_option\) function. Options can also be passed to implicit_plot function as optional arguments.

implicit_plot works by tracking sign changes on the area given by \(x\_range\) and \(y\_range\) and can fail for complicated expressions.

load(implicit_plot) loads this function.

Example:

(%i1) implicit_plot (x^2 = y^3 - 3*y + 1, [x, -4, 4], [y, -4, 4], [gnuplot_preamble, "set zeroaxis"]);
63 interpol

63.1 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'.

63.2 Functions and Variables for interpol

Function lagrange (points)
Function 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.

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)   35
   -------------------------------
   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) := \frac{35}{(x - 8) (x - 6) (x - 3) (x - 1)}
- \frac{12}{(x - 8) (x - 7) (x - 3) (x - 1)}
+ \frac{30}{(x - 8) (x - 7) (x - 6) (x - 1)}
- \frac{60}{(x - 8) (x - 7) (x - 6) (x - 3)}
+ \frac{84}{(x - 8) (x - 7) (x - 6) (x - 1)}

\%(i5) /* Evaluate the polynomial at some points */
expand(map(f,[2.3,5/7,%pi]));
\%(o5) \[- 1.567535, \frac{919062}{84035}, \frac{73}{420} %\pi, \frac{701}{210} %\pi, \frac{8957}{420} %\pi\]
\%(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);
\%(o9) \frac{35}{(w - 8) (w - 6) (w - 3) (w - 1)}
- \frac{12}{(w - 8) (w - 7) (w - 3) (w - 1)}
+ \frac{30}{(w - 8) (w - 7) (w - 6) (w - 1)}
\[(w - 8) (w - 7) (w - 6) (w - 3)\]

\[+ \frac{60}{84}\]

\[\text{charfun2}(x, a, b)\]

Returns \texttt{true} if number \(x\) belongs to the interval \([a, b)\), and \texttt{false} otherwise.

\texttt{linearinterpol}(points)

\texttt{linearinterpol}(points, option)

Computes the polynomial interpolation by the linear method. Argument \texttt{points} must be either:

- a two column matrix, \(p:\text{matrix}([2,4],[5,6],[9,3])\),
- 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 \texttt{option} argument it is possible to select the name for the independent variable, which is \texttt{'x} by default; to define another one, write something like \texttt{varname='z}.

Examples:

\begin{verbatim}
(%i1) load(interpol)$
(%i2) p: matrix([7,2],[8,3],[1,5],[3,2],[6,7])$
(%i3) linearinterpol(p);
    13  3 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

(%i4) f(x):=''%;
    13  3 x
   --- - --- charfun2(x, minf, 3)
    2  2
+ (x - 5) charfun2(x, 7, inf) + (37 - 5 x) charfun2(x, 6, 7)
   5 x
+ (--- - 3) charfun2(x, 3, 6)
   3
(%i5) /* Evaluate the polynomial at some points */
    map(f,[7.3,25/7,%pi]);
    62 5 %pi
(%o5) [2.3, --, ----- - 3]
    21 3
\end{verbatim}
(%i6) %, numer;
(%o6) [2.3, 2.952380952380953, 2.235987755982989]
(%i7) load(draw)$ /* load draw package */
(%i8) /* Plot the polynomial together with points */
draw2d(
  color = red,  
  key = "Linear interpolator",  
  explicit(f(x),x,-5,20),  
  point_size = 3,  
  color = blue,  
  key = "Sample points",  
  points(args(p)))$
(%i9) /* Change variable name */
linearinterpol(p, varname='s);
13 3 s
(%o9) (-- - ---) charfun2(s, minf, 3)
 2 2
+ (s - 5) charfun2(s, 7, inf) + (37 - 5 s) charfun2(s, 6, 7)
 5 s
+ (--- - 3) charfun2(s, 3, 6)
 3

cspline (points)                          Function
cspline (points, option1, option2, ...)  Function

Computes the polynomial interpolation by the cubic splines method. Argument points must be either:

- a two column matrix, p:matrix([[2,4],[5,6],[9,3]],
- a list of pairs, p: [[2,4],[5,6],[9,3]],
- a list of numbers, p: [4,6,3], in which case the abscissas will be assigned automatically to 1, 2, 3, etc.

In the first two cases the pairs are ordered with respect to the first coordinate before making computations.

There are three options to fit specific needs:

- 'd1, default 'unknown, is the first derivative at x_1; if it is 'unknown, the second derivative at x_1 is made equal to 0 (natural cubic spline); if it is equal to a number, the second derivative is calculated based on this number.
- 'dn, default 'unknown, is the first derivative at x_n; if it is 'unknown, the second derivative at x_n is made equal to 0 (natural cubic spline); if it is equal to a number, the second derivative is calculated based on this number.
- 'varname, default 'x, is the name of the independent variable.

Examples:
(%i11) load(interpol)$
(%i12) p: [[7,2],[8,2],[1,5],[3,2],[6,7]]$
(%i13) /* 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 */
 3 2
1949 x  11437 x  17027 x  1247
(%o8) (------- - -------- + ------- + ----) charfun2(x, minf, 3)
       2256      2256      2256       752
3 2
1547 x  35681 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)
Constructing new interpolating functions:

```maxima
ratsimp(derivative(f(x),x)
```

Plotting both functions together:

```maxima
draw2d(
  color = black,
  key = "Cubic splines (default)",
  implicit(f(x),x,0,10),
  color = red,
  key = "Cubic splines (d1=0,dn=0)",
  implicit(g(x),x,0,10),
  point_size = 3,
  color = blue,
  key = "Sample points",
  points(p))$
```

### `ratinterpol` Function

**`ratinterpol(points, numdeg)`**

**`ratinterpol(points, numdeg, option1, option2, ...)`**

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 are two options to fit specific needs:

- `'denterm`, default 1, is the independent term of the polynomial in the denominator.
- `'varname`, default `'x`, is the name of the independent variable.

Examples:

```maxima
(%i11) load(interpol)$
(%i12) load(draw)$
(%i13) p: [[7.2,2.5],[8.5,2.1],[1.6,5.1],[3.4,2.4],[6.7,7.9]]$
(%i14) 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])$
```
64 lapack

64.1 Introduction to lapack

lapack is a Common Lisp translation (via the program f2c) of the Fortran library LAPACK, as obtained from the SLATEC project.

64.2 Functions and Variables for lapack

**Function**

dgeev \((A)\)
dgeev \((A, \text{right}_p, \text{left}_p)\)

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 \((A, \text{right}_p, \text{left}_p)\) computes the eigenvalues of \(A\) and the right eigenvectors when \(\text{right}_p = \text{true}\) and the left eigenvectors when \(\text{left}_p = \text{true}\).

A list of three items is returned. The first item is a list of the eigenvalues. The second item is false or the matrix of right eigenvectors. The third item is false or the matrix of left eigenvectors.

The right eigenvector \(v(j)\) (the \(j\)-th column of the right eigenvector matrix) satisfies \(A.v(j) = \lambda(j).v(j)\)

where \(\lambda(j)\) is the corresponding eigenvalue. The left eigenvector \(u(j)\) (the \(j\)-th column of the left eigenvector matrix) satisfies

\[ u(j) \star\star H . A = \lambda(j).u(j) \star\star H \]

where \(u(j) \star\star H\) 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:

```lisp
(%i11) load (lapack)$
(%i12) fpprintprec : 6; 6
(%i13) M : matrix ([9.5, 1.75], [3.25, 10.45]);
   [ 9.5  1.75 ]
   [           ]
   [ 3.25 10.45 ]
(%i14) dgeev (M);
(%i15) [L, v, u] : dgeev (M, true, true);
```

```lisp
[[7.54331, 12.4067], [7.54331, 12.4067], [ - .666642 - .515792 ], [ - .666642 - .515792 ]
```

```lisp
[.745378 - .856714 ],
```
820       Maxima 5.24.0 Manual

(\%i6) D : apply (diag_matrix, L);
[ 7.54331  0 ]
[ 0     12.4067]

(\%o6)

(\%i7) M . v - v . D;
[ 0.0   - 8.88178E-16 ]
[ - 8.88178E-16 0.0 ]

(\%o7)

(\%i8) transpose (u) . M - D . transpose (u);
[ 0.0   - 4.44089E-16 ]
[ 0.0 0.0]

\textbf{dgesv} \ (A, b)

\textit{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 \texttt{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.

\texttt{dgesv} computes the solution via the LU decomposition of \(A\).

\textbf{Examples:}

\texttt{dgesv} computes the solution of the linear equation \(Ax = b\).

\begin{verbatim}
(\%i1) A : matrix ([1, -2.5], [0.375, 5]);
[ 1 - 2.5 ]
[ 0.375 5 ]

(\%o1)

(\%i2) b : matrix ([1.75], [-0.625]);
[ 1.75 ]
[ - 0.625 ]

(\%o2)

(\%i3) x : dgesv (A, b);
[ 1.210526315789474 ]
[ - 0.215789473684211 ]

(\%o3)

(\%i4) dlange (inf_norm, b - A.x);
[ 0.0 ]
(\%o4)

\end{verbatim}

\(b\) is a matrix with the same number of rows as \(A\) and any number of columns. \(x\) is the same size as \(b\).

\begin{verbatim}
(\%o0)

(\%i1) A : matrix ([1, -0.15], [1.82, 2]);
[ 1 - 0.15 ]
[ ]

(\%o1)

\end{verbatim}
Chapter 64: lapack

(%i2) b : matrix ([3.7, 1, 8], [-2.3, 5, -3.9]);
(%o2) 

(%i3) x : dgesv (A, b);
(%o3) 

(%i4) dlange (inf_norm, b - A . x);
(%o4) 

The elements of A and b must evaluate to real floating point numbers.

(%i1) A : matrix ([5, -%pi], [1b0, 11/17]);
(%o1) 

(%i2) b : matrix ([%e], [sin(1)]);
(%o2) 

(%i3) x : dgesv (A, b);
(%o3) 

(%i4) dlange (inf_norm, b - A . x);
(%o4) 

**dgesvd** (A)  Function
**dgesvd** (A, left_p, right_p)  Function

Computes the singular value decomposition (SVD) of a matrix A, comprising the singular values and, optionally, the left and right singular vectors. All elements of A must be integer or floating point numbers. A might or might not be square (same number of rows and columns).

Let m be the number of rows, and n the number of columns of A. The singular value decomposition of A comprises three matrices, U, Sigma, and V^T, such that

\[ A = U.Sigma.V^T \]

where U is an m-by-m unitary matrix, Sigma is an m-by-n diagonal matrix, and V^T is an n-by-n unitary matrix.

Let sigma[i] be a diagonal element of Sigma, that is, Sigma[i, i] = sigma[i]. The elements sigma[i] are the so-called singular values of A; these are real and nonnegative, and returned in descending order. The first min(m, n) columns of U and V are the left and right singular vectors of A. Note that dgesvd returns the transpose of V, not V itself.
**dgesvd**(*A*) computes only the singular values of *A*. **dgesvd**(*A*, *left_p*, *right_p*) computes the singular values of *A* and the left singular vectors when *left_p* = true and the right singular vectors when *right_p* = true.

A list of three items is returned. The first item is a list of the singular values. The second item is false or the matrix of left singular vectors. The third item is false or the matrix of right singular vectors.

Example:

```
(%i1) load (lapack)$
(%i2) fpprintprec : 6;$
(%i3) M: matrix([1, 2, 3], [3.5, 0.5, 8], [-1, 2, -3], [4, 9, 7]);
  [ 1  2   3   ]
  [             ]
  [ 3.5  0.5  8  ]
  [             ]
  [ -1   2   -3 ]
  [             ]
  [ 4    9  7  ]
(%o3)                  [  ]
(%o4) dgesvd (M);
(%o5) dgesvd (M, true, true);
(%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  ]
[               ]
[ 0     0  .452547 ]
[               ]
[ 0     0  0     ]
```
Chapter 64: lapack

\[(\%i9)\] \( U \cdot \Sigma \cdot VT - M; \)
\[
\begin{bmatrix}
1.11022E-15 & 0.0 & 1.77636E-15 \\
1.33227E-15 & 1.66533E-15 & 0.0 \\
-4.44089E-16 & 8.88178E-16 & 4.44089E-16 \\
8.88178E-16 & 1.77636E-15 & 8.88178E-16 \\
\end{bmatrix}
\]

\[(\%o9)\]

\[(\%i10)\] \( \text{transpose}(U) \cdot U; \)
\[
\begin{bmatrix}
1.0 & 5.55112E-17 & 2.498E-16 & 2.77556E-17 \\
5.55112E-17 & 1.0 & 5.55112E-17 & 4.16334E-17 \\
2.498E-16 & 5.55112E-17 & 1.0 & -2.08167E-16 \\
2.77556E-17 & 4.16334E-17 & -2.08167E-16 & 1.0 \\
\end{bmatrix}
\]

\[(\%o10)\]

\[(\%i11)\] \( VT \cdot \text{transpose}(VT); \)
\[
\begin{bmatrix}
1.0 & 0.0 & -5.55112E-17 \\
0.0 & 1.0 & 5.55112E-17 \\
-5.55112E-17 & 5.55112E-17 & 1.0 \\
\end{bmatrix}
\]

\[(\%o11)\]

**dlange** (norm, A)  
**zrange** (norm, A)

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** (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, \( \text{dgemm}(A, B) \) computes the product of the two real matrices, A and B.

In the second form, \( \text{dgemm} \) computes the \( \alpha \cdot A \cdot B + \beta \cdot C \) 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`.

```maxima
(%i1) load (lapack)$
(%i2) A : matrix([1,2,3],[4,5,6],[7,8,9]);
     [ 1  2  3 ]
     [        ]
     [ 4  5  6 ]
     [        ]
     [ 7  8  9 ]
(%o2) [ ]
(%i3) B : matrix([-1,-2,-3],[-4,-5,-6],[-7,-8,-9]);
     [ - 1  - 2  - 3 ]
     [                ]
     [ - 4  - 5  - 6 ]
     [                ]
     [ - 7  - 8  - 9 ]
(%o3) [ ]
(%i4) C : matrix([3,2,1],[6,5,4],[9,8,7]);
     [ 3  2  1 ]
     [        ]
     [ 6  5  4 ]
     [        ]
     [ 9  8  7 ]
(%o4) [ ]
(%i5) dgemm(A,B);
     [ - 30.0  - 36.0  - 42.0 ]
     [                      ]
     [ - 66.0  - 81.0  - 96.0 ]
     [                      ]
     [ - 102.0  - 126.0  - 150.0 ]
(%o5) [ ]
(%i6) A . B;
     [ - 30  - 36  - 42 ]
     [                  ]
     [ - 66  - 81  - 96 ]
     [                  ]
     [ - 102  - 126  - 150 ]
(%o6) [ ]
```
(%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 ]
65 lbfgs

65.1 Introduction to lbfgs

lbfgs is an implementation of the L-BFGS algorithm [1] to solve unconstrained minimization problems via a limited-memory quasi-Newton (BFGS) algorithm. It is called a limited-memory method because a low-rank approximation of the Hessian matrix inverse is stored instead of the entire Hessian inverse. The program was originally written in Fortran [2] by Jorge Nocedal, incorporating some functions originally written by Jorge J. Moré and David J. Thuente, and translated into Lisp automatically via the program f2cl. The Maxima package lbfgs comprises the translated code plus an interface function which manages some details.

References:

65.2 Functions and Variables for lbfgs

lbfgs (FOM, X, X0, epsilon, iprint) Function
lbfgs ([FOM, grad] X, X0, epsilon, iprint) Function

Finds an approximate solution of the unconstrained minimization of the figure of merit FOM over the list of variables X, starting from initial estimates X0, such that \( \| \text{grad}(\text{FOM}) \| < \epsilon \times \max(1, \| X \|) \).

\text{grad}, if present, is the gradient of FOM with respect to the variables X. \text{grad} is a list, with one element for each element of X. If not present, the gradient is computed automatically by symbolic differentiation.

The algorithm applied is a limited-memory quasi-Newton (BFGS) algorithm [1]. It is called a limited-memory method because a low-rank approximation of the Hessian matrix inverse is stored instead of the entire Hessian inverse. Each iteration of the algorithm is a line search, that is, a search along a ray in the variables X, with the search direction computed from the approximate Hessian inverse. The FOM is always decreased by a successful line search. Usually (but not always) the norm of the gradient of FOM also decreases.

\text{iprint} controls progress messages printed by lbfgs.

\text{iprint[1]} controls progress messages.
\text{iprint[1]} < 0 No progress messages.
\text{iprint[1]} = 0 Messages at the first and last iterations.
i\text{print}[1] > 0
   \text{Print a message every } i\text{print}[1] \text{ iterations.}

\text{iprint}[2]
\text{iprint}[2] \text{ controls the verbosity of progress messages.}
\text{iprint}[2] = 0
   \text{Print out iteration count, number of evaluations of FOM, value of FOM, norm of the gradient of FOM, and step length.}
\text{iprint}[2] = 1
   \text{Same as } \text{iprint}[2] = 0, \text{ plus } X0 \text{ and the gradient of FOM evaluated at } X0.
\text{iprint}[2] = 2
   \text{Same as } \text{iprint}[2] = 1, \text{ plus values of } X \text{ at each iteration.}
\text{iprint}[2] = 3
   \text{Same as } \text{iprint}[2] = 2, \text{ plus the gradient of FOM at each iteration.}

The columns printed by \text{lbfgs} are the following.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Number of iterations. It is incremented for each line search.</td>
</tr>
<tr>
<td>NFN</td>
<td>Number of evaluations of the figure of merit.</td>
</tr>
<tr>
<td>FUNC</td>
<td>Value of the figure of merit at the end of the most recent line search.</td>
</tr>
<tr>
<td>GNORM</td>
<td>Norm of the gradient of the figure of merit at the end of the most recent line search.</td>
</tr>
</tbody>
</table>

\text{STEPLENGTH}
   An internal parameter of the search algorithm.

Additional information concerning details of the algorithm are found in the comments of the original Fortran code [2].

See also \text{lbfgs_nfeval_max} and \text{lbfgs_ncorrections}.

References:

Examples:
The same FOM as computed by FGCOMPUTE in the program sdrive.f in the LBFGS package from Netlib. Note that the variables in question are subscripted variables. The FOM has an exact minimum equal to zero at $u[k] = 1$ for $k = 1, \ldots, 8$.

\begin{verbatim}
(\%i1) load (lbfgs);
(\%o1) /usr/share/maxima/5.10.0cvs/share/lbfgs/lbfgs.mac
(\%i2) t1[j] := 1 - u[j];
(\%o2) t1 := 1 - u
      \hfill j
(\%i3) t2[j] := 10*(u[j + 1] - u[j]^2);
\end{verbatim}
Chapter 65: lbfgs

\[
\begin{align*}
\text{(\%o3)} & \quad t2 := 10 (u - u_j) \\
\text{(\%i4)} & \quad n : 8; \\
\text{(\%o4)} & \quad 8 \\
\text{(\%i5)} & \quad \text{FOM : sum (t1[2*j - 1]^2 + t2[2*j - 1]^2, j, 1, n/2);} \\
\text{(\%o5)} & \quad 100 (u - u_j) + (1 - u_j) + 100 (u - u_6) + (1 - u_6) \\
& \quad + 100 (u - u_4) + (1 - u_4) + 100 (u - u_3) + (1 - u_3). \\
\text{(\%i6)} & \quad \text{lbfgs (FOM, '\{u[1],u[2],u[3],u[4],u[5],u[6],u[7],u[8]\}',} \\
& \quad \{-1.2, 1, -1.2, 1, -1.2, 1, -1.2, 1, 1e-3, [1, 0]\}; \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>I</th>
<th>NFN</th>
<th>FUNC</th>
<th>GNorm</th>
<th>STEPLENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1.651479526340304D+01</td>
<td>4.3243592913357977D+00</td>
<td>7.926153934960631D-04</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1.650209316638371D+01</td>
<td>3.57588161060007D+00</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1.645461701328371D+01</td>
<td>6.230869936015777D+00</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>1.636867301273588D+01</td>
<td>1.177589920734880D+01</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>1.612153014409201D+01</td>
<td>2.292797147151280D+01</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>1.569118407906628D+01</td>
<td>3.687447158775517D+01</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>1.510361968985424D+01</td>
<td>4.501931728123680D+01</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>1.391077875774294D+01</td>
<td>4.526061463810632D+01</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>1.165625686278198D+01</td>
<td>2.748348965356917D+01</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>9.859422687859137D+00</td>
<td>2.11149947231644D+01</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>7.815442521732281D+00</td>
<td>6.110762235766556D+00</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>12</td>
<td>15</td>
<td>7.346380905773160D+00</td>
<td>2.165281166714631D+01</td>
<td>1.285316401779533D-01</td>
</tr>
<tr>
<td>13</td>
<td>16</td>
<td>6.330460634638371D+00</td>
<td>1.401220851762050D+01</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>14</td>
<td>17</td>
<td>5.238763983914394D+00</td>
<td>1.702473787613255D+01</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>15</td>
<td>18</td>
<td>3.750416790406701D+00</td>
<td>7.981845727704576D+00</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
<td>3.001238402309352D+00</td>
<td>3.925482944716691D+00</td>
<td>2.333129631296807D-01</td>
</tr>
<tr>
<td>17</td>
<td>22</td>
<td>2.794390709712890D+00</td>
<td>8.24332982546473D+00</td>
<td>2.50357728378232D+01</td>
</tr>
<tr>
<td>18</td>
<td>23</td>
<td>2.563783562918759D+00</td>
<td>1.035413426521790D+01</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>19</td>
<td>24</td>
<td>2.019429976377856D+00</td>
<td>1.056187312346769D+01</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>20</td>
<td>25</td>
<td>1.420003187670903D+00</td>
<td>2.475962450826961D+00</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>21</td>
<td>27</td>
<td>1.197874268461340D+00</td>
<td>8.44107983493810D+00</td>
<td>4.303451060808756D-01</td>
</tr>
<tr>
<td>22</td>
<td>28</td>
<td>9.023848941942773D-01</td>
<td>1.113189216635162D+01</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>23</td>
<td>29</td>
<td>5.508226405867377D-01</td>
<td>2.380830600326308D+00</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>24</td>
<td>31</td>
<td>3.902893258815567D-01</td>
<td>5.625595816584421D+00</td>
<td>4.834988416524465D-01</td>
</tr>
<tr>
<td>25</td>
<td>32</td>
<td>3.207542206990315D-01</td>
<td>1.14944645416472D+01</td>
<td>1.0000000000000000D+00</td>
</tr>
<tr>
<td>26</td>
<td>33</td>
<td>1.874468266362791D-01</td>
<td>3.63248215288097D+00</td>
<td>1.0000000000000000D+00</td>
</tr>
</tbody>
</table>
THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
IFLAG = 0
(%o6) [u = 1.000005339815974, u = 1.000009942839805,
1 2
u = 1.000005339815974, u = 1.000009942839805,
3 4
u = 1.000005339815974, u = 1.000009942839805,
5 6
u = 1.000005339815974, u = 1.000009942839805]
7 8

A regression problem. The FOM is the mean square difference between the predicted value \( F(X[i]) \) and the observed value \( Y[i] \). The function \( F \) is a bounded monotone function (a so-called "sigmoidal" function). In this example, \texttt{lbfgs} computes approximate values for the parameters of \( F \) and \texttt{plot2d} displays a comparison of \( F \) with the observed data.

(%i1) load (lbfgs);
(%o1) /usr/share/maxima/5.10.0cvs/share/lbfgs/lbfgs.mac
(%i2) FOM : '((1/length(X))*sum((F(X[i]) - Y[i])^2, i, 1, length(X)));
(%o2) ((F(X[i]) - Y[i])^2, i, 1, 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)));
(%o5) F(x) := A/(1 + exp(-B*(x - C)))
(%i6) ''FOM;
(%o6) (((------------------- - 1.5) + (------------------- - 1.25)

\%e + 1 \%e + 1

- B (5 - C) - B (4 - C)

\%e + 1 \%e + 1

\%e + 1 \%e + 1

\%e + 1 \%e + 1
Chapter 65: lbfgs

\[
\frac{A}{2} + \left( \frac{2}{\text{e} + 1} \right) + \left( \frac{2}{\text{e} + 1} \right)
\]

\[
\begin{align*}
&- B (3 - C) \quad \%e + 1 \\
&- B (2 - C) \quad \%e + 1 \\
&\frac{A}{2} \\
&+ \left( \frac{\text{e} - 1}{\text{e} - 1} \right) / 5 \\
&- B (1 - C) \quad 2 \\
&\left( \%e + 1 \right)
\end{align*}
\]

(%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= 2.00021531936760D-01
*************************************************

<table>
<thead>
<tr>
<th>I</th>
<th>NFN</th>
<th>FUNC</th>
<th>GNORM</th>
<th>STEPLENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1.177820636622582D-01</td>
<td>9.893183394539922D-02</td>
<td>8.55435968992371D-01</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2.302653892214013D-02</td>
<td>1.180098521565904D-01</td>
<td>2.100000000000000D+01</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>1.496348495303005D-02</td>
<td>9.611201567691633D-02</td>
<td>5.257340567840707D-01</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>7.90460841091393D-03</td>
<td>1.325041647391314D-02</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>7.314495451266917D-03</td>
<td>1.510670810312237D-02</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>6.750147275936680D-03</td>
<td>1.91496458023047D-02</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>5.850716021108205D-03</td>
<td>1.028089194579363D-02</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td>5.776864230657791D-03</td>
<td>3.676866074530332D-04</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td>5.777818823650782D-03</td>
<td>3.010740179797255D-04</td>
<td>1.000000000000000D+00</td>
</tr>
</tbody>
</table>

THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
IFLAG = 0
(%o7) [A = 1.461933911464101, B = 1.601593973254802,
C = 2.528933072164854]
(%i8) plot2d ([F(x), [discrete, X, Y]], [x, -1, 6]), 'estimates;

Gradient of FOM is specified (instead of computing it automatically).
(%i1) load (lbfgs)$
(%i2) F(a, b, c) := (a - 5)^2 + (b - 3)^4 + (c - 2)^6;
2 4 6
(%o2) F(a, b, c) := (a - 5) + (b - 3) + (c - 2)
(%i3) F_grad : map (lambda ([x], diff (F(a, b, c), x)), [a, b, c]);
3 5
(%o3) \{2 (a - 5), 4 (b - 3), 6 (c - 2) \}
(%i4) estimates : lbfgs ([F(a, b, c), F_grad], [a, b, c], [0, 0, 0], 1e-4, [1, 0],

*************************************************
N= 3 NUMBER OF CORRECTIONS=25
INITIAL VALUES
F= 1.700000000000000D+02 GNORM= 2.205175729958953D+02
*************************************************
<table>
<thead>
<tr>
<th>I</th>
<th>NFN</th>
<th>FUNC</th>
<th>GNORM</th>
<th>STEPLENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>6.632967565917638D+01</td>
<td>6.49841132518770D+01</td>
<td>4.534785987412505D-03</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4.36890936228036D+01</td>
<td>3.784147651974131D+01</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2.685298972775190D+01</td>
<td>1.640262125898521D+01</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1.909064767659852D+01</td>
<td>9.733664001790506D+00</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>1.006493272061515D+01</td>
<td>6.344808151880209D+00</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>1.215263596054294D+00</td>
<td>2.204727876126879D+00</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>1.080252896385334D-02</td>
<td>1.431637116951849D-01</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>8.407195124830908D-03</td>
<td>1.126344579730013D-03</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>5.022091686198527D-03</td>
<td>7.750731829225274D-02</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>2.277152808939775D-03</td>
<td>5.032810859286795D-02</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>6.489384688303218D-04</td>
<td>1.932007150271008D-02</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>2.075791943844548D-04</td>
<td>6.964319310814364D-03</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>7.349472666162257D-05</td>
<td>4.017449067849554D-03</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>2.293617477985237D-05</td>
<td>1.334590390856715D-03</td>
<td>1.000000000000000D+00</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>7.683645404048675D-06</td>
<td>6.011057038099201D-04</td>
<td>1.000000000000000D+00</td>
</tr>
</tbody>
</table>

THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
IFLAG = 0
(%o4) [a = 5.000086823042934, b = 3.05239542970518, c = 1.927980629919583]

lbfgs_nfeval_max

Variable
Default value: 100

lbfgs_nfeval_max is the maximum number of evaluations of the figure of merit (FOM) in lbfgs. When lbfgs_nfeval_max is reached, lbfgs returns the result of the last successful line search.

lbfgs_ncorrections

Variable
Default value: 25

lbfgs_ncorrections is the number of corrections applied to the approximate inverse Hessian matrix which is maintained by lbfgs.
66 lindstedt

66.1 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

2
7 e e

T = (- ---- - -- + 1) t]

3072 16
```

To use this function write first load("makeOrders") and load("lindstedt").
Chapter 67: linearalgebra

67 linearalgebra

67.1 Introduction to linearalgebra

linearalgebra is a collection of functions for linear algebra.

Example:

(%i1) M : matrix ([[1, 2], [1, 2]]);

(%o1) [[1, 2], [1, 2]]

(%i2) nullspace (M);

(%o2) span([1])

(%i3) columnspace (M);

(%i4) ptriangularize (M - z*ident(2), z);

(%i5) M : matrix ([[1, 2, 3], [4, 5, 6], [7, 8, 9]] - z*ident(3));

(%i6) MM : ptriangularize (M, z);

(%i7) algebraic : true;

(%i8) tellrat (MM [3, 3]);
(\%i9) MM : ratsimp (MM);
\[
\begin{bmatrix}
4 & 5 - z & 6 \\
2 \\
66 & 7 z - 102 z - 132 \\
49 & 49 \\
0 & 0
\end{bmatrix}
\]
\[
\begin{aligned}
\text{(\%o9)} \quad & \frac{66}{49} \\
& \frac{7 z - 102 z - 132}{49} \\
& \frac{49}{49} \\
& \frac{0}{49}
\end{aligned}
\]
(\%i10) nullspace (MM);
\[
\begin{bmatrix}
1 \\
2 \\
z - 14 \\
--
\end{bmatrix}
\]
\[
\text{(\%o10)} \quad \text{span}\left(\begin{bmatrix}
8 \\
2 \\
z - 18 \\
--
\end{bmatrix}\right)
\]
(\%i11) M : matrix ([1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]);
\[
\begin{bmatrix}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
13 & 14 & 15 & 16
\end{bmatrix}
\]
(\%o11)
(\%i12) columnspace (M);
\[
\begin{bmatrix}
1 \\
2 \\
z - 18 \\
--
\end{bmatrix}
\]
\[
\text{(\%o12)} \quad \text{span}\left(\begin{bmatrix}
0 \\
0 \\
1 \\
3
\end{bmatrix}\right)
\]
(\%i13) apply ('orthogonal_complement, args (nullspace (transpose (M))));
\[
\begin{bmatrix}
0 \\
1 \\
1 \\
0
\end{bmatrix}
\]
\[
\text{(\%o13)} \quad \text{span}\left(\begin{bmatrix}
2 \\
- 1 \\
0 \\
- 2
\end{bmatrix}\right)
\]

67.2 Functions and Variables for linearalgebra
addmatrices \((f, M_1, ..., M_n)\) \hspace{1cm} \text{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) \quad m1 : \text{matrix}([[1,2],[3,4]])

(\%i2) \quad m2 : \text{matrix}([[7,8],[9,10]])

(\%i3) \quad \text{addmatrices('max,m1,m2)};

(\%o3) \quad \text{matrix}([[7,8],[9,10]])

(\%i4) \quad \text{addmatrices('max,m1,m2,5*m1)};

(\%o4) \quad \text{matrix}([[7,10],[15,20]])
\]

blockmatrixp \((M)\) \hspace{1cm} \text{Function}

Return true if and only if \(M\) is a matrix and every entry of \(M\) is a matrix.

columnop \((M, i, j, \theta)\) \hspace{1cm} \text{Function}

If \(M\) is a matrix, return the matrix that results from doing the column operation \(C_i \leftarrow C_i - \theta \cdot C_j\). If \(M\) doesn’t have a row \(i\) or \(j\), signal an error.

columnswap \((M, i, j)\) \hspace{1cm} \text{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 \((M)\) \hspace{1cm} \text{Function}

If \(M\) is a matrix, return \(\text{span} (v_1, ..., v_n)\), where the set \(\{v_1, ..., v_n\}\) is a basis for the column space of \(M\). The span of the empty set is \(\{0\}\). Thus, when the column space has only one member, return \(\text{span} ()\).

copy \((e)\) \hspace{1cm} \text{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) \quad m : [1,[2,3]]$

(\%i2) \quad mm : m$

(\%i3) \quad \text{mm[2][1]} : x$

(\%i4) \quad m;

(\%o4) \quad \text{[1,[x,3]]}$

(\%i5) \quad mm;

(\%o5) \quad \text{[1,[x,3]]}$

Let’s try the same experiment, but this time let \(mm\) be a copy of \(m\)

\[
(\%i6) \quad m : [1,[2,3]]$

(\%i7) \quad mm : \text{copy(m)}$

(\%i8) \quad \text{mm[2][1]} : x$

(\%i9) \quad m;

(\%o9) \quad \text{[1,[2,3]]}$

(\%i10) \quad mm;

(\%o10) \quad \text{[1,[x,3]]}$
\]

This time, the assignment to \(mm\) does not change the value of \(m\).
cholesky \((M)\)  
Function

Return the Cholesky factorization of the matrix selfadjoint (or hermitian) matrix \(M\).
The second argument defaults to 'generalring.' For a description of the possible values
for field, see \texttt{lu\_factor}.

ctranspose \((M)\)  
Function

Return the complex conjugate transpose of the matrix \(M\). The function \texttt{ctranspose}
uses \texttt{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:

\[
\begin{pmatrix}
1 & 0 \\
0 & 0 \\
0 & 2 \\
0 & 0
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 1 \\
0 & 0 \\
0 & 1 \\
0 & 0
\end{pmatrix}
\]

\[
\begin{pmatrix}
p & 0 \\
0 & q
\end{pmatrix}
\]

dotproduct \((u, v)\)  
Function

Return the dotproduct of vectors \(u\) and \(v\). This is the same as \texttt{conjugate\(\)} \(\texttt{transpose\(\)}\) \(u\) \(\times\) \(v\). The arguments \(u\) and \(v\) must be column vectors.

eigens\_by\_jacobi \((A)\)  
Function

Computes the eigenvalues and eigenvectors of \(A\) by the method of Jacobi rotations.
\(A\) must be a symmetric matrix (but it need not be positive definite nor positive
semidefinite). \texttt{field\_type} indicates the computational field, either \texttt{floatfield}
or \texttt{bigfloatfield}. If \texttt{field\_type} is not specified, it defaults to \texttt{floatfield}.
The elements of \(A\) must be numbers or expressions which evaluate to numbers via
\texttt{float} or \texttt{bfloat} (depending on \texttt{field\_type}).

Examples:

\[
\begin{pmatrix}
1/sqrt(2) & 1/sqrt(2) \\
-1/sqrt(2) & 1/sqrt(2)
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 1 \\
0 & 0
\end{pmatrix}
\]

\[
\begin{pmatrix}
sqrt(2) & sqrt(2) \\
0 & 0
\end{pmatrix}
\]
Chapter 67: linearalgebra

```lisp
(\%i2) L : matrix ([sqrt(3), 0], [0, sqrt(5)]);
[ sqrt(3) 0 ]
[ ]
[ 0 sqrt(5) ]
(\%o2) [

(\%i3) M : S . L . transpose (S);
[ sqrt(5) sqrt(3) sqrt(5) sqrt(3) ]
[ ------- + ------- ------- - ------- ]
[ 2 2 2 2 ]
(\%o3) [

(\%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 \text{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 \( (\text{col}) \)

Function

hankel \( (\text{col}, \text{row}) \)

Function

Return a Hankel matrix \( H \). The first column of \( H \) is \( \text{col} \); except for the first entry, the last row of \( H \) is \( \text{row} \). The default for \( \text{row} \) is the zero vector with the same length as \( \text{col} \).
**hessian** $(f, x)$

Returns the Hessian matrix of $f$ with respect to the list of variables $x$. The $(i, j)$-th element of the Hessian matrix is $\text{diff}(f, x[i], 1, x[j], 1)$.

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 ]
  (%o3) [     ]
          [ d F   d F ]
          [ ---   --- ]
          [ 2    da db ]
          [ da     ]
```

**hilbert_matrix** $(n)$

Return the $n$ by $n$ Hilbert matrix. When $n$ isn’t a positive integer, signal an error.

**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** $(M, \text{\texttt{ring generalring}})$

Invert a matrix $M$ by using the LU factorization. The LU factorization is done using the ring $\text{\texttt{rng}}$.

**jacobian** $(f, x)$

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 $\text{diff}(f[i], x[j])$.

Examples:

```
(%i11) jacobian ([sin (u - v), sin (u * v)], [u, v]);
          [ cos(v - u) - cos(v - u) ]
```
Chapter 67: linearalgebra

(%o1) \[ v \cos(u v) \quad u \cos(u v) \]

(%i2) depends ([F, G], [y, z]);
(%o2) [F(y, z), G(y, z)]

(%i3) jacobian ([F, G], [y, z]);
\[
\begin{bmatrix}
\frac{dF}{dy} & \frac{dF}{dz} \\
\frac{dG}{dy} & \frac{dG}{dz}
\end{bmatrix}
\]

kronecker_product \((A, B)\) Function
Return the Kronecker product of the matrices \(A\) and \(B\).

listp \((e, p)\) Function
listp \((e)\) Function
Given an optional argument \(p\), return \texttt{true} if \(e\) is a Maxima list and \(p\) evaluates to \texttt{true} for every list element. When \texttt{listp} is not given the optional argument, return \texttt{true} if \(e\) is a Maxima list. In all other cases, return \texttt{false}.

locate_matrix_entry \((M, r_1, c_1, r_2, c_2, f, rel)\) Function
Find a entry in the sub-matrix \(M\) that satisfies some property. Three cases:
\([1]\) \(rel = \text{'bool}\) 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 \texttt{false}.
\([2]\) \(rel = \text{'max}\) and \(f\) real-valued:
Scan the sub-matrix looking for an entry that maximizes \(f\). Return the index of a maximizing entry.
\([3]\) \(rel = \text{'min}\) 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 = \text{lu_factor} \((A, \text{field})\), then \texttt{lu_backsub} \((M, b)\) solves the linear system \(A x = b\).

lu_factor \((M, \text{field})\) Function
Return a list of the form \([LU, \text{perm}, \text{fld}]\), or \([LU, \text{perm}, \text{fld}, \text{lower-cnd upper-cnd}]\), 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 \texttt{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 \texttt{floatfield} or \texttt{complexfield}, the numbers \texttt{lower-cnd} and \texttt{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, \texttt{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 \texttt{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) \texttt{generalring} – the ring of Maxima expressions,
- (b) \texttt{floatfield} – the field of floating point numbers of the type double,
- (c) \texttt{complexfield} – the field of complex floating point numbers of the type double,
- (d) \texttt{crering} – the ring of Maxima CRE expressions,
- (e) \texttt{rationalfield} – the field of rational numbers,
- (f) \texttt{runningerror} – track the all floating point rounding errors,
- (g) \texttt{noncommutingring} – the ring of Maxima expressions where multiplication is the non-commutative dot operator.

When the field is \texttt{floatfield}, \texttt{complexfield}, or \texttt{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 \texttt{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 \texttt{mring}. The file \texttt{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 \texttt{complexfield}, for example, are Common Lisp complex numbers. Thus after computing the factorization, the matrix entries must be converted to Maxima expressions.

See also \texttt{get.lu_factors}.

Examples:

\[
(\%i1) \ w[i,j] := \text{random}(1.0) * \%i * \text{random}(1.0);
\]
\begin{verbatim}
(%o1) w := random(1.) + %i random(1.)

(%i2) 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]);

(%o7) [
    [ 1 - z 3 ]
    [ 3 8 - z ]
]

(%i8) lu_factor (M, generalring);

(%o8) [
    [ 1 - z 3 ]
    [ 3 9 ]
]

(%i9) get_lu_factors (%);

(%o9) [
    [ 1 0 ]
    [ 1 - z 3 ]
    [ 1 0 ]
    [ 1 - z ]
]

(%i10) %[1] . %[2] . %[3];

(%o10) [
    [ 1 - z 3 ]
    [ 3 8 - z ]
]

mat_cond (M, 1)  Function
matCond (M, inf)  Function

Return the p-norm matrix condition number of the matrix m. The allowed values
for p are 1 and inf. This function uses the LU factorization to invert the matrix
m. Thus the running time for mat_cond is proportional to the cube of the matrix
size; lu_factor determines lower and upper bounds for the infinity norm condition
number in time proportional to the square of the matrix size.

mat_norm (M, 1)  Function
mat_norm (M, inf)  Function
mat_norm (M, frobenius)  Function

Return the matrix p-norm of the matrix M. The allowed values for p are 1, inf,
and frobenius (the Frobenius matrix norm). The matrix M should be an unblocked
matrix.
\end{verbatim}
matrixp \((e, p)\)  
Function  
Given an optional argument \(p\), return \text{true} if \(e\) is a matrix and \(p\) evaluates to \text{true} for every matrix element. When \text{matrixp} is not given an optional argument, return \text{true} if \(e\) is a matrix. In all other cases, return \text{false}.

See also \text{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, \text{mat\_trace}(M) returns the same value as does \text{mat\_trace(mat\_unblocker(m))}.

mat_unblocker \((M)\)  
Function  
If \(M\) is a block matrix, unblock \(M\) one level. If \(M\) is a matrix, \text{mat\_unblocker} \((M)\) returns \(M\); otherwise, signal an error.

Thus if each entry of \(M\) is matrix, \text{mat\_unblocker} \((M)\) returns an unblocked matrix, but if each entry of \(M\) is a block matrix, \text{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 \text{matrix\_element\_mult} to "." and \text{matrix\_element\_transpose} to ’transpose. See also \text{mat\_fullunblocker}.

Example:

\begin{verbatim}
(%i1) A : matrix ([1, 2], [3, 4]);

(%o1)  
\begin{bmatrix}
1 & 2 \\
3 & 4 \\
\end{bmatrix}

(%i2) B : matrix ([7, 8], [9, 10]);

(%o2)  
\begin{bmatrix}
7 & 8 \\
9 & 10 \\
\end{bmatrix}

(%i3) matrix ([A, B]);

(%o3)  
\begin{bmatrix}
\begin{bmatrix} 1 & 2 \end{bmatrix} & \begin{bmatrix} 7 & 8 \end{bmatrix} \\
\begin{bmatrix} 3 & 4 \end{bmatrix} & \begin{bmatrix} 9 & 10 \end{bmatrix} \\
\end{bmatrix}

(%i4) mat_unblocker (%);

(%o4)  
\begin{bmatrix}
1 & 2 & 7 & 8 \\
3 & 4 & 9 & 10 \\
\end{bmatrix}
\end{verbatim}
**nonnegintegerp** \((n)\)

Function

Return true if and only if \(n \geq 0\) and \(n\) is an integer.

**nullspace** \((M)\)

Function

If \(M\) is a matrix, return \(\text{span} (v_1, \ldots, v_n)\), where the set \(\{v_1, \ldots, v_n\}\) 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 \(\text{span} ()\).

**nullity** \((M)\)

Function

If \(M\) is a matrix, return the dimension of the nullspace of \(M\).

**orthogonal_complement** \((v_1, \ldots, v_n)\)

Function

Return \(\text{span} (u_1, \ldots, u_m)\), where the set \(\{u_1, \ldots, u_m\}\) is a basis for the orthogonal complement of the set \(\{v_1, \ldots, v_n\}\).

Each vector \(v_1\) through \(v_n\) must be a column vector.

**polynomialp** \((p, L, \text{coeffp}, \text{exponp})\)

**polynomialp** \((p, L, \text{coeffp})\)

**polynomialp** \((p, L)\)

Function

Return true if \(p\) is a polynomial in the variables in the list \(L\), The predicate \(\text{coeffp}\) must evaluate to true for each coefficient, and the predicate \(\text{exponp}\) must evaluate to true for all exponents of the variables in \(L\). If you want to use a non-default value for \(\text{exponp}\), you must supply \(\text{coeffp}\) with a value even if you want to use the default for \(\text{coeffp}\).

polynomialp \((p, L, \text{coeffp})\) is equivalent to polynomialp \((p, L, \text{coeffp}, '\text{nonnegintegerp}')\).

polynomialp \((p, L)\) is equivalent to polynomialp \((p, L, '\text{constantp}', '\text{nonnegintegerp}')\).

The polynomial needn’t be expanded:

\[\begin{align*}
\texttt{(\%i1) polynomialp ((x + 1)*(x + 2), [x]);} \\
\texttt{(\%o1) \quad \text{true}} \\
\texttt{(\%i2) polynomialp ((x + 1)*(x + 2)^a, [x]);} \\
\texttt{(\%o2) \quad \text{false}}
\end{align*}\]

An example using non-default values for \(\text{coeffp}\) and \(\text{exponp}\):

\[\begin{align*}
\texttt{(\%i1) polynomialp ((x + 1)*(x + 2)^{(3/2)}, [x], \text{numberp}, \text{numberp});} \\
\texttt{(\%o1) \quad \text{true}} \\
\texttt{(\%i2) polynomialp ((x^{(1/2)} + 1)*(x + 2)^{(3/2)}, [x], \text{numberp}, \text{numberp});} \\
\texttt{(\%o2) \quad \text{true}}
\end{align*}\]

Polynomials with two variables:

\[\begin{align*}
\texttt{(\%i1) polynomialp (x^2 + 5*x*y + y^2, [x]);} \\
\texttt{(\%o1) \quad \text{false}} \\
\texttt{(\%i2) polynomialp (x^2 + 5*x*y + y^2, [x, y]);} \\
\texttt{(\%o2) \quad \text{true}}
\end{align*}\]
**polytocompanion** \((p, x)\)  
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** \((M, v)\)  
If \(M\) is a matrix with each entry a polynomial in \(v\), return a matrix \(M2\) such that  
1. \(M2\) is upper triangular,  
2. \(M2 = E_n \ldots E_1 M\), where \(E_1\) through \(E_n\) are elementary matrices whose entries are polynomials in \(v\),  
3. \(|\text{det} (M)| = |\text{det} (M2)|\),  
Note: This function doesn’t check that every entry is a polynomial in \(v\).

**rowop** \((M, i, j, \theta)\)  
If \(M\) is a matrix, return the matrix that results from doing the row operation \(R_i \leftarrow R_i - \theta \cdot R_j\). If \(M\) doesn’t have a row \(i\) or \(j\), signal an error.

**rank** \((M)\)  
Return the rank of that matrix \(M\). The rank is the dimension of the column space.  
Example:  
\[
(\%i1) \quad \text{rank(matrix([1,2],[2,4]));
(\%o1) \quad 1
(\%i2) \quad \text{rank(matrix([1,b],[c,d]));
(\%o2) \quad \{d-b \cdot c \neq 0\}
(\%o2) \quad 2
\]

**rowswap** \((M, i, j)\)  
If \(M\) is a matrix, swap rows \(i\) and \(j\). If \(M\) doesn’t have a row \(i\) or \(j\), signal an error.

**toeplitz** \((col)\)  
**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:  
\[
(\%i1) \quad \text{toeplitz([1,2,3],[x,y,z]);
\]
\[
\begin{bmatrix}
1 & y & z \\
 &  & \\
2 & 1 & y \\
 &  & \\
3 & 2 & 1 \\
\end{bmatrix}
\]
(\%o1)  
(\%i2) \quad \text{toeplitz([1,1+%i]);
\]
\[
\begin{bmatrix}
1 & 1 - %i \\
 & \\
%i + 1 & 1 \\
\end{bmatrix}
\]
(\%o2)
vandermonde_matrix ([x_1, ..., x_n])
Return a n by n matrix whose i-th row is [1, x_i, x_i^2, ... x_i^(n-1)].

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)
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.
68 lsquares

68.1 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.

68.2 Functions and Variables for lsquares

\[ \text{lsquares_estimates} \quad (D, x, e, a) \]
\[ \text{lsquares_estimates} \quad (D, x, e, a, initial = L, tol = t) \]

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. \text{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 \([a = \ldots, b = \ldots, c = \ldots]\). 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 \text{lsquares_estimates} are specified as equations and passed on verbatim to the function \text{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 \text{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.

\text{load(lsquares)} loads this function.

See also \text{lsquares_estimates_exact}, \text{lsquares_estimates_approximate}, \text{lsquares_mse}, \text{lsquares_residuals}, and \text{lsquares_residual_mse}.

Examples:

A problem for which an exact solution is found.

\begin{verbatim}
(%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 ]
[   ]
\end{verbatim}
lsquares_estimates

Estimate parameters \( a \) to minimize the mean square error \( MSE \), by constructing a system of equations and attempting to solve them symbolically via \texttt{solve}. The mean square error is an expression in the parameters \( a \), such as that returned by \texttt{lsquares_mse}.

The return value is a list of lists of equations of the form \([a = \ldots, b = \ldots, c = \ldots]\). 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 \texttt{lsquares_estimates}, \texttt{lsquares_estimates_approximate}, \texttt{lsquares_mse}, \texttt{lsquares_residuals}, and \texttt{lsquares_residual_mse}.

Example:

\begin{verbatim}
(%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  1  2  ]
     [           ]
     [ 9  2  1  ]
     [           ]
     [ -2  1  2 ]
     [           ]
     [ 3  2  2  ]
     [           ]
     [ 2  2  1  ]

(%o2) [ 1  1  1  ]
     [           ]
     [ 3  1  2  ]
     [           ]
     [ 2  1  2  ]

(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);

5 ==
  \( ((D + M) - C - M B - M A) \over i, 1 i, 3 i, 2 \)

5 ==
i = 1
---------------------------------------------

(%o3) [ 16  27  10921  107]

(%i4) lsquares_estimates_exact (mse, [A, B, C, D]);

5

[ [A = - --, B = - --, C = -----, D = - --]]
   16    16     1024   32
\end{verbatim}

\texttt{lsquares_estimates_approximate} \hspace{1em} (\textit{MSE}, \textit{a}, initial = \textit{L}, tol = \textit{t})  
\textit{Function}

Estimate parameters \textit{a} to minimize the mean square error \textit{MSE}, via the numerical minimization function \texttt{lbfgs}. The mean square error is an expression in the parameters \textit{a}, such as that returned by \texttt{lsquares_mse}.

The solution returned by \texttt{lsquares_estimates_approximate} is a local (perhaps global) minimum of the mean square error. For consistency with \texttt{lsquares_estimates_exact}, the return value is a nested list which contains one element, namely a list of equations of the form \([a = \ldots, b = \ldots, c = \ldots]\).

Additional arguments to \texttt{lsquares_estimates_approximate} are specified as equations and passed on verbatim to the function \texttt{lbfgs}.

\textit{MSE} must evaluate to a number when the parameters are assigned numeric values. This requires that the data from which \textit{MSE} was constructed comprise only numeric constants such as \texttt{pi} and \texttt{e} and literal numbers (integers, rationals, ordinary floats,
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]);
(%o2) [ 1 1 1 ]
     [ 3 1 2 ]
     [ 2 1 ]
[ 9 ]
[ 3 2 2 ]
[ 2 2 1 ]
(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);
5
====
\    2
> (D + M ) - C - M B - M A)
/ i, 1 i, 3 i, 2
====
(%o3) -------------------------------------------------------------
5
(%i4) lsquares_estimates_approximate (mse, [A, B, C, D], iprint = [-1, 0]);
(%o4) [[A = - 3.67850494740174, B = - 1.683070351177813, 
    C = 10.63469950148635, D = - 3.340357993175206]]
```

**lsquares_mse** (*D*, *x*, *e*)

Returns the mean square error (MSE), a summation expression, for the equation *e* in the variables *x*, with data *D*.

The MSE is defined as:

\[
\frac{1}{n} \sum_{i=1}^{n} \left[ \text{lhs}(e_i) - \text{rhs}(e_i) \right]^2,
\]

where *n* is the number of data and *e*[i] is the equation *e* evaluated with the variables in *x* assigned values from the *i*-th datum, *D*[i].
load(lsquares) loads this function.

Example:

(%i1) load (lsquares)$
(%i2) M : matrix ([1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1];

(%o2)

(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);

(%i4) diff (mse, D);

(%i5) ''mse, nouns;
Function \texttt{lsquares}\residuals\ (D, x, e, a)\quad\text{Returns the residuals for the equation } e \text{ with specified parameters } a \text{ and data } D.\ D \text{ is a matrix, } x \text{ is a list of variables, } e \text{ is an equation or general expression; if not an equation, } e \text{ is treated as if it were } e = 0.\ a \text{ is a list of equations which specify values for any free parameters in } e \text{ aside from } x.\ \text{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 \( a \).

\texttt{load(\textsc{lsquares})} loads this function.

Example:
\begin{verbatim}
(%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  1  ]
[         ]
[ 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]
[          ]
[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]
[64 64 32 64 64]
\end{verbatim}

Function \texttt{lsquares\_residual\_mse}\ (D, x, e, a)\quad\text{Returns the residual mean square error (MSE) for the equation } e \text{ with specified parameters } a \text{ and data } D.\ \text{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 \( a \).

\( \text{load(lsquares)} \) loads this function.

Example:

\[
\begin{align*}
\text{(%i1)} \quad \text{load (lsquares)} \\
\text{(%i2)} \quad M & : \text{matrix (}
\begin{bmatrix}
1,1,1, & [3/2,1,2], & [9/4,2,1], & [3,2,2], & [2,2,1] \\
1 & 1 & 1 \\
3 & \\
-1 & 2 & \\
2 & \\
9 & \\
-2 & 1 & \\
4 & \\
3 & 2 & 2 & \\
2 & 2 & 1 & 
\end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
\text{(%o2)} \\
\text{(%i3)} \quad a & : \text{lsquares_estimates (}
M, \ [z,x,y], \ (z + D)^2 = A*x + B*y + C, \ [A,B,C,D]);
\end{align*}
\]

\[
\begin{align*}
\text{(%o3)} \\
59 & 27 & 10921 & 107 \\
16 & 16 & 1024 & 32
\end{align*}
\]

\[
\begin{align*}
\text{(%i4)} \quad \text{lsquares_residual_mse (}
M, \ [z,x,y], \ (z + D)^2 = A*x + B*y + C, \ \text{first (a));}
\end{align*}
\]

\[
\begin{align*}
\text{(%o4)} \\
169 & \\
2560
\end{align*}
\]

\textbf{plsquares} \ (\text{Mat,VarList,depvars}) \quad \text{Function}

\textbf{plsquares} \ (\text{Mat,VarList,depvars,maxexpon}) \quad \text{Function}

\textbf{plsquares} \ (\text{Mat,VarList,depvars,maxexpon,maxdegree}) \quad \text{Function}

Multivariable polynomial adjustment of a data table by the "least squares" method. \( \text{Mat} \) is a matrix containing the data, \( \text{VarList} \) is a list of variable names (one for each \( \text{Mat} \) column, but use ",-" instead of varnames to ignore \( \text{Mat} \) columns), \( \text{depvars} \) is the name of a dependent variable or a list with one or more names of dependent variables (which names should be in \( \text{VarList} \)), \( \text{maxexpon} \) is the optional maximum exponent for each independent variable (1 by default), and \( \text{maxdegree} \) is the optional maximum polynomial degree (\( \text{maxexpon} \) by default); note that the sum of exponents of each term must be equal or smaller than \( \text{maxdegree} \), and if \( \text{maxdegree} = 0 \) then no limit is applied.

If \( \text{depvars} \) is the name of a dependent variable (not in a list), \textbf{plsquares} returns the adjusted polynomial. If \( \text{depvars} \) is a list of one or more dependent variables, \textbf{plsquares} returns a list with the adjusted polynomial(s). The Coefficients of Determination are displayed in order to inform about the goodness of fit, which ranges
from 0 (no correlation) to 1 (exact correlation). These values are also stored in the
global variable DETCOEF (a list if depvars is a list).

A simple example of multivariable linear adjustment:

```maxima
(%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:

```maxima
(%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?

```maxima
(%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 */
(diagonals = 27)
```

How many ways do we have to put two queens without they are threatened into a n
x n chessboard?

```maxima
(%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 */
(positions = 1288)
```

An example with six dependent variables:

```maxima
(%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").
69 minpack

69.1 Introduction to minpack

Minpack is a Common Lisp translation (via f2cl) of the Fortran library MINPACK, as obtained from Netlib.

69.2 Functions and Variables for minpack

\texttt{minpack} \texttt{lsquares} \texttt{(flist, varlist, guess [tolerance, jacobian])}

Compute the point that minimizes the sum of the squares of the functions in the list \texttt{flist}. The variables are in the list \texttt{varlist}. An initial guess of the optimum point must be provided in \texttt{guess}.

The optional keyword arguments, \texttt{tolerance} and \texttt{jacobian} provide some control over the algorithm. \texttt{tolerance} is the estimated relative error desired in the sum of squares. \texttt{jacobian} can be used to specify the Jacobian. If \texttt{jacobian} is not given or is \texttt{true} (the default), the Jacobian is computed from \texttt{flist}. If \texttt{jacobian} is \texttt{false}, a numerical approximation is used.

\texttt{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 \quad \text{improper input parameters.}
1 \quad \text{algorithm estimates that the relative error in the sum of squares is at most} \texttt{tolerance}.
2 \quad \text{algorithm estimates that the relative error between x and the solution is at most} \texttt{tolerance}.
3 \quad \text{conditions for info = 1 and info = 2 both hold.}
4 \quad \text{fvec is orthogonal to the columns of the jacobian to machine precision.}
5 \quad \text{number of calls to fcn with iflag = 1 has reached} 100*(n+1).
6 \quad \text{tol is too small. no further reduction in the sum of squares is possible.}
7 \quad \text{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],
   6.10932785920777e-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) \[
[5.060282149485331\times 10^{-11}, - 5.060282149491206\times 10^{-12},
2.179447843547218\times 10^{-11}, 2.179447843547218\times 10^{-11}],
3.534491794847031\times 10^{-21}, 5]
\]

Function

\textbf{minpack\_solve} \hspace{1em} (\textit{flist}, \textit{varlist}, \textit{guess} [, \textit{tolerance}, \textit{jacobian}])

Solve a system of \(n\) equations in \(n\) unknowns. The \(n\) equations are given in the list \textit{flist}, and the unknowns are in \textit{varlist}. An initial guess of the solution must be provided in \textit{guess}.

The optional keyword arguments, \textit{tolerance} and \textit{jacobian} provide some control over the algorithm. \textit{tolerance} is the estimated relative error desired in the sum of squares. \textit{jacobian} can be used to specify the Jacobian. If \textit{jacobian} is not given or is \textbf{true} (the default), the Jacobian is computed from \textit{flist}. If \textit{jacobian} is \textbf{false}, a numerical approximation is used.

\textbf{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 \hspace{1em} improper input parameters.
1 \hspace{1em} algorithm estimates that the relative error in the solution is at most \textit{tolerance}.
2 \hspace{1em} number of calls to \textit{fcn} with iflag = 1 has reached 100*(n+1).
3 \hspace{1em} tol is too small. no further reduction in the sum of squares is possible.
4 \hspace{1em} Iteration is not making good progress.
70 makeOrders

70.1 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) a b + a b + b + a b + a b + b + a b + a b + b + a + a + 1

where [0, 1] is associated with the term b and [2, 3] with \(a^2b^3\).

To use this function write first load("makeOrders").
71 mnewton

71.1 Introduction to mnewton

mnewton is an implementation of Newton’s method for solving nonlinear equations in one or more variables.

71.2 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 isn’t 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"), fpprec : 25, newtonepsilon : bfloat(10^(-fpprec+5)))$

(%i2) mnewton([2*3^u-v/u-5, u+2^v-4], [u, v], [2, 2]);

(%o2) \([u = 1.066618389595406772591173b0, v = 1.552564766841786450100418b0]\]

To use this function write first load("mnewton"). See also newtonepsilon and newtonmaxiter.
72 numericalio

72.1 Introduction to numericalio

\texttt{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, \texttt{numericalio} input functions can write data into that object. Otherwise, \texttt{numericalio} can guess, to some degree, the structure of an object to store the data, and return that object.

72.1.1 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 \texttt{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 \texttt{?} 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.

72.1.2 Separator flag values for input

The functions for plain-text input and output take an optional argument, \texttt{separator\_flag}, that tells what character separates data.

For plain-text input, these values of \texttt{separator\_flag} are recognized: \texttt{comma} for comma separated values, \texttt{pipe} for values separated by the vertical bar character |, \texttt{semicolon} for values separated by semicolon ;, and \texttt{space} for values separated by space or tab characters. If the file name ends in \texttt{.csv} and \texttt{separator\_flag} is not specified, \texttt{comma} is assumed. If the file name ends in something other than \texttt{.csv} and \texttt{separator\_flag} is not specified, \texttt{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 \texttt{false} between the separators. For example, 1234,,Foo is treated the same as 1234,false,Foo.

72.1.3 Separator flag values for output

For plain-text output, \texttt{tab}, for values separated by the tab character, is recognized as a value of \texttt{separator\_flag}, as well as \texttt{comma}, \texttt{pipe}, \texttt{semicolon}, and \texttt{space}.

In plain-text output, \texttt{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.
72.1.4 Binary floating-point input and output

numeralio 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, numeralio assumes the external byte order is most-significant byte first.

Other kinds of numbers are coerced to 8-byte floats; numeralio 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 numeralio is undefined.

numeralio includes functions to open a stream for reading or writing a stream of bytes.

72.2 Functions and Variables for plain-text input and output

read_matrix (S) Function
read_matrix (S, M) Function
read_matrix (S, separator_flag) Function
read_matrix (S, M, separator_flag) Function

read_matrix(S) reads the source S and returns its entire content as a matrix. The size of the matrix is inferred from the input data; each line of the file becomes one row of the matrix. If some lines have different lengths, read_matrix complains.

read_matrix(S, M) read the source S into the matrix M, until M is full or the source is exhausted. Input data are read into the matrix in row-major order; the input need not have the same number of rows and columns as M.

The source S may be a file name or a stream.

The recognized values of separator_flag are comma, pipe, semicolon, and space. If separator_flag is not specified, the file is assumed space-delimited.

read_array (S, A) Function
read_array (S, A, separator_flag) Function

Reads the source S into the array A, until A is full or the source is exhausted. Input data are read into the array in row-major order; the input need not conform to the dimensions of A.

The source S may be a file name or a stream.

The recognized values of separator_flag are comma, pipe, semicolon, and space. If separator_flag is not specified, the file is assumed space-delimited.

read_hashed_array (S, A) Function
read_hashed_array (S, A, separator_flag) Function

Reads the source S and returns its entire content as a hashed array. The source S may be a file name or a stream.
read_hashed_array treats the first item on each line as a hash key, and associates the remainder of the line (as a list) with the key. For example, the line 567 12 17 32 55 is equivalent to A[567]: [12, 17, 32, 55]. Lines need not have the same numbers of elements.

The recognized values of separator_flag are comma, pipe, semicolon, and space. If separator_flag is not specified, the file is assumed space-delimited.

read_nested_list (S)
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 elements. Empty lines are not ignored: an empty line yields an empty sublist.

The recognized values of separator_flag are comma, pipe, semicolon, and space. If separator_flag is not specified, the file is assumed space-delimited.

read_list (S)
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 (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.
72.3 Functions and Variables for binary input and output

`assume_external_byte_order (byte_order_flag)` Function
Tells `numericalio` the byte order for reading and writing binary data. Two values of `byte_order_flag` are recognized: `lsb` which indicates least-significant byte first, also called little-endian byte order; and `msb` which indicates most-significant byte first, also called big-endian byte order.

If not specified, `numericalio` assumes the external byte order is most-significant byte first.

`openr_binary (file_name)` Function
Returns an input stream of 8-bit unsigned bytes to read the file named by `file_name`.

`openw_binary (file_name)` Function
Returns an output stream of 8-bit unsigned bytes to write the file named by `file_name`.

`opena_binary (file_name)` Function
Returns an output stream of 8-bit unsigned bytes to append the file named by `file_name`.

`read_binary_matrix (S, M)` Function
Reads binary 8-byte floating point numbers from the source `S` into the matrix `M` until `M` is full, or the source is exhausted. Elements of `M` are read in row-major order.

The source `S` may be a file name or a stream.

The byte order in elements of the source is specified by `assume_external_byte_order`.

`read_binary_array (S, A)` Function
Reads binary 8-byte floating point numbers from the source `S` into the array `A` until `A` is full, or the source is exhausted. `A` must be an array created by `array` or `make_array`. Elements of `A` are read in row-major order.

The source `S` may be a file name or a stream.

The byte order in elements of the source is specified by `assume_external_byte_order`.

`read_binary_list (S)` Function
`read_binary_list (S, L)` Function

`read_binary_list(S)` reads the entire content of the source `S` as a sequence of binary 8-byte floating point numbers, and returns it as a list. The source `S` may be a file name or a stream.

`read_binary_list(S, L)` reads 8-byte binary floating point numbers from the source `S` until the list `L` is full, or the source is exhausted.

The byte order in elements of the source is specified by `assume_external_byte_order`. 
write_binary_data \((X, D)\)  
Function

Writes the object \(X\), comprising binary 8-byte IEEE 754 floating-point numbers, to the destination \(D\). Other kinds of numbers are coerced to 8-byte floats. \texttt{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 \texttt{array} or \texttt{make_array}; \(X\) cannot be an undeclared array or any other type of object. \texttt{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 \texttt{file_output_append} governs whether the output file is appended or truncated. When the destination is a stream, no special action is taken by \texttt{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 \texttt{assume_external_byte_order}. 

73 opsubst

73.1 Functions and Variables for opsubst

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:

```lisp
(%i1) load (opsubst)$
(%i2) opsubst(f,g,g(g(x)));
   f(f(x))
(%i3) opsubst(f,g,g(g));
   f(g)
(%i4) opsubst(f,g[x],g[x](z));
   g(z)
   x
(%i5) opsubst(tan, sin, sin(sin));
   tan(sin)
(%i6) opsubst([f=g,g=h],f(x));
   h(x)
```

Internally, Maxima does not use the unary negation, division, or the subtraction operators; thus:

```lisp
(%i8) opsubst("+","-",a-b);
   a - b
(%i9) opsubst("f","-",-a);
   a
(%i10) opsubst("^","/",a/b);
   a
   b
```

The internal representation of \(-a\times b\) is \(*(1,a,b)\); thus

```lisp
(%i11) opsubst("[","*", -a*b);
   [ - 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").
74 orthopoly

74.1 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).

74.1.1 Getting Started with orthopoly

load (orthopoly) loads the orthopoly package.

To find the third-order Legendre polynomial,

(%i1) legendre_p (3, x);

\[
\frac{5 (1 - x)^2}{2} + \frac{15 (1 - x)^2}{2} - \frac{6 (1 - x) + 1}{2}
\]

(%o1)

To express this as a sum of powers of \(x\), apply ratsimp or rat to the result.

(%i2) [ratsimp (%), rat (%)];

\[
\frac{5 x - 3 x}{2} - \frac{3 x}{2}, \quad \frac{5 x - 3 x}{2}
\]

(%o2)/R/

Alternatively, make the second argument to legendre_p (its “main” variable) a canonical rational expression (CRE).

(%i1) legendre_p (3, rat (x));

\[
\frac{5 x - 3 x}{2}
\]

(%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 877 for more information.

Most functions in `orthopoly` have a `gradeff` 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)
(%o2) n L (x) - (n + a) L (x) unit_step(n)
     n     n - 1
     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 (%i2, n = 0);
(%o3) 0
```

The `gradeff` property only applies to the “main” variable; derivatives with respect other arguments usually result in an error message; for example

```
(%i1) diff (hermite (n, x), x);
(%o1) 2 n H (x)
     n - 1
(%i2) diff (hermite (n, x), n);
```

Maxima doesn’t know the derivative of hermite with respect the first argument
-- an error. Quitting. To debug this try debugmode(true);

Generally, functions in `orthopoly` map over lists and matrices. For the mapping to fully evaluate, the option variables `doallmxops` and `listarith` must both be `true` (the defaults). To illustrate the mapping over matrices, consider

```
(%i1) hermite (2, x);
  2
(%o1) - 2 (1 - 2 x )
(%i2) m : matrix ([[0, x]], [y, 0]);
(%o2) [ 0 x ]
    [ y 0 ]
(%i3) hermite (2, m);
[   2    ]
[ - 2 - 2 (1 - 2 x ) ]
(%o3) [         ]
[     2    ]
In the second example, the $i, j$ element of the value is `hermite (2, m[i,j])`; this is not the same as computing $-2 + 4 m . m$, as seen in the next example.

```
(%i4) -2 * matrix ([1, 0], [0, 1]) + 4 * m . m;
     [ 4 x y - 2  0  ]
     [             ]
     [ 0  4 x y - 2 ]
(%o4)
```

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) \text{P}^{2/3}(x)
```

`orthopoly` supports translation into TeX; it also does two-dimensional output on a terminal.

```
(%i1) spherical_harmonic (1, m, theta, phi);
(%o1) Y^m_1(\vartheta,\varphi)
```

```
(%i2) tex (%);
$$Y_{1}^{m}\left(\vartheta,\varphi\right)$$
(%o2) false
```

```
(%i1) jacobi_p (n, a, a - b, x/2);
   (a, a - b) x
(%o1) \text{P}^{(-)}_{n}\left({{x}\over{2}}\right)
```

```
(%i4) tex (%);
$$P_{n}^{(a,a-b)}\left({{x}\over{2}}\right)$$
(%o4) false
```

### 74.1.2 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);
     n - 1
(%o1) (2 n - 1) P^{(x)}\left(\text{P}_{n}\right) + (1 - n) P^{(x)}\left(\text{P}_{n-2}\right)
```

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) \text{P}\left(x\right)^n
(i2) float (%);
  (o2) \text{P}\left(x\right)^n
(i3) ev (%, n=2, x=0.9);
  (o3) \text{P}\left(0.9\right)^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) (10)
(i2) float (x);
  (o2) (1.0)
(i3) float (x), pochhammer_max_index : 11;
  (o3) 3628800.0

Applying float doesn’t evaluate x to a float

(i2) float (x);
  (o2) (1.0)

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

```plaintext
(%i1) shifted_legendre_p (n, x) := legendre_p (n, 2*x - 1)
(%i2) shifted_legendre_p (2, rat (x));
2
(%o2)/R/ 6 x - 6 x + 1
(%i3) legendre_p (2, rat (x));
2
(%o3)/R/ --------
2
```

### 74.1.3 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.

```plaintext
(%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) 1.841360913516856309137022495891349369086890463668b-1
```

Let’s test that the actual error is smaller than the error estimate

```plaintext
(%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))$

(%o3) 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)

74.1.4 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)`.

### 74.1.5 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 `gradeff` statements.

To convert Pochhammer symbols into quotients of gamma functions, use `makegamma`.

```
(%i1) makegamma (pochhammer (x, n));

 gamma(x + n)
-------------
gamma(x)

(%i2) makegamma (pochhammer (1/2, 1/2));

1
---------
sqrt(%pi)
```

Derivatives of the Pochhammer symbol are given in terms of the `psi` function.

```
(%i1) diff (pochhammer (x, n), x);

(x) (psi (x + n) - psi (x))
-------------
n 0 0

(%i2) diff (pochhammer (x, n), n);

(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 \( x = -1, -2, \ldots, -n \). 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)
   ------------
   gamma(x)
(%o1)

(%i2) sublis ([x=11/3, n=-6], q);
   729
   ----
   2240
(%o2)

Alternatively, we can get this result directly.

(%i1) pochhammer (11/3, -6);
   729
   ----
   2240
(%o1)

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)];
(%o2)

1
[0, -, 1]
2

Do not redefine unit_step itself; some code in orthopoly requires that the unit step function be left-continuous.

74.1.6 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.

74.2 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) 
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) 
The Chebyshev function of the first kind.
Reference: Abramowitz and Stegun, equation 22.5.47, page 779.

chebyshev\_u (n, x) 
The Chebyshev function of the second kind.
Reference: Abramowitz and Stegun, equation 22.5.48, page 779.

gen\_laguerre (n, a, x) 
The generalized Laguerre polynomial of degree n.
Reference: Abramowitz and Stegun, equation 22.5.54, page 780.

hermite (n, x) 
The Hermite polynomial.
Reference: Abramowitz and Stegun, equation 22.5.55, page 780.

intervalp (e) 
Return true if the input is an interval and return false if it isn’t.

jacobi\_p (n, a, b, x) 
The Jacobi polynomial.
The Jacobi polynomials are actually defined for all a and b; however, the Jacobi polynomial weight \((1 - x)^a (1 + x)^b\) isn’t integrable for \(a <= -1\) or \(b <= -1\).
Reference: Abramowitz and Stegun, equation 22.5.42, page 779.

laguerre (n, x) 
The Laguerre polynomial.
Reference: Abramowitz and Stegun, equations 22.5.16 and 22.5.54, page 780.

legendre\_p (n, x) 
The Legendre polynomial of the first kind.
Reference: Abramowitz and Stegun, equations 22.5.50 and 22.5.51, page 779.

legendre\_q (n, x) 
The Legendre polynomial of the first kind.
Reference: Abramowitz and Stegun, equations 8.5.3 and 8.1.8.
Function
orthopoly_recur (f, args)
Returns a recursion relation for the orthogonal function family f with arguments args.
The recursion is with respect to the polynomial degree.

(%i1) orthopoly_recur (legendre_p, [n, x]);
   (2 n - 1) P (x) x + (1 - n) P (x)
      n - 1                     n - 2
(%o1) P (x) = -----------------------------------------
             n  n
The second argument to orthopoly_recur must be a list with the correct number of
arguments for the function f; if it isn't, Maxima signals an error.

(%i1) orthopoly_recur (jacobi_p, [n, x]);
Function jacobi_p needs 4 arguments, instead it received 2
-- an error. Quitting. To debug this try debugmode(true);
Additionally, when f isn't the name of one of the families of orthogonal polynomials,
an error is signalled.

(%i1) orthopoly_recur (foo, [n, x]);
A recursion relation for foo isn't known to Maxima
-- an error. Quitting. To debug this try debugmode(true);

Variable
orthopoly_returns_intervals
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.

Function
orthopoly_weight (f, args)
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
(%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.

Function
pochhammer (n, x)
The Pochhammer symbol. For nonnegative integers n with n <= pochhammer_max_index,
the expression pochhammer (x, n) evaluates to the product x (x + 1) (x + 2) ... (x + n - 1)
when n > 0 and to 1 when n = 0. For negative n, pochhammer (x, n) is defined as (-1)^n / pochhammer (1 - x, -n). Thus
Chapter 74: orthopoly

(%i1) pochhammer (x, 3);
(%o1) x (x + 1) (x + 2)

(%i2) pochhammer (x, -3);
(%o2) \frac{1}{(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) \frac{\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)

Variable pochhammer_max_index

Default value: 100

pochhammer ($n$, $x$) expands to a product if and only if $n \leq$ 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)

(%i3) pochhammer (x, 5), pochhammer_max_index : 3;
(%o3) (x)

(%i4) pochhammer (x, 6), pochhammer_max_index : 3;
(%o4) (x)

(%i5) pochhammer (x, 7), pochhammer_max_index : 3;
(%o5) (x)

(%i6) pochhammer (x, 8), pochhammer_max_index : 3;
(%o6) (x)

(%i7) pochhammer (x, 9), pochhammer_max_index : 3;
(%o7) (x)

(%i8) pochhammer (x, 10), pochhammer_max_index : 3;
(%o8) (x)

(%i9) pochhammer (x, 11), pochhammer_max_index : 3;
(%o9) (x)

(%i10) pochhammer (x, 12), pochhammer_max_index : 3;
(%o10) (x)

(%i11) pochhammer (x, 13), pochhammer_max_index : 3;
(%o11) (x)

(%i12) pochhammer (x, 14), pochhammer_max_index : 3;
(%o12) (x)

(%i13) pochhammer (x, 15), pochhammer_max_index : 3;
(%o13) (x)

(%i14) pochhammer (x, 16), pochhammer_max_index : 3;
(%o14) (x)

(%i15) pochhammer (x, 17), pochhammer_max_index : 3;
(%o15) (x)

(%i16) pochhammer (x, 18), pochhammer_max_index : 3;
(%o16) (x)

(%i17) pochhammer (x, 19), pochhammer_max_index : 3;
(%o17) (x)

(%i18) pochhammer (x, 20), pochhammer_max_index : 3;
(%o18) (x)

Reference: Abramowitz and Stegun, equation 6.1.16, page 256.

spherical_bessel_j ($n$, $x$)

The spherical Bessel function of the first kind.


spherical_bessel_y ($n$, $x$)

The spherical Bessel function of the second kind.


spherical_hankel1 ($n$, $x$)

The spherical Hankel function of the first kind.

Reference: Abramowitz and Stegun, equation 10.1.36, page 439.

spherical_hankel2 ($n$, $x$)

The spherical Hankel function of the second kind.

**spherical_harmonic** \((n, m, x, y)\)

The spherical harmonic function.

Reference: Merzbacher 9.64.

**unit_step** \((x)\)

The left-continuous unit step function; thus \(\text{unit\_step} \ (x)\) 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** \((n, a, x)\)

The ultraspherical polynomial (also known as the Gegenbauer polynomial).

Reference: Abramowitz and Stegun, equation 22.5.46, page 779.
75 plotdf

75.1 Introduction to plotdf

The function `plotdf` creates a plot of the direction field (also called slope field) for a first-order Ordinary Differential Equation (ODE) or a system of two autonomous first-order ODE’s.

Plotdf requires Xmaxima. It can be used from the console or any other interface to Maxima, but the resulting file will be sent to Xmaxima for plotting. Please make sure you have installed Xmaxima before trying to use plotdf.

To plot the direction field of a single ODE, the ODE must be written in the form:

\[
d\frac{y}{dx} = F(x, y)
\]

and the function \(F\) should be given as the argument for `plotdf`. If the independent and dependent variables are not \(x\), and \(y\), as in the equation above, then those two variables should be named explicitly in a list given as an argument to the `plotdf` command (see the examples).

To plot the direction field of a set of two autonomous ODE’s, they must be written in the form

\[
\frac{dx}{dt} = G(x, y) \quad \frac{dy}{dt} = F(x, y)
\]

and the argument for `plotdf` should be a list with the two functions \(G\) and \(F\), in that order; namely, the first expression in the list will be taken to be the time derivative of the variable represented on the horizontal axis, and the second expression will be the time derivative of the variable represented on the vertical axis. Those two variables do not have to be \(x\) and \(y\), but if they are not, then the second argument given to `plotdf` must be another list naming the two variables, first the one on the horizontal axis and then the one on the vertical axis.

If only one ODE is given, `plotdf` will implicitly admit \(x=t\), and \(G(x,y)=1\), transforming the non-autonomous equation into a system of two autonomous equations.

75.2 Functions and Variables for plotdf

```plaintext
plotdf (dydx, ...options...) Function
plotdf (dvdu, [u,v], ...options...) Function
plotdf ([dxdt,dydt], ...options...) Function
plotdf ([dudt,dvdt], [u,v], ...options...) Function
```

Displays a direction field in two dimensions \(x\) and \(y\).

\(dydx\), \(dxdt\) and \(dydt\) are expressions that depend on \(x\) and \(y\). \(dvdu\), \(dudt\) and \(dvdt\) are expressions that depend on \(u\) and \(v\). In addition to those two variables, the expressions can also depend on a set of parameters, with numerical values given with
the parameters option (the option syntax is given below), or with a range of allowed values specified by a sliders option.

Several other options can be given within the command, or selected in the menu. Integral curves can be obtained by clicking on the plot, or with the option trajectory_at. The direction of the integration can be controlled with the direction option, which can have values of forward, backward or both. The number of integration steps is given by nsteps and the time interval between them is set up with the tstep option. The Adams Moulton method is used for the integration; it is also possible to switch to an adaptive Runge-Kutta 4th order method.

Plot window menu:

The menu in the plot window has the following options: Zoom, will change the behavior of the mouse so that it will allow you to zoom in on a region of the plot by clicking with the left button. Each click near a point magnifies the plot, keeping the center at the point where you clicked. Holding the (Shift) key while clicking, zooms out to the previous magnification. To resume computing trajectories when you click on a point, select Integrate from the menu.

The option Config in the menu can be used to change the ODE(s) in use and various other settings. After configuration changes are made, the menu option Replot should be selected, to activate the new settings. If a pair of coordinates are entered in the field Trajectory at in the Config dialog menu, and the (enter) key is pressed, a new integral curve will be shown, in addition to the ones already shown. When Replot is selected, only the last integral curve entered will be shown.

Holding the right mouse button down while the cursor is moved, can be used to drag the plot sideways or up and down. Additional parameters such as the number of steps, the initial value of t and the x and y centers and radii, may be set in the Config menu.

A copy of the plot can be saved as a postscript file, using the menu option Save.

Plot options:

The plotdf command may include several commands, each command is a list of two or more items. The first item is the name of the option, and the remainder comprises the value or values assigned to the option.

The options which are recognized by plotdf are the following:

- **tstep** defines the length of the increments on the independent variable t, used to compute an integral curve. If only one expression dydx is given to plotdf, the x variable will be directly proportional to t. The default value is 0.1.

- **nsteps** defines the number of steps of length tstep that will be used for the independent variable, to compute an integral curve. The default value is 100.

- **direction** defines the direction of the independent variable that will be followed to compute an integral curve. Possible values are forward, to make the independent variable increase nsteps times, with increments tstep, backward, to make the independent variable decrease, or both that will lead to an integral curve that extends nsteps forward, and nsteps backward. The keywords right and left can be used as synonyms for forward and backward. The default value is both.
• \textit{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.

• \textit{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 \textit{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.

• \textit{trajectory\_at} defines the coordinates $x_{initial}$ and $y_{initial}$ for the starting point of an integral curve. The option is empty by default.

• \textit{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 \texttt{name=value}.

• \textit{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 \texttt{name=min:max}.

• \textit{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.

\textbf{Examples:}

• To show the direction field of the differential equation $y' = \exp(-x) + y$ and the solution that goes through $(2, -0.1)$:
\textbullet{} To obtain the direction field for the equation \(\frac{dy}{dx} = x - y^2\) and the solution with initial condition \(y(-1) = 3\), we can use the command:

\begin{verbatim}
(%i1) plotdf(x-y^2,[xfun, "sqrt(x);-sqrt(x)"],
[trajectory_at,-1,3], [direction,forward],
[y,-5,5], [x,-4,16])
\end{verbatim}

The graph also shows the function \(y = \sqrt{x}\).

\textbullet{} The following example shows the direction field of a harmonic oscillator, defined by the two equations \( \frac{dz}{dt} = v \) and \( \frac{dv}{dt} = -k \cdot z/m \), 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):

\[
(%i1) \text{plotdf([v,-k*z/m], [z,v], [parameters,"m=2,k=2"], [sliders,"m=1:5"], [trajectory_at,6,0])}
\]

To plot the direction field of the Duffing equation,

\[
m \cdot x'' + c \cdot x' + k \cdot x + b \cdot x^3 = 0,
\]

we introduce the variable \(y = x'\) and use:

- • To plot the direction field of the Duffing equation, \(m \cdot x'' + c \cdot x' + k \cdot x + b \cdot x^3 = 0\), we introduce the variable \(y = x'\) and use:
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])$

\begin{align*}
\text{w} & \quad 12 \\
    & \quad 8 \\
    & \quad 4 \\
    & \quad 0 \\
    & \quad -4 \\
    & \quad -8 \\
\text{a} & \quad -12
\end{align*}$

\begin{align*}
\text{t} & \quad 0.4 \\
    & \quad 0.8 \\
    & \quad 1.2 \\
    & \quad 1.6 \\
    & \quad 2
\end{align*}$
76 romberg

76.1 Functions and Variables for romberg

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 rombergtol. romberg terminates successfully when the absolute difference between successive approximations is less than rombergabs, or the relative difference in successive approximations is less than rombergtol. Thus when rombergabs is 0.0 (the default) only the relative error test has any effect on romberg.

romberg halves the stepsize at most rombergit times before it gives up; the maximum number of function evaluations is therefore 2^rombergit. If the error criterion established by rombergabs and rombergtol is not satisfied, romberg prints an error message. romberg always makes at least rombergmin iterations; this is a heuristic intended to prevent spurious termination when the integrand is oscillatory.

romberg repeatedly evaluates the integrand after binding the variable of integration to a specific value (and not before). This evaluation policy makes it possible to nest calls to romberg, to compute multidimensional integrals. However, the error calculations do not take the errors of nested integrations into account, so errors may be underestimated. Also, methods devised especially for multidimensional problems may yield the same accuracy with fewer function evaluations.

load(romberg) loads this function.

See also QUADPACK, a collection of numerical integration functions.

Examples:

A 1-dimensional integration.

(%i1) load (romberg);  
(%o1) /usr/share/maxima/5.11.0/share/numeric/romberg.lisp  
(%i2) f(x) := 1/((x - 1)^2 + 1/100) + 1/((x - 2)^2 + 1/1000)  
+ 1/((x - 3)^2 + 1/200);  
1 1 1  
(%o2) f(x) := ----------- + ----------- + -----------  
2 1 2 1 2 1  
(x - 1) + --- (x - 2) + ---- (x - 3) + ---
(%i3) rombergtol : 1e-6;
(%o3) 9.9999999999999995E-7
(%i4) rombergit : 15;
(%o4) 15
(%i5) estimate : romberg (f(x), x, -5, 5);
(%o5) 173.6730736617464
(%i6) exact : integrate (f(x), x, -5, 5);
(%o6) 10 sqrt(10) atan(70 sqrt(10))
+ 10 sqrt(10) atan(30 sqrt(10)) + 10 sqrt(2) atan(80 sqrt(2))
+ 10 sqrt(2) atan(20 sqrt(2)) + 10 atan(60) + 10 atan(40)
(%i7) abs (estimate - exact) / exact, numer;
(%o7) 7.5527060865060088E-11
A 2-dimensional integration, implemented by nested calls to romberg.
(%i1) load (romberg);
(%o1) /usr/share/maxima/5.11.0/share/numeric/romberg.lisp
(%i2) g(x, y) := x*y / (x + y);
 x y
(%o2) g(x, y) := ----- 
 x + y
(%i3) rombergtol : 1e-6;
(%o3) 9.9999999999999995E-7
(%i4) estimate : romberg (romberg (g(x, y), y, 0, x/2), x, 1, 3);
(%o4) 0.81930239628356
(%i5) assume (x > 0);
(%o5) [x > 0]
(%i6) integrate (integrate (g(x, y), y, 0, x/2), x, 1, 3);

3
 2 log(-) - 1
(%o6) - 9 log(-) + 9 log(3) + ------------- + -
 2     6   2
(%i7) exact : radcan (%);

26 log(3) - 26 log(2) - 13
- ---------------------------
 3
(%i8) abs (estimate - exact) / exact, numer;
(%o8) 1.3711979871851024E-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.
Chapter 76: romberg

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 $2^{rombergit}$. romberg always makes at least rombergmin iterations; this is a heuristic intended to prevent spurious termination when the integrand is oscillatory.  
See also rombergabs and rombergtol.

rombergmin  
Option variable  
Default value: 0  
romberg always makes at least rombergmin iterations; this is a heuristic intended to prevent spurious termination when the integrand is oscillatory.  
See also rombergit, rombergabs, and rombergtol.

rombergtol  
Option variable  
Default value: 1e-4  
The accuracy of 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.
Chapter 77: simplex

77 simplex

77.1 Introduction to simplex

simplex is a package for linear optimization using the simplex algorithm.

Example:

```lisp
(%i1) load("simplex")$  
(%i2) minimize_lp(x+y, [3*x+2*y>2, x+4*y>3]);
 (9) (7) (1)
   (--) [y = --, x = -]
 (10) (10) (5)
```

77.2 Functions and Variables for simplex

epsilon_lp

Option variable

Default value: 10^-8

Epsilon used for numerical computations in linear_program.

See also: linear_program.

linear_program (A, b, c)

Function

linear_program is an implementation of the simplex algorithm. linear_program(A, b, c) computes a vector x for which c.x is minimum possible among vectors for which A.x = b and x >= 0. Argument A is a matrix and arguments b and c are lists.

linear_program returns a list which contains the minimizing vector x and the minimum value c.x. If the problem is not bounded, it returns "Problem not bounded!" and if the problem is not feasible, it returns "Problem not feasible!".

To use this function first load the simplex package with load(simplex);.

Example:

```lisp
(%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)
   [--) [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])

Minimizes a linear objective function obj subject to some linear constraints cond. cond a list of linear equations or inequalities. In strict inequalities > is replaced by >= and < by <=. The optional argument pos is a list of decision variables which are assumed to be positive.

If the minimum exists, minimize_lp returns a list which contains the minimum value of the objective function and a list of decision variable values for which the minimum is attained. If the problem is not bounded, minimize_lp returns "Problem not bounded!" and if the problem is not feasible, it returns "Problem not feasible!".

The decision variables are not assumed to be nonegative by default. If all decision variables are nonegative, set nonegative_lp to true. If only some of decision variables are positive, list them in the optional argument pos (note that this is more efficient than adding constraints).

minimize_lp uses the simplex algorithm which is implemented in maxima linear_program function.

To use this function first load the simplex package with load(simplex);

Examples:

(%%11) minimize_lp(x+y, [3*x+y=0, x+2*y>2]);
   4    6     2
(%%o1)       [-, [y = -, x = - -]]
          5    5     5

(%%12) minimize_lp(x+y, [3*x+y>0, x+2*y>2]), nonegative_lp=true;
(%%o2) [1, [y = 1, x = 0]]

(%%13) minimize_lp(x+y, [3*x+y=0, x+2*y>2]), nonegative_lp=true;
(%%o3) Problem not feasible!

(%%14) minimize_lp(x+y, [3*x+y>0]);
(%%o4) Problem not bounded!

See also: maximize_lp, nonegative_lp, epsilon_lp.

nonegative_lp

Option variable

Default value: false

If nonegative_lp is true all decision variables to minimize_lp and maximize_lp are assumed to be positive.

See also: minimize_lp.
Chapter 78: simplification

78 simplification

78.1 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.

78.2 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;
(\%o2) x^2
(\%i3) diff (abs (x), x);
(\%o3) \frac{x}{abs(x)}
(\%i4) cosh (abs (x));
(\%o4) \cosh(x)

78.3 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.
Function \texttt{facsum}(expr, \texttt{arg}_1, \ldots, \texttt{arg}_n)\]

Returns a form of \texttt{expr} which depends on the arguments \texttt{arg}_1, \ldots, \texttt{arg}_n. The arguments can be any form suitable for \texttt{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 \texttt{factor} on the coefficients of the arguments, \texttt{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 \texttt{facsum} with respect to more complicated subexpressions, such as \texttt{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 \texttt{facsum} with respect to all \texttt{log}'s. In this situation, one may include among the arguments either the specific \texttt{log}'s which are to be treated in this way, or alternatively, either the expression \texttt{operator(log)} or \texttt{'operator(log)}. If one wished to \texttt{facsum} the expression \texttt{expr} with respect to the operators \texttt{op}_1, \ldots, \texttt{op}_n, one would evaluate \texttt{facsum(expr, operator(op}_1, \ldots, \texttt{op}_n)). The \texttt{operator} form may also appear inside list arguments.

In addition, the setting of the switches \texttt{facsum_combine} and \texttt{nextlayerfactor} may affect the result of \texttt{facsum}.

\texttt{nextlayerfactor} \hspace{1cm} \text{Global variable}

Default value: \texttt{false}

When \texttt{nextlayerfactor} is \texttt{true}, recursive calls of \texttt{facsum} are applied to the factors of the factored form of the coefficients of the arguments.

When \texttt{false}, \texttt{facsum} is applied to each coefficient as a whole whenever recursive calls to \texttt{facsum} occur.

Inclusion of the atom \texttt{nextlayerfactor} in the argument list of \texttt{facsum} has the effect of \texttt{nextlayerfactor: true}, but for the next level of the expression \textit{only}. Since \texttt{nextlayerfactor} is always bound to either \texttt{true} or \texttt{false}, it must be presented single-quoted whenever it appears in the argument list of \texttt{facsum}.

\texttt{facsum_combine} \hspace{1cm} \text{Global variable}

Default value: \texttt{true}

\texttt{facsum_combine} controls the form of the final result returned by \texttt{facsum} when its argument is a quotient of polynomials. If \texttt{facsum_combine} is \texttt{false} then the form will be returned as a fully expanded sum as described above, but if \texttt{true}, then the expression returned is a ratio of polynomials, with each polynomial in the form described above.

The \texttt{true} setting of this switch is useful when one wants to \texttt{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.
Chapter 78: simplification

**factorfacsum** $(expr, \text{arg}_1, \ldots, \text{arg}_n)$

Function

Returns a form of $expr$ which is obtained by calling `facsum` on the factors of $expr$ with $\text{arg}_1, \ldots, \text{arg}_n$ 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** $(expr, \text{arg}_1, \ldots, \text{arg}_n)$

Function

If several expressions have been simplified with `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.

### 78.4 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** $([f_1, \ldots, f_n], x)$

Function

Returns the Wronskian matrix of the list of expressions $[f_1, \ldots, f_n]$ 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:

```lisp
(%i1) load(functs)$
(%i2) wronskian([f(x), g(x)], x);
(%o2) matrix([f(x),g(x)],[diff(f(x),x,1),diff(g(x),x,1)])
```

**tracematrix** $(M)$

Function

Returns the trace (sum of the diagonal elements) of matrix $M$.

To use this function write first `load(functs)`.

**rational** $(z)$

Function

Multiplies numerator and denominator of $z$ by the complex conjugate of denominator, thus rationalizing the denominator. Returns canonical rational expression (CRE) form if given one, else returns general form.

To use this function write first `load(functs)`.

**logand** $(x, y)$

Function

Returns logical (bit-wise) "and" of arguments $x$ and $y$.

To use this function write first `load(functs)`.
logor(x,y) Function
Returns logical (bit-wise) "or" of arguments x and y.
To use this function write first load(functs).

logxor(x,y) Function
Returns logical (bit-wise) exclusive-or of arguments x and y.
To use this function write first load(functs).

nonzeroandfreeof(x, expr) Function
Returns true if expr is nonzero and freeof(x, expr) returns true. Returns false otherwise.
To use this function write first load(functs).

linear(expr, x) Function
When expr is an expression linear in variable x, linear returns \(a \times x + b\) where \(a\) is nonzero, and \(a\) and \(b\) are free of \(x\). Otherwise, linear returns expr.
To use this function write first load(functs).

gcdivide(p, q) Function
When takegcd is true, gcdivide divides the polynomials \(p\) and \(q\) by their greatest common divisor and returns the ratio of the results.
When takegcd is false, gcdivide returns the ratio \(p/q\).
To use this function write first load(functs).

arithmetic(a, d, n) Function
Returns the \(n\)-th term of the arithmetic series \(a, a + d, a + 2d, \ldots, a + (n - 1)d\).
To use this function write first load(functs).

geometric(a, r, n) Function
Returns the \(n\)-th term of the geometric series \(a, a \times r, a \times r^2, \ldots, a \times r^{(n - 1)}\).
To use this function write first load(functs).

harmonic(a, b, c, n) Function
Returns the \(n\)-th term of the harmonic series \(a/b, a/(b + c), a/(b + 2c), \ldots, a/(b + (n - 1)c)\).
To use this function write first load(functs).

arithsum(a, d, n) Function
Returns the sum of the arithmetic series from 1 to \(n\).
To use this function write first load(functs).
geosum \( (a, r, n) \)
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 \texttt{load(functs)}.

gaussprob \( (x) \)
Returns the Gaussian probability function \( \%e^{-(x^2/2)} / \sqrt{2*\%pi} \).
To use this function write first \texttt{load(functs)}.

gd \( (x) \)
Returns the Gudermannian function \( 2*\text{atan}(\%e^x)-\%pi/2 \).
To use this function write first \texttt{load(functs)}.

agd \( (x) \)
Returns the inverse Gudermannian function \( \log (\tan(\%pi/4 + x/2)) \).
To use this function write first \texttt{load(functs)}.

vers \( (x) \)
Returns the versed sine \( 1 - \cos (x) \).
To use this function write first \texttt{load(functs)}.

covers \( (x) \)
Returns the covered sine \( 1 - \sin (x) \).
To use this function write first \texttt{load(functs)}.

exsec \( (x) \)
Returns the exsecant \( \sec (x) - 1 \).
To use this function write first \texttt{load(functs)}.

hav \( (x) \)
Returns the haversine \( (1 - \cos(x))/2 \).
To use this function write first \texttt{load(functs)}.

combination \( (n, r) \)
Returns the number of combinations of \( n \) objects taken \( r \) at a time.
To use this function write first \texttt{load(functs)}.

permutation \( (n, r) \)
Returns the number of permutations of \( r \) objects selected from a set of \( n \) objects.
To use this function write first \texttt{load(functs)}.
78.5 Package ineq

The `ineq` package contains simplification rules for inequalities.

Example session:

```
(%i1) load(ineq)
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
(%i2) a>=4; /* a sample inequality */
     4
(%o2) a >=
(%i3) (b>c)+%; /* add a second, strict inequality */
     4
(%o3) b + a > c +
(%i4) 7*(x<y); /* multiply by a positive number */
     7
(%o4) 7 x < 7 y
(%i5) -2*(x>=3*z); /* multiply by a negative number */
     6
(%o5) - 2 x <= - 6 z
(%i6) (1+a^2)*(1/(1+a^2)<=1); /* Maxima knows that 1+a^2 > 0 */
2
(%o6)
1 <= a + 1
(%i7) assume(x>0)$
(%o7)
2 x < 3 x
(%i8) a>=b; /* another inequality */
(%o8)
 a >= b
(%i9) 3+%; /* add something */
 (%o9) a + 3 >= b + 3
(%i10) %-3; /* subtract it out */
(%o10) a >= b
(%i11) a>=c-b; /* yet another inequality */
(%o11) a >= c - b
(%i12) b+%; /* add b to both sides */
(%o12) b + a >= c
(%i13) %-c; /* subtract c from both sides */
(%o13) - c + b + a >= 0
(%i14) %-1; /* multiply by -1 */
(%o14) c - b - a <= 0
(%i15) (z-1)^2>-2*z; /* determining truth of assertion */
2
(%o15) (z - 1) > - 2 z
(%i16) expand(%)+2*z; /* expand this and add 2*z to both sides */
2
(%o16) z + 1 > 0
(%i17) %,pred;
(%o17) true
```
Be careful about using parentheses around the inequalities: when the user types in \((A > B) + (C = 5)\) the result is \(A + C > B + 5\), but \(A > B + C = 5\) is a syntax error, and \((A > B + C) = 5\) is something else entirely.

Do \texttt{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:

\begin{verbatim}
(%i1) eq: a > b;
(%o1) a > b
(%i2) 2*eq;
(%o2) 2 (a > b)
(%i3) % - eq;
(%o3) a > b
\end{verbatim}

Another problem is 0 times an inequality; the default to have this turn into 0 has been left alone. However, if you type \texttt{X*some_inequality} and Maxima asks about the sign of \(X\) and you respond \texttt{zero} (or \texttt{z}), the program returns \texttt{X*some_inequality} and not use the information that \(X\) is 0. You should do \texttt{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 \texttt{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 \texttt{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: \texttt{*rule1}, \texttt{...}, \texttt{*rule8}, \texttt{+rule1}, \texttt{...}, \texttt{+rule18}, and you must enclose the rulename in quotes to refer to it, as in \texttt{remrule ("+", "+rule1"}) to specifically remove the first rule on "+" or \texttt{disprule ("*rule2")} to display the definition of the second multiplicative rule.

### 78.6 Package rducon

**reduce_consts** (expr)

Replaces constant subexpressions of \(expr\) with constructed constant atoms, saving the definition of all these constructed constants in the list of equations \texttt{const_eqns}, and returning the modified \(expr\). Those parts of \(expr\) are constant which return \texttt{true} when operated on by the function \texttt{constantp}. Hence, before invoking \texttt{reduce_consts}, one should do

\begin{verbatim}
declare ([objects to be given the constant property], constant)$
\end{verbatim}

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

\begin{verbatim}
map ('fortran, const_eqns)$
\end{verbatim}

Variables besides \texttt{const_eqns} which affect \texttt{reduce_consts} are:
**const_prefix** (default value: **xx**) is the string of characters used to prefix all symbols generated by **reduceconsts** 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 **reduceconsts**.

**load (rducon)** loads this function. **demo (rducon)** shows a demonstration of this function.

### 78.7 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.

### 78.8 Package sqdnst

**sqrtdenest** (*expr*)

Function

Denests **sqrt** of simple, numerical, binomial surds, where possible. E.g.

```
(%i11) load (sqdnst)$
(%i12) sqrt(sqrt(3)/2+1)/sqrt(11*sqrt(2)-12);
     sqrt(3)
     sqrt(------- + 1)
     2
    ---------------------
    sqrt(11 sqrt(2) - 12)
(%o12) sqrt(3) 1
     -------- + -
     2 2
    1/4 3/4
    3 2 - 2
```
Sometimes it helps to apply \texttt{sqrtdenest} more than once, on such as \((19601-13860 \sqrt{2})^{7/4}\).
\texttt{load (sqdnnst)} loads this function.
Chapter 79: solve_rec

79 solve_rec

79.1 Introduction to solve_rec

solve_rec is a package for solving linear recurrences with polynomial coefficients. A demo is available with demo(solve_rec);

Example:

(%i1) load("solve_rec")$
(%i2) solve_rec((n+4)*s[n+2] + s[n+1] - (n+1)*s[n], s[n])

\( n \frac{\%k (2 n + 3) (- 1)}{1} \) \( \frac{\%k}{2} \)

\( s = \frac{\text{-----------------------------}}{n (n + 1) (n + 2)} \) \( \frac{\text{-----------------------------}}{(n + 1) (n + 2)} \)

79.2 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;
(%o3) x = x + \--
    n + 2 n + 1 n

(%i4) solve_rec(rec, x[n])

WARNING: found some hypergeometrical solutions!

(%o4) x = \%k n
     n - 1

(%i5) reduce_order(rec, n, x[n]);
(%i5) reduce_order(rec, n, x[n]);
(%t5) x = n \%z
     n n

\( n - 1 \)

\( \%z = > \%u \)
\( n / \%j \)

\( \%j = 0 \)

(%o6) (- n - 2) \%u - \%u
     n + 1 n
(%i6) solve_rec((n+2)*%u[n+1] + %u[n], %u[n]);
   n
%k (- 1)
1
(%o6) %u = ------------
     n (n + 1)!

So the general solution is

\[
\sum_{j=0}^{n-1} \frac{(-1)}{2(j+1)!} + \frac{(-1)}{1} \sum_{j=0}^{2} \frac{n}{(j+1)!}
\]

simplify_products
Option variable

Default value: true

If simplify_products is true, solve_rec will try to simplify products in result.
See also: solve_rec.

simplify_sum (expr)
Function

Tries to simplify all sums appearing in expr to a closed form.
To use this function first load the simplify_sum package with load(simplify_sum).
Example:

(%i1) load("simplify_sum")$
(%i2) sum(binomial(n+k,k)/2^k,k,1,n)+sum(binomial(2*n,2*k),k,1,n);
   n
\sum_{k=1}^{n} \binomial(n+k, k) / 2^k
\sum_{k=1}^{n} \binomial(2n, 2k)
(%o2) \sum_{k=1}^{n} \binomial(n+k, k) / 2^k
(%i3) simplify_sum(%);
2 n - 1 n
2 + 2 - 2
(%o3)

solve_rec (eqn, var, [init])
Function

Solves for hypergeometrical solutions to linear recurrence eqn with polynomials coefficient in variable var. Optional arguments init are initial conditions.
solve_rec can solve linear recurrences with constant coefficients, finds hypergeometrical solutions to homogeneous linear recurrences with polynomial coefficients, rational solutions to linear recurrences with polynomial coefficients and can solve Ricatti type recurrences.
Chapter 79: solve_rec

Note that the running time of the algorithm used to find hypergeometrical solutions is exponential in the degree of the leading and trailing coefficient.

To use this function first load the solve_rec package with load(solve_rec);.

Example of linear recurrence with constant coefficients:

```
(%i2) solve_rec(a[n]=a[n-1]+a[n-2]+n/2^n, a[n]);
n n
(sqrt(5) - 1) %k (- 1)
1 n
a = ------------------------- - ----
n 2
n n 5 2
2
(sqrt(5) + 1) %k
2 2
+ ------------------ - ----
n 2
n 2
```

Example of linear recurrence with polynomial coefficients:

```
(%i7) 2*x*(x+1)*y[x] - (x^2+3*x-2)*y[x+1] + (x-1)*y[x+2];
2
(%o7) (x - 1) y - (x + 3 x - 2) y + 2 x (x + 1) y
x + 2
x + 1
x
(%i8) solve_rec(%, y[x], y[1]=1, y[3]=3);
x
3 2 x!
y = ---- - --
x 4 2
```

Example of Ricatti type recurrence:

```
(%i2) x*y[x+1]*y[x] - y[x+1]/(x+2) + y[x]/(x-1) = 0;
y y
x + 1 x
(%o2) x y y
x x + 1 x + 2 x - 1
(%i3) solve_rec(%, y[x], y[3]=5)$
(%i4) ratsimp(minfactorial(factcomb(%)));
3
30 x - 30 x
```

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]);
(%o2) a = ------------------

1
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

\[
\sum_{k=lo}^{hi} \text{summand}
\]

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) \text{binomial(n, k)}

(%i3) summand_to_rec(summand,k,n);
(%o3) 2 sm - sm = 0

\text{n} \quad \text{n + 1}

(%i7) summand: binom(n, k)/(k+1);
(%o7) \text{binomial(n, k)}

\text{-------------}

k + 1

(%i8) summand_to_rec(summand, [k, 0, n], n);
(%o8) 2 (n + 1) sm - (n + 2) sm = - 1

\text{n} \quad \text{n + 1}
80 stats

80.1 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'.

80.2 Functions and Variables for inference_result

\texttt{inference\_result(} \texttt{title, values, numbers)}

Function

Constructs an inference_result object of the type returned by the stats functions. Argument \texttt{title} is a string with the name of the procedure; \texttt{values} is a list with elements of the form \texttt{symbol = value} and \texttt{numbers} is a list with positive integer numbers ranging from one to \texttt{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 \texttt{base}, \texttt{height}, \texttt{diagonal}, \texttt{area}, and \texttt{perimeter}, but only the first, second, fifth, and fourth will be displayed. The \texttt{diagonal} is stored in this object, but it is not displayed; to access its value, make use of function \texttt{take\_inference}.

\begin{verbatim}
(\%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
\end{verbatim}
\begin{verbatim}
Inferencep (obj) Function
Returns true or false, depending on whether obj is an inference_result object or not.

Items_inference (obj) Function
Returns a list with the names of the items stored in obj, which must be an inference_result object.
Example:
The inference_result object stores two values, named 'pi and 'e, but only the second is displayed. The items_inference function returns the names of all items, no matter they are displayed or not.

\begin{verbatim}
(%i1) load(inference_result)$
(%i2) inference_result("Hi", ['pi=%pi,'e=%e],[2]);
     | Hi
    (%o2)
     | e = %e
(%i3) items_inference(%);
    (%o3) [pi, e]
\end{verbatim}

Take_inference (n, obj) Function
Take_inference (name, obj) Function
Take_inference (list, obj) Function
Returns the n-th value stored in obj if n is a positive integer, or the item named name if this is the name of an item. If the first argument is a list of numbers and/or symbols, function take_inference returns a list with the corresponding results.
Example:
Given an inference_result object, function take_inference is called in order to extract some information stored in it.

\begin{verbatim}
(%i1) load(inference_result)$
(%i2) b: 3$ h: 2$
(%i3) sol: inference_result("Rectangle",
     ['base=b,
     'height=h,
     'diagonal=sqrt(b^2+h^2),
     'perimeter=10,
     'area=6];
    (%o3)
\end{verbatim}
\end{verbatim}

See also take_inference.
\end{verbatim}

Inferencep (obj) Function
Returns true or false, depending on whether obj is an inference_result object or not.

Items_inference (obj) Function
Returns a list with the names of the items stored in obj, which must be an inference_result object.
Example:
The inference_result object stores two values, named 'pi and 'e, but only the second is displayed. The items_inference function returns the names of all items, no matter they are displayed or not.

\begin{verbatim}
(%i1) load(inference_result)$
(%i2) inference_result("Hi", ['pi=%pi,'e=%e],[2]);
     | Hi
    (%o2)
     | e = %e
(%i3) items_inference(%);
    (%o3) [pi, e]
\end{verbatim}

Take_inference (n, obj) Function
Take_inference (name, obj) Function
Take_inference (list, obj) Function
Returns the n-th value stored in obj if n is a positive integer, or the item named name if this is the name of an item. If the first argument is a list of numbers and/or symbols, function take_inference returns a list with the corresponding results.
Example:
Given an inference_result object, function take_inference is called in order to extract some information stored in it.

\begin{verbatim}
(%i1) load(inference_result)$
(%i2) b: 3$ h: 2$
(%i3) sol: inference_result("Rectangle",
     ['base=b,
     'height=h,
     'diagonal=sqrt(b^2+h^2),
     'perimeter=10,
     'area=6];
    (%o3)
\end{verbatim}
\end{verbatim}
Chapter 80: stats

\[
\begin{align*}
\text{Area} &= bh, \\
\text{Perimeter} &= 2(b+h)
\end{align*}
\]

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>base = 3</td>
</tr>
<tr>
<td>height = 2</td>
</tr>
<tr>
<td>perimeter = 10</td>
</tr>
<tr>
<td>area = 6</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{(%o3)} & \text{take_inference('base,sol);} \\
\text{(%o4)} & 3 \\
\text{(%o5)} & 10 \\
\text{(%o6)} & \text{take_inference([1,'diagonal],sol);} \\
\text{(%o7)} & \text{[3, sqrt(13)]}
\end{align*}
\]

See also inference_result and take_inference.

80.3 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)  
Function

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.
- 'conflevel, default 95/100, confidence level for the confidence interval; it must be an expression which takes a value in (0,1).
- 'asymptotic, default false, indicates whether it performs an exact t-test or an asymptotic one based on the Central Limit Theorem; valid values are true and false.
The output of function `test_mean` is an `inference_result` Maxima object showing the following results:

1. `mean_estimate`: the sample mean.
2. `conf_level`: confidence level selected by the user.
3. `conf_interval`: confidence interval for the population mean.
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:
Perform an exact t-test with unknown variance. The null hypothesis is \( H_0 : \text{mean} = 50 \) against the one sided alternative \( H_1 : \text{mean} < 50 \); according to the results, the p-value is too great, there are no evidence for rejecting \( H_0 \).

```
(%i1) load("stats")$
(%i2) data: [78,64,35,45,45,75,43,74,42,42]$
(%i3) test_mean(data,'conflevel=0.9,'alternative='less,'mean=50);
```
```
<table>
<thead>
<tr>
<th>MEAN TEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean_estimate = 54.3</td>
</tr>
<tr>
<td>conf_level = 0.9</td>
</tr>
<tr>
<td>conf_interval = [minf, 61.51314273502712]</td>
</tr>
<tr>
<td>method = Exact t-test. Unknown variance.</td>
</tr>
<tr>
<td>hypotheses = H0: mean = 50 , H1: mean &lt; 50</td>
</tr>
<tr>
<td>statistic = .8244705235071678</td>
</tr>
<tr>
<td>distribution = [student_t, 9]</td>
</tr>
<tr>
<td>p_value = .7845100411786889</td>
</tr>
</tbody>
</table>
```

This time Maxima performs an asymptotic test, based on the Central Limit Theorem. The null hypothesis is \( H_0 : \text{equal(mean,50)} \) against the two sided alternative \( H_1 : \text{notequal(mean,50)} \); according to the results, the p-value is very small, \( H_0 \) should be rejected in favor of the alternative \( H_1 \). Note that, as indicated by the Method component, this procedure should be applied to large samples.

```
(%i1) load("stats")$
(%i2) test_mean([36,118,52,87,35,256,56,178,57,57,89,34,25,98,35,
              98,41,45,198,54,79,63,35,45,44,75,42,75,45,45,
              45,51,123,54,151],
              'asymptotic=true,'mean=50);
```
**Mean Test**

| mean_estimate = 74.88571428571429  |
| conf_level = 0.95               |
| conf_interval = [57.72848600856194, 92.04294256286663] |

(%o2) method = Large sample z-test. Unknown variance.

| hypotheses = H0: mean = 50 , H1: mean # 50  |
| statistic = 2.842831192874313              |
| distribution = [normal, 0, 1]             |
| p_value = .004471474652002261            |

**Function**

test_means_difference (x1, x2)

test_means_difference (x1, x2, options ...)

This is the difference of means t-test for two samples. Arguments x1 and x2 are lists or column matrices containing two independent samples. In case of different unknown variances (see options 'dev1, 'dev2 and 'varequal bellow), the degrees of freedom are computed by means of the Welch approximation. It also performs an asymptotic test based on the Central Limit Theorem if option 'asymptotic is set to true.

Options:

- 'alternative, default 'twosided, is the alternative hypothesis; valid values are: 'twosided, 'greater and 'less.
- 'dev1, default 'unknown, this is the value of the standard deviation of the x1 sample when it is known; valid values are: 'unknown or a positive expression.
- 'dev2, default 'unknown, this is the value of the standard deviation of the x2 sample when it is known; valid values are: 'unknown or a positive expression.
- 'varequal, default false, whether variances should be considered to be equal or not; this option takes effect only when 'dev1 and/or 'dev2 are 'unknown.
- 'conflevel, default 95/100, confidence level for the confidence interval; it must be an expression which takes a value in (0,1).
- 'asymptotic, default false, indicates whether it performs an exact t-test or an asymptotic one based on the Central Limit Theorem; valid values are true and false.

The output of function test_means_difference is an inference_result Maxima object showing the following results:

1. 'diff_estimate: the difference of means estimate.
2. 'conf_level: confidence level selected by the user.
3. 'conf_interval: confidence interval for the difference of means.
4. 'method: inference procedure.
5. 'hypotheses: null and alternative hypotheses to be tested.
6. 'statistic: value of the sample statistic used for testing the null hypothesis.
7. 'distribution: distribution of the sample statistic, together with its parameter(s).
8. 'p_value: p-value of the test.

Examples:
The equality of means is tested with two small samples \( x \) and \( y \), against the alternative \( H_1: m_1 > m_2 \), being \( m_1 \) and \( m_2 \) the populations means; variances are unknown and supposed to be different.

```plaintext
(%i1) load("stats")$
(%i2) x: [20.4,62.5,61.3,44.2,11.1,23.7]$
(%i3) y: [1.2,6.9,38.7,20.4,17.2]$
(%i4) test_means_difference(x,y,'alternative='greater);
```

```
DIFFERENCE OF MEANS TEST

| diff_estimate = 20.31999999999999 |
| conf_level = 0.95               |
| conf_interval = [- .04597417812882298, inf] |
```

```
(%o4) | method = Exact t-test. Welch approx. |
| hypotheses = H0: mean1 = mean2 , H1: mean1 > mean2 |
| statistic = 1.838004300728477 |
| distribution = [student_t, 8.62758740184604] |
| p_value = .05032746527991905 |
```

The same test as before, but now variances are supposed to be equal.

```plaintext
(%i1) load("stats")$
(%i2) x: [20.4,62.5,61.3,44.2,11.1,23.7]$
(%i3) y: matrix([1.2],[6.9],[38.7],[20.4],[17.2])$
(%i4) test_means_difference(x,y,'alternative='greater,
               'varequal=true);
```

```
DIFFERENCE OF MEANS TEST

| diff_estimate = 20.31999999999999 |
| conf_level = 0.95               |
| conf_interval = [- .7722627696897568, inf] |
```

```
(%o4) | method = Exact t-test. Unknown equal variances |
```
Chapter 80: stats

| hypotheses = H0: mean1 = mean2 , H1: mean1 > mean2 |
| statistic = 1.765996124515009 |
| distribution = [student_t, 9] |
| p_value = .05560320992529344 |

test_variance (x)

Function

test_variance (x, options, ...)

Function

This is the variance chi^2-test. Argument x is a list or a column matrix containing 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.
- 'conflevel, default 95/100, confidence level for the confidence interval; it must be an expression which takes a value in (0,1).

The output of function test_variance is an inference_result Maxima object showing the following results:
1. 'var_estimate: the sample variance.
2. 'conf_level: confidence level selected by the user.
3. 'conf_interval: confidence interval for the population variance.
4. 'method: inference procedure.
5. 'hypotheses: null and alternative hypotheses to be tested.
6. 'statistic: value of the sample statistic used for testing the null hypothesis.
7. 'distribution: distribution of the sample statistic, together with its parameter.
8. 'p_value: p-value of the test.

Examples:
It is tested whether the variance of a population with unknown mean is equal to or greater than 200.

(%i1) load("stats")$
(%i2) x: [203,229,215,220,223,233,208,228,209]$
(%i3) test_variance(x,'alternative='greater,'variance=200);  

VARIANCE TEST

| var_estimate = 110.75
| conf_level = 0.95
| conf_interval = [57.13433376937479, inf]
\begin{verbatim}
(%o3) method = Variance Chi-square test. Unknown mean.
   hypotheses = H0: var = 200 , H1: var > 200
   statistic = 4.43
   distribution = [chi2, 8]
   p_value = .8163948512777689

\end{verbatim}

\textbf{Function test_variance_ratio (x1, x2)}

This is the variance ratio \( F \)-test for two normal populations. Arguments \( x_1 \) and \( x_2 \) 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 \( x_1 \) was taken.
- 'mean2, default 'unknown, when it is known, this is the mean of the population from which \( x_2 \) was taken.
- 'conflevel, default 95/100, confidence level for the confidence interval of the ratio; it must be an expression which takes a value in \((0,1)\).

The output of function \texttt{test_variance_ratio} is an \texttt{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.

\begin{verbatim}
(%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
\end{verbatim}
Chapter 80: stats

| 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 |

Function test_proportion (x, n)
Function 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);

<table>
<thead>
<tr>
<th>PROPORTION TEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample_proportion = .4368932038834951</td>
</tr>
<tr>
<td>conf_level = 0.95</td>
</tr>
<tr>
<td>conf_interval = [0, 0.522714149150231]</td>
</tr>
</tbody>
</table>

(%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);

<table>
<thead>
<tr>
<th>PROPORTION TEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample_proportion = .43689</td>
</tr>
<tr>
<td>conf_level = 0.99</td>
</tr>
<tr>
<td>conf_interval = [.31422, .56749]</td>
</tr>
</tbody>
</table>

(%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 (x1, n1, x2, n2) Function

test_proportions_difference (x1, n1, x2, n2, options ...) Function

Inferences on the difference of two proportions. Argument x1 is the number of successes in n1 trials in a Bernoulli experiment in the first population, and x2 and n2 are the corresponding values in the second population. Samples are independent and the test is asymptotic.
Chapter 80: stats

Options:
- 'alternative', default 'twosided, is the alternative hypothesis; valid values are:
  'twosided (p1 ≠ p2), 'greater (p1 > p2) and 'less (p1 < p2).
- 'conflevel', default 95/100, confidence level for the confidence interval; it must
  be an expression which takes a value in (0,1).
- 'correct', default true, indicates whether Yates correction is applied or not.

The output of function test_proportions_difference is an inference_result
Maxima object showing the following results:
1. 'proportions: list with the two sample proportions.
2. 'conf_level: confidence level selected by the user.
3. 'conf_interval: Confidence interval for the difference of proportions p1 - p2.
4. 'method: inference procedure and warning message in case of any of the samples
   sizes is less than 10.
5. 'hypotheses: null and alternative hypotheses to be tested.
6. 'statistic: value of the sample statistic used for testing the null hypothesis.
7. 'distribution: distribution of the sample statistic, together with its parameters.
8. 'p_value: p-value of the test.

Examples:
A machine produced 10 defective articles in a batch of 250. After some maintenance
work, it produces 4 defective in a batch of 150. In order to know if the machine has
improved, we test the null hypothesis \( H_0: p_1 = p_2 \), 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);
```

DIFERENCE 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.

\[
\frac{1}{n_2 + n_1} \sqrt{\frac{1}{n_2} \frac{n_2}{x_2 + x_1} + \frac{1}{n_1} \frac{n_1}{x_1 + x_2}} (1 - \frac{1}{\frac{n_2}{x_2 + x_1} + \frac{n_1}{x_1 + x_2}})
\]

Function test_sign (x)

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 \( H_1 : \text{median} > 6 \).

\[
\begin{align*}
\text{(\%i1) } & \text{load("stats")} \\
\text{(\%i2) } & \text{x: [2,0.1,7,1.8,4,2.3,5.6,7.4,5.1,6.1,6]} \\
\text{(\%i3) } & \text{test_sign(x,'median=6,'alternative='greater);}
\end{align*}
\]

SIGN TEST
med_estimate = 5.1
method = Non parametric sign test.

\[
\text{\%o3) } \text{hypotheses = H0: median = 6 , H1: median > 6}
\]
Chapter 80: stats

| statistic = 7 |
| distribution = [binomial, 10, 0.5] |
| p_value = .0546874999999989 |

Function

test_signed_rank (x)

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 |
| hypotheses = H0: med = 15 , H1: med > 15 |
| statistic = 14 |
| distribution = [signed_rank, 6] |
```
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  
  hypotheses = H0: med = 2.5 , H1: med # 2.5  
      statistic = 76.5  
      distribution = [normal, 60.5, 17.58195097251724]  
      p_value = .3628097734643669
```

**test_rank_sum** \((x_1, x_2)\)

Function

This is the Wilcoxon-Mann-Whitney test for comparing the medians of two continuous populations. The first two arguments \(x_1\) and \(x_2\) 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.

Options:
- `alternative`, default `twosided`, is the alternative hypothesis; valid values are: `twosided`, `greater` and `less`.

The output of function **test_rank_sum** is an **inference_result** Maxima object with the following results:

1. `method`: inference procedure.
2. `hypotheses`: null and alternative hypotheses to be tested.
3. `statistic`: value of the sample statistic used for testing the null hypothesis.
4. `distribution`: distribution of the sample statistic, together with its parameters.
5. `p_value`: p-value of the test.

Examples:

Checks whether populations have similar medians. Samples sizes are small and an exact test is made.

```
(%i1) load("stats")$
(%i2) x:[12,15,17,38,42,10,23,35,28]$  
(%i3) y:[21,18,25,14,52,65,40,43]$ 
```
Chapter 80: stats

%i4) test_rank_sum(x,y);

| RANK SUM TEST |
| method = Exact test |
| hypotheses = H0: med1 = med2 , H1: med1 ≠ med2 |

(%o4) statistic = 22
| distribution = [rank_sum, 9, 8] |
| p_value = .1995886466474702 |

Now, with greater samples and ties, the procedure makes normal approximation. The alternative hypothesis is $H_1: \text{median}_1 < \text{median}_2$.

(%i1) load("stats")$
 (%i2) x: [39,42,35,13,10,23,15,20,17,27]$
 (%i3) y: [20,52,66,19,41,32,44,25,14,39,43,35,19,56,27,15]$
 (%i4) test_rank_sum(x,y,'alternative='less);

| RANK SUM TEST |
| method = Asymptotic test. Ties |
| hypotheses = H0: med1 = med2 , H1: med1 < med2 |

(%o4) statistic = 48.5
| distribution = [normal, 79.5, 18.95419580097078] |
| p_value = .05096985666598441 |

test_normality (x) Function

Shapiro-Wilk test for normality. Argument x is a list of numbers, and sample size must be greater than 2 and less or equal than 5000, otherwise, function test_normality signals an error message.

Reference:

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 |
\texttt{simple\_linear\_regression} \ (x) \quad \text{Function}

\texttt{simple\_linear\_regression} \ (x \ \text{option}) \quad \text{Function}

Simple linear regression, \( y_i = a + bx_i + e_i \), where \( e_i \) are \( N(0, \sigma) \) independent random variables. Argument \( x \) must be a two column matrix or a list of pairs.

Options:

- \'conflevel, default \( 95/100 \), confidence level for the confidence interval; it must be an expression which takes a value in \((0,1)\).
- \'regressor, default \'x, name of the independent variable.

The output of function \texttt{simple\_linear\_regression} is an \texttt{inference\_result} Maxima object with the following results:

1. \'model: the fitted equation. Useful to make new predictions. See examples bellow.
2. \'means: bivariate mean.
3. \'variances: variances of both variables.
4. \'correlation: correlation coefficient.
5. \'adc: adjusted determination coefficient.
6. \'a\_estimation: estimation of parameter \( a \).
7. \'a\_conf\_int: confidence interval of parameter \( a \).
8. \'b\_estimation: estimation of parameter \( b \).
9. \'b\_conf\_int: confidence interval of parameter \( b \).
10. \'hypotheses: null and alternative hypotheses about parameter \( b \).
11. \'statistic: value of the sample statistic used for testing the null hypothesis.
12. \'distribution: distribution of the sample statistic, together with its parameter.
13. \'p\_value: \( p \)-value of the test about \( b \).
14. \'v\_estimation: unbiased variance estimation, or residual variance.
15. \'v\_conf\_int: variance confidence interval.
16. \'cond\_mean\_conf\_int: confidence interval for the conditioned mean. See examples bellow.
17. \'new\_pred\_conf\_int: confidence interval for a new prediction. See examples bellow.
18. \'residuals: list of pairs (prediction, residual), ordered with respect to predictions. This is useful for goodness of fit analysis. See examples bellow.

Only items 1, 4, 14, 9, 10, 11, 12, and 13 above, in this order, are shown by default. The rest remain hidden until the user makes use of functions \texttt{items\_inference} and \texttt{take\_inference}.

Example:
Fitting a linear model to a bivariate sample. Input %i4 plots the sample together with the regression line; input %i5 computes \( y \) given \( x=113 \); the means and the confidence interval for a new prediction when \( x=113 \) are also calculated.

\[
\begin{align*}
\text{(%i1)} & \quad \text{load("stats")}$
\text{(%i2)} & \quad s:=[[125,140.7], [130,155.1], [135,160.3], [140,167.2], [145,169.8]]$
\text{(%i3)} & \quad z:simple_linear_regression(s,conflevel=0.99);
& \quad \text{SIMPLE LINEAR REGRESSION}
& \quad \text{model} = 1.405999999999985 x - 31.18999999999804
& \quad \text{correlation} = .9611685255255155
& \quad \text{v_estimation} = 13.57966666666665
\text{(%o3)} & \quad b\_conf\_int = [.04469633662525263, 2.767303663374718]
& \quad \text{hypotheses} = H0: b = 0 , H1: b \neq 0
& \quad \text{statistic} = 6.032686683658114
& \quad \text{distribution} = [\text{student}_t, 3]
& \quad \text{p_value} = 0.0038059549413203
\text{(%i4)} & \quad \text{plot2d([[discrete, s], take_inference(model,z)],}
& \quad \text{[x,120,150],}
& \quad \text{gnuplot_curve_styles, ["with points","with lines"]])$
\text{(%i5)} & \quad \text{take_inference(model,z), x=133;}
\text{(%o5)} & \quad 155.808
\text{(%i6)} & \quad \text{take_inference(means,z);}
\text{(%o6)} & \quad [135.0, 158.62]
\text{(%i7)} & \quad \text{take_inference(new_pred_conf_int,z), x=133;}
\text{(%o7)} & \quad [132.0728595995113, 179.543140404887]
\end{align*}
\]

80.4 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 \text{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 \text{test_signed_rank}.
pdf_rank_sum (x, n, m)
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)
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.
71 stirling

71.1 Functions and Variables for stirling

stirling \((z,n)\)  

Function  

stirling \((z,n,pred)\)  

Replace \(\text{gamma}(x)\) with the \(O(1/x(2n - 1))\) Stirling formula. when \(n\) isn't a nonnegative integer, signal an error. With the optional third argument \(\text{pred}\), the Stirling formula is applied only when \(\text{pred}\) is true.


Examples:

\[(%i1)\) load (stirling)$

\[(%i2)\) stirling(gamma(%alpha+x)/gamma(x),1);

\[(%o2)\] \(1/2 - x\) \(x + %alpha - 1/2\)

\[(%i3)\) taylor(%,x,inf,1);

\[(%o3)\] \(\%alpha 2 %alpha\)

\[(%i4)\) map('factor,%);

\[(%o4)\] \(\%alpha - 1\)

The function \text{stirling} knows the difference between the variable 'gamma' and the function \text{gamma}:

\[(%i5)\) stirling(gamma + gamma(x),0);

\[(%o5)\] \(\text{gamma} + \text{sqrt}(2) \text{sqrt}(\%pi) x \ %e\)

\[(%i6)\) stirling(gamma(y) + gamma(x),0);

\[(%o6)\] \(\text{sqrt}(2) \text{sqrt}(\%pi) y \ %e\)

\[(%i7)\) makegamma(pochhammer(a,k)/pochhammer(b,k));

\[(%o7)\] \(\text{(gamma}(b)\text{*gamma}(k+a))/(\text{gamma}(a)\text{*gamma}(k+b))\)

\[(%i8)\) stirling(%i,1, lambda([s], not(freeof(k,s))));
(%o8) $\frac{e^{(b-a)} \cdot \text{gamma}(b) \cdot (k+a)^{(k+a-1/2)} \cdot (k+b)^{(-k-b+1/2)}}{\text{gamma}(a)}$

The terms $\text{gamma}(a)$ and $\text{gamma}(b)$ are free of $k$, so the Stirling formula was not applied to these two terms.

To use this function write first `load("stirling").`
82 stringproc

82.1 Introduction to string processing

`stringproc.lisp` enlarges Maxima's capabilities of working with strings and adds some useful functions for file input/output.

For questions and bugs please mail to van.nek at arcor.de.

In Maxima a string is easily constructed by typing "text". `stringp` tests for strings.

```lisp
(%i1) m: "text";
(%o1) text
(%i2) stringp(m);
(%o2) true
```

Characters are represented as strings of length 1. These are not Lisp characters. Tests can be done with `charp` (respectively `lcharp` and conversion from Lisp to Maxima characters with `cunlisp`).

```lisp
(%i1) c: "e";
(%o1) e
(%i2) [charp(c),lcharp(c)];
(%o2) [true, false]
(%i3) supcase(c);
(%o3) E
(%i4) charp(%);
(%o4) true
```

All functions in `stringproc.lisp` that return characters, return Maxima-characters. Due to the fact, that the introduced characters are strings of length 1, you can use a lot of string functions also for characters. As seen, `supcase` is one example.

It is important to know, that the first character in a Maxima-string is at position 1. This is designed due to the fact that the first element in a Maxima-list is at position 1 too. See definitions of `charat` and `charlist` for examples.

In applications string-functions are often used when working with files. You will find some useful stream- and print-functions in `stringproc.lisp`. The following example shows some of the here introduced functions at work.

Example:

`openw` returns an output stream to a file, `printf` then allows formatted writing to this file. See `printf` for details.

```lisp
(%i1) s: openw("E:/file.txt");
(%o1) #<output stream E:/file.txt>
(%i2) for n:0 thru 10 do printf( s, "~d ", fib(n) ) ;
(%o2) done
(%i3) printf( s, "~%~d ~f ~a ~a ~f ~e ~a~%",
42, 1.234, sqrt(2), %pi, 1.0e-2, 1.0e-2, 1.0b-2 ) ;
(%o3) false
(%i4) close(s);
(%o4) true
```
After closing the stream you can open it again, this time with input direction. `readline` returns the entire line as one string. The `stringproc` package now offers a lot of functions for manipulating strings. Tokenizing can be done by `split` or `tokens`.

```
(%i5) s: openr("E:/file.txt");
(%o5) #<input stream E:/file.txt>
(%i6) readline(s);
  0 1 1 2 3 5 8 13 21 34 55
(%i7) line: readline(s);
(%o7) 42 1.234 sqrt(2) %pi 0.01 1.0E-2 1.0b-2
(%i8) list: tokens(line);
(%o8) [42, 1.234, sqrt(2), %pi, 0.01, 1.0E-2, 1.0b-2]
(%i9) map( parse_string, list );
(%o9) [42, 1.234, 1.414213562373095, 3.141592653589793, 0.01, 0.01, 0.01]
(%i10) readline(s);
(%o10) false
(%i11) close(s)$
```

`readline` returns `false` when the end of file occurs.

### 82.2 Functions and Variables for input and output

Example:

```
(%i1) s: openw("E:/file.txt");
(%o1) #<output stream E:/file.txt>
(%i2) control:
"¬2tAn atom: ¬20t¬a¬2tand a list: ¬20t¬{¬r ¬}¬2t
and an integer: ¬20t¬d¬2t"$
(%i3) printf( s,control, 'true,[1,2,3],42 )$
(%o3) false
(%i4) close(s);
(%o4) true
(%i5) s: openr("E:/file.txt");
(%o5) #<input stream E:/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*)

Closes *stream* and returns `true` if *stream* had been open.

**flength** (*stream*)

Returns the number of elements in *stream*. 
**fposition** (*stream*)

Function

**fposition** (*stream, pos*)  
Returns the current position in *stream*, if *pos* is not used. If *pos* is used, **fposition** sets the position in *stream*. *pos* has to be a positive number, the first element in *stream* is in position 1.

**freshline** ()

Function

**freshline** (*stream*)  
Writes a new line (to *stream*), if the position is not at the beginning of a line. See also **newline**.

**newline** ()

Function

**newline** (*stream*)  
Writes a new line (to *stream*). See **sprint** for an example of using **newline()**. Note that there are some cases, where **newline()** does not work as expected.

**opena** (*file*)

Function

Returns an output stream to *file*. If an existing file is opened, **opena** appends elements at the end of file.

**openr** (*file*)

Function

Returns an input stream to *file*. If *file* does not exist, it will be created.

**openw** (*file*)

Function

Returns an output stream to *file*. If *file* does not exist, it will be created. If an existing file is opened, **openw** destructively modifies *file*.

**printf** (*dest, string*)

Function

**printf** (*dest, string, expr_1, ..., expr_n*)  
Makes the Common Lisp function FORMAT available in Maxima. (From gcl.info: "format produces formatted output by outputting the characters of control-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 args to create their output."")

The following description and the examples may give an idea of using **printf**. See a Lisp reference for more information.

- `%n`  
  new line
- `%k`  
  fresh line
- `%t`  
  tab
- `%$`  
  monetary
- `%d`  
  decimal integer
- `%b`  
  binary integer
- `%o`  
  octal integer
- `%x`  
  hexadecimal integer
- `%br`  
  base-b integer
\[\text{\textasciitilde}r\] spell an integer
\[\text{\textasciitilde}p\] plural
\[\text{\textasciitilde}f\] floating point
\[\text{\textasciitilde}e\] scientific notation
\[\text{\textasciitilde}g\] \[\text{\textasciitilde}f\] or \[\text{\textasciitilde}e\], depending upon magnitude
\[\text{\textasciitilde}h\] bigfloat
\[\text{\textasciitilde}a\] uses Maxima function \text{string}
\[\text{\textasciitilde}s\] like \[\text{\textasciitilde}a\], but output enclosed in "double quotes"
-- -
\[\text{\textasciitilde}<\] justification, \[\text{\textasciitilde}>\] terminates
\[\text{\textasciitilde}(\] case conversion, \[\text{\textasciitilde)}\] terminates
\[\text{\textasciitilde}[\] selection, \[\text{\textasciitilde}]\] terminates
\[\text{\textasciitilde}\{\] iteration, \[\text{\textasciitilde}\}\] terminates

Note that the selection directive \[\text{\textasciitilde}[\] is zero-indexed. Also note that the directive \[\text{\textasciitilde}*\] is not supported.

\begin{verbatim}
(%i1) printf( false, "\text{\textasciitilde}a \text{\textasciitilde}a \text{\textasciitilde}4f \text{\textasciitilde}a \text{\textasciitilde}@r",
            "String",sym,bound,sqrt(12),144), bound = 1.234;
(%o1) String sym 1.23 2*sqrt(3) CXLIV
(%i2) printf( false,"\text{\textasciitilde}a ~\{\text{\textasciitilde}\}~",["one",2,"THREE"] );
(%o2) one 2 THREE
(%i3) printf(true,"\text{\textasciitilde}a \text{\textasciitilde}a \text{\textasciitilde}9,1f \text{\textasciitilde}a \text{\textasciitilde}9%",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: "\text{\textasciitilde}a ~\{\text{\textasciitilde}\}~", bird\text{\textasciitilde}p ~\[\text{\textasciitilde};\text{\textasciitilde}are~\] singing.$$
(%i5) printf( false,control, n,n,if n=1 then 0 else 1 ), n=2;
(%o5) Two birds are singing.
\end{verbatim}

If \text{dest} is a stream or \text{true}, then \text{printf} returns \text{false}. Otherwise, \text{printf} returns a string containing the output.

\textbf{readline (stream)}

Function

Returns a string containing the characters from the current position in \text{stream} up to the end of the line or \text{false} if the end of the file is encountered.

\textbf{sprint (expr.1, ..., expr.n)}

Function

Evaluates and displays its arguments one after the other 'on a line' starting at the leftmost position. The numbers are printed with the '-' right next to the number, and it disregards line length. \text{newline()}, which will be autoloaded from \text{stringproc.lisp} might be useful, if you whish to place intermediate line breaking.

\begin{verbatim}
(%i1) for n:0 thru 20 do sprint( fib(n) )$
  0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765$
(%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
\end{verbatim}
82.3 Functions and Variables for characters

**alphacharp** *(char)*

Returns **true** if *char* is an alphabetic character.

**alphanumericp** *(char)*

Returns **true** if *char* is an alphabetic character or a digit.

**ascii** *(int)*

Returns the character corresponding to the ASCII number *int*. (%i1) for n from 0 thru 255 do (tmp: ascii(n), if alphacharp(tmp) then sprint(tmp), 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)*

Returns **true** if *char_1* and *char_2* are the same.

**cequalignore** *(char_1, char_2)*

Like **cequal** but ignores case.

**cgreaterp** *(char_1, char_2)*

Returns **true** if the ASCII number of *char_1* is greater than the number of *char_2*.

**cgreaterpignore** *(char_1, char_2)*

Like **cgreaterp** but ignores case.

**charp** *(obj)*

Returns **true** if *obj* is a Maxima-character. See introduction for example.

**cint** *(char)*

Returns the ASCII number of *char*.

**clessp** *(char_1, char_2)*

Returns **true** if the ASCII number of *char_1* is less than the number of *char_2*.

**clesspignore** *(char_1, char_2)*

Like **clessp** but ignores case.

**constituent** *(char)*

Returns **true** if *char* is a graphic character and not the space character. A graphic character is a character one can see, plus the space character. (**constituent** is defined by Paul Graham, ANSI Common Lisp, 1996, page 67.)
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 { | } ~

cunlisp (lisp, char)
   Converts a Lisp-character into a Maxima-character. (You won’t need it.)

digitcharp (char)
   Returns true if char is a digit.

lcharp (obj)
   Returns true if obj is a Lisp-character. (You won’t need it.)

lowercasep (char)
   Returns true if char is a lowercase character.

newline
   The newline character.

space
   The space character.

tab
   The tab character.

uppercasep (char)
   Returns true if char is an uppercase character.

stringp (obj)
   Returns true if obj is a string. See introduction for example.

charat (string, n)
   Returns the n-th character of string. The first character in string is returned with n = 1.

   (%i1) charat("Lisp",1);
   (%o1) L

charlist (string)
   Returns the list of all characters in string.

   (%i1) charlist("Lisp");
   (%o1) [L, i, s, p]
   (%i2) %[1];
   (%o2) L
**Function eval_string (str)**

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`.

**Function parse_string (str)**

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 : foo + baz)

See also `eval_string`.

**Function scopy (string)**

Returns a copy of `string` as a new string.

**Function sdowncase (string)**

**Function sdowncase (string, start)**

**Function sdowncase (string, start, end)**

Like `supcase`, but uppercase characters are converted to lowercase.

**Function sequal (string_1, string_2)**

Returns `true` if `string_1` and `string_2` are the same length and contain the same characters.

**Function sequalignore (string_1, string_2)**

Like `sequal` but ignores case.

**Function sexplode (string)**

`sexplode` is an alias for function `charlist`.
simplode (list) Function
simplode (list, delim) Function

simplode takes a list of expressions and concatenates them into a string. If no
delimiter `delim` is specified, `simplode` uses no delimiter. `delim` can be any string.

(%i1) simplode(["xx[",3,"]:",expand((x+y)^3)]);
(%)1  xx[3]:y^3+3*x*y^2+3*x^2*y+x^3
(%i2) simplode( sexplode("stars")," * " );
(%)2  s * t * a * r * s
(%i3) simplode( ["One","more","coffee."]," " );
(%)3  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.

(%i1) s: "A submarine."
(%i2) concat( substring(s,1,3),"yellow ",substring(s,3) );
(%)2  A yellow submarine.
(%i3) sinsert("hollow ",s,3);
(%)3  A hollow submarine.

sinvertcase (string) Function
sinvertcase (string, start) Function
sinvertcase (string, start, end) Function

Returns string except that each character from position start to end is inverted. If
end is not given, all characters from start to the end of string are replaced.

(%i1) sinvertcase("sInvertCase");
(%)1  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.

(%i1) smake(3,"w");
(%)1  www

smismatch (string_1, string_2) Function
smismatch (string_1, string_2, test) Function

Returns the position of the first character of string_1 at which string_1 and string_2
differ or false. Default test function for matching is sequal. If smismatch should
ignore case, use sequalignore as test.

(%i1) smismatch("seven","seventh");
(%)1  6
**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.

```lisp
(%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)

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.

**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.

```lisp
(%i1) sremove("n't","I don't like coffee.");
(%o1) I do like coffee.
(%i2) sremove ("DO ",%,'sequalignore);
(%o2) I like coffee.
```

**sremovefirst** (seq, string)

**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)

Returns a string with all the characters of string in reverse order.

**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.
ssort (string)  Function

ssort (string, test)  Function
Returns a string that contains all characters from string in an order such there are no two successive characters c and d such that test (c, d) is false and test (d, c) is true. Default test function for sorting is clessp. The set of test functions is {clessp, clesspignore, cgreaterp, cgreaterpignore, cequal, cequalignore}.

(%i1) ssort("I don’t like Mondays.");
(%o1) `.IMaddeiklnnoosty
(%i2) ssort("I don’t like Mondays.",’cgreaterpignore);
(%o2) ytscoonMlkliedda.’

ssubst (new, old, string)  Function
ssubst (new, old, string, test)  Function
ssubst (new, old, string, test, start)  Function
ssubst (new, old, string, test, start, end)  Function
Returns a string like string except that all substrings matching old are replaced by new. old and new need not to be of the same length. Default test function for matching is sequal. If subst 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.

(%i1) ssubst("like","hate","I hate Thai food. I hate green tea.");
(%o1) I like Thai food. I like green tea.
(%i2) ssubst("Indian","thai",’,’sequalignore,8,12);
(%o2) I like Indian food. I like green tea.

ssubstfirst (new, old, string)  Function
ssubstfirst (new, old, string, test)  Function
ssubstfirst (new, old, string, test, start)  Function
ssubstfirst (new, old, string, test, start, end)  Function
Like subst except that only the first substring that matches old is replaced.

strim (seq,string)  Function
Returns a string like string, but with all characters that appear in seq removed from both ends.

(%i1) "/* comment */"$
(%i2) strim(" /*",’%);
(%o2) comment
(%i3) slength(%);
(%o3) 7

striml (seq, string)  Function
Like strim except that only the left end of string is trimmed.
strimr (seq, string)
Like strim except that only the right end of string is trimmed.

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.

```
(%i1) substring("substring",4);
(%o1) string
(%i2) substring(%,4,6);
(%o2) in
```

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.

```
(%i1) supcase("english",1,2);
(%o1) English
```

tokens (string)
tokens (string, test)
Returns a list of tokens, which have been extracted from string. The tokens are sub-strings whose characters satisfy a certain test function. If test is not given, constituent is used as the default test. \{constituent, alphacharp, digitcharp, lowercasep, uppercasep, charp, characterp, alphanumericp\} is the set of test functions. (The Lisp-version of tokens is written by Paul Graham. ANSI Common Lisp, 1996, page 67.)

```
(%i1) tokens("24 October 2005");
(%o1) [24, October, 2005]
(%i2) tokens("05-10-24",'digitcharp);
(%o2) [05, 10, 24]
(%i3) map(parse_string,%);
(%o3) [5, 10, 24]
```
83 unit

83.1 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:

```maxima
(%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 TOLEVEL-MACSYMA-EVAL ...
WARNING: DEFUN/DEFMACRO: redefining function MSETCK ...
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.

83.2 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.

\begin{verbatim}
(%i2) N;
kg m
====
2
s
(%i3) dyn;
1
--- kg m
100000 2
s
(%i4) g;
1
---- (kg)
1000
(%i5) centigram*inch/minutes^2;
127 kg m
--------------
1800000000000 2
s
\end{verbatim}

In some cases this is the desired behavior. If the user wishes to use other units, this is achieved with the setunits command:

\begin{verbatim}
(%i6) setunits([centigram,inch,minutes]);
done
(%i7) N;
1800000000000 %in cg
------------
127 2
%min
\end{verbatim}
Chapter 83: unit

(%i8) dyn;
18000000 %in cg
(%o8) (--------) (-----)
127 2
%min

(%i9) g;
(%o9) (100) (cg)

(%i10) centigram*inch/minutes^2;
%in cg
(%o10) -------
2
%min

The setting of units is quite flexible. For example, if we want to get back to kilograms, meters, and seconds as defaults for those dimensions we can do:

(%i11) setunits([kg,m,s]);
(%o11) done

(%i12) centigram*inch/minutes^2;
127 kg m
(%o12) (-------------) (----)
1800000000000 2
s

Derived units are also handled by this command:

(%i17) setunits(N);
(%o17) done

(%i18) N;
(%o18) N

(%i19) dyn;
1
(%o19) (------) (N)
100000

(%i20) kg*m/s^2;
(%o20) N

(%i21) centigram*inch/minutes^2;
127
(%o21) (-------------) (N)
1800000000000

Notice that the unit package recognized the non MKS combination of mass, length, and inverse time squared as a force, and converted it to Newtons. This is how Maxima works in general. If, for example, we prefer dyne to Newtons, we simply do the following:

(%i22) setunits(dyn);
(%o22) done

(%i23) kg*m/s^2;
(%o23) (100000) (dyn)
To discontinue simplifying to any force, we use the uforget command:

```
(%i26) uforget(dyn);
(%o26) false
(%i27) kg*m/s^2;
   kg m
---
 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*)

By default, the unit package converts all units to the seven fundamental dimensions using MKS units. This behavior can be changed with the setunits command. After that, the user can restore the default behavior for a particular dimension by means of the uforget command:

```
(%i13) setunits([centigram,inch,minute]);
(%o13) done
(%i14) centigram*inch/minutes^2;
   %in cg
---
 2
%min
(%i15) uforget([cg,%in,%min]);
(%o15) [false, false, false]
(%i16) centigram*inch/minutes^2;
   127 kg m
----------- ----
18000000000000 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*)

When resetting the global environment is overkill, there is the convert command, which allows one time conversions. It can accept either a single argument or a list
of units to use in conversion. When a convert operation is done, the normal global evaluation system is bypassed, in order to avoid the desired result being converted again. As a consequence, for inexact calculations "rat" warnings will be visible if the global environment controlling this behavior (ratprint) is true. This is also useful for spot-checking the accuracy of a global conversion. Another feature is convert will allow a user to do Base Dimension conversions even if the global environment is set to simplify to a Derived Dimension.

(%i2) kg*m/s^2;
    kg m ----
     2

(%i3) convert(kg*m/s^2,[g,km,s]);
    g km ----
     2

(%i4) convert(kg*m/s^2,[g,inch,minute]);

'rat' replaced 39.37007874015748 by 5000/127 = 39.37007874015748
18000000000 %in g

(%i5) convert(kg*m/s^2,[N]);

(%i6) convert(kg*m^2/s^2,[N]);

(%i7) setunits([N,J]);

(%i8) convert(kg*m^2/s^2,[N]);

(%i9) convert(kg*m^2/s^2,[N,inch]);

'rat' replaced 39.37007874015748 by 5000/127 = 39.37007874015748
5000

(%i10) convert(kg*m^2/s^2,[J]);

(%i11) kg*m^2/s^2;

(%i12) setunits([g,inch,s]);

(%i13) kg*m/s^2;

(%i14) uforget(N);

(%o2) N

(%o6) m N

(%o7) done

(%o8) m N

(%o9) done

(%o10) J

(%o11) J

(%o12) done

(%o13) N

(%o14) false
See also `setunits` and `uforget`. To use this function write first `load("unit")`.

**usersetunits**

Optional variable

Default value: none

If a user wishes to have a default unit behavior other than that described, they can make use of `maxima-init.mac` and the `usersetunits` variable. The `unit` package will check on startup to see if this variable has been assigned a list. If it has, it will use setunits on that list and take the units from that list to be defaults. `uforget` will revert to the behavior defined by usersetunits over its own defaults. For example, if we have a `maxima-init.mac` file containing:

```maxima
usersetunits : [N,J];
```

we would see the following behavior:

```maxima
(%i1) load("unit")$
```

Redefining necessary functions...

WARNING: DEFUN/DEFMACRO: redefining function TOLEVEL-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.

```maxima
(%i2) kg*m/s^2;

(%o2) N

(%i3) kg*m^2/s^2;

(%o3) J
```
Without `usersetunits`, the initial inputs would have been converted to MKS, and `uforget` would have resulted in a return to MKS rules. Instead, the user preferences are respected in both cases. Notice these can still be overridden if desired. To completely eliminate this simplification - i.e. to have the user defaults reset to factory defaults - the `dontusedimension` command can be used. `uforget` can restore user settings again, but only if `usedimension` frees it for use. Alternately, `kill(usersetunits)` will completely remove all knowledge of the user defaults from the session. Here are some examples of how these various options work.

```markdown
(%i2) kg*m/s^2;  
(%o2) N

(%i3) kg*m^2/s^2;  
(%o3) J

(%i4) setunits([dyn,eV]);  
(%o4) done

(%i5) kg*m/km/s^2;  
(%o5) (1000) (J)

(%i6) kg*m/s^2;  
(%o6) (100000) (dyn)

(%i7) kg*m^2/s^2;  
(%o7) (6241509596477042688) (eV)

(%i8) uforget([dyn,eV]);  
(%o8) [false, false]

(%i9) kg*m^3/s^2;  
(%o9) (6241509596477042688) (eV m)

(%i10) kg*m*km/s^2;  
(%o10) (6241509596477042688000) (eV)

(%i11) kg*m^3/s^2;  
(%o11) J m

(%i12) kg*m^2/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)
```
(%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) kg m ----  
2 s  
(%i13) kg*m^2/s^2;  
(%o13) kg m -----  
2 s  
(%i14) setunits([dyn,eV]);  
(%o14) done  
(%i15) kg*m/s^2;  
(%o15) kg m ----  
2 s  
(%i16) kg*m^2/s^2;  
(%o16) kg m -----  
2 s  
(%i17) uforget([dyn,eV]);  
(%o17) [false, false]  
(%i18) kg*m/s^2;  
(%o18) kg m ----  
2 s  
(%i19) kg*m^2/s^2;  
(%o19) kg m -----  
2 s  
(%i20) usedimension(N);  
Done. To have Maxima simplify to this dimension, use setunits([unit]) to select a unit.  
(%o20) true
Chapter 83: unit

(%i21) usedimension(J);
Done. To have Maxima simplify to this dimension, use setunits([unit]) to select a unit.
(%o21) true

(%i22) kg*m/s^2;
   kg  m
(%o22) ----
   2   s

(%i23) kg*m^2/s^2;
   2
   kg  m
(%o23) ----- 
   2   s

(%i24) setunits([dyn,eV]);
(%o24) done

(%i25) kg*m/s^2;
   (100000) (dyn)
(%o25)

(%i26) kg*m^2/s^2;
   (6241509596477042688) (eV)
(%o26)

(%i27) uforget([dyn,eV]);
(%o27) [false, false]

(%i28) kg*m/s^2;
   N
(%o28)

(%i29) kg*m^2/s^2;
   J
(%o29)

(%i30) kill(usersetunits);
(%o30) done

(%i31) uforget([dyn,eV]);
(%o31) [false, false]

(%i32) kg*m/s^2;
   kg  m
(%o32) ----
   2   s

(%i33) kg*m^2/s^2;
   2
   kg  m
(%o33) ----- 
   2   s

Unfortunately this wide variety of options is a little confusing at first, but once the user grows used to them they should find they have very full control over their working environment.
Function

**metricexpandall** (x)

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**.

Variable

**%unitexpand**

Default value: 2

This is the value supplied to **metricexpandall** during the initial loading of *unit*. 
84 zeilberger

84.1 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.

84.1.0.1 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

$$F_k = f_{k+1} - f_k.$$

84.1.0.2 The definite summation problem

zeilberger implements Zeilberger’s algorithm for definite hypergeometric summation. Given a proper hypergeometric term $(in n and k) F_{n,k}$ and a positive integer $d$ we want to find a $d$-th order linear recurrence with polynomial coefficients (in $n$) for $F_{n,k}$ and a rational function $R$ in $n$ and $k$ such that

$$a_0 F_{n,k} + \ldots + a_d F_{n+d}, \quad k = \Delta_K (R(n,k) F_{n,k}),$$

where $\Delta_k$ is the $k$-forward difference operator, i.e., $\Delta_k (t_k) \equiv t_{k+1} - t_k$.

84.1.1 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.
84.2 Functions and Variables for zeilberger

**AntiDifference** \((F_k, k)\)

Function

Returns the hypergeometric anti-difference of \(F_k\), if it exists. Otherwise, \texttt{AntiDifference} returns \texttt{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, \texttt{Gosper} returns \texttt{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, \texttt{GosperSum} returns \texttt{mngosper_summable}.

Examples:

\begin{verbatim}
(%i1) load (zeilberger)$
(%i2) GosperSum ((-1)^k*k / (4*k^2 - 1), k, 1, n);
Dependent equations eliminated: (1)
3 n + 1
(n - 1)(- 1)
2 1
- ------------- - -
2 4
2 (4 (n + 1) - 1)

(%i3) GosperSum (1 / (4*k^2 - 1), k, 1, n);
3
- n -
2 1
------------- + -
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)
---------- a!
(a - 1)!

(%i6) GosperSum (k*k!, k, 1, n);
Dependent equations eliminated: (1)
(n + 1)! - 1

(%i7) GosperSum ((k + 1)*k! / (k + 1)!, k, 1, n);
(n + 1) (n + 2) (n + 1)!

(%o7)
- --------------------- - 1
(n + 2)!
\end{verbatim}
Chapter 84: zeilberger

```plaintext
(%i8) GosperSum (1 / ((a - k)!*k!), k, 1, n);
(%o8) NON_GOSPER_SUMMABLE
```

Function `parGosper (F_n,k, k, n, d)`
Attempts to find a d-th order recurrence for F_{n,k}.
The algorithm yields a sequence \([s_1, s_2, ..., s_m]\) of solutions. Each solution has the form

\[
[R(n,k), [a_0, a_1, ..., a_d]].
\]

`parGosper` returns \([]\) if it fails to find a recurrence.

Function `Zeilberger (F_n,k, k, n)`
Attempts to compute the indefinite hypergeometric summation of \(F_{n,k}\).
`Zeilberger` first invokes `Gosper`, and if that fails to find a solution, then invokes `parGosper` with order 1, 2, 3, ..., up to `MAX_ORD`. If `Zeilberger` finds a solution before reaching `MAX_ORD`, it stops and returns the solution.
The algorithms yields a sequence \([s_1, s_2, ..., s_m]\) of solutions. Each solution has the form

\[
[R(n,k), [a_0, a_1, ..., a_d]].
\]

`Zeilberger` returns \([]\) if it fails to find a solution.
`Zeilberger` invokes `Gosper` only if `Gosper_in_Zeilberger` is true.

### 84.3 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.

84.4 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.
85 Indices
Appendix A Function and Variable Index

! (Operator) ......................... 38
!! (Operator) ......................... 39

# (Operator) ......................... 39

% (Operator) ......................... 39

% (System variable) .............. 139
% (System variable) .............. 139
%c (Variable) ....................... 594
%e (Constant) ....................... 205
%e_to_numlog (Option variable) 209
%edispflag (Option variable) 140
%emode (Option variable) .......... 81
%enumer (Option variable) ....... 81
%f (Function) ....................... 238
%gamma (Constant) ................. 436
%i (Constant) ....................... 205
%iargs (Option variable) ....... 214
%k1 (Variable) ..................... 505
%k2 (Variable) ..................... 505
%m (Function) ....................... 238
%phi (Constant) .................... 205
%p (Constant) ...................... 206
%piargs (Option variable) ....... 213
%rnum_list (System variable) ... 285
%s (Function) ....................... 224
%th (Function) ..................... 140
%unitexpand (Variable) .......... 954
%w (Function) ....................... 238

. (Operator) ......................... 40

/ (Operator) ......................... 33

: (Operator) ......................... 40
:: (Operator) ......................... 41
:= (Operator) ....................... 42
:= (Operator) ....................... 43

< (Operator) ......................... 36
<= (Operator) ....................... 37

= (Operator) ......................... 44

> (Operator) ......................... 37
>= (Operator) ....................... 37

? (Special symbol) .............. 140
?? (Special symbol) ............ 140

@ (Operator) ......................... 530

[ (Special symbol) ............ 350

] (Special symbol) ............ 350

^ (Operator) ......................... 33
^^ (Operator) ....................... 38
- (System variable) .......................... 138
_ (System variable) ......................... 137

' (Operator) ................................ 744
'' (Operator) ............................... 746

| (Operator) ................................ 384

" (Operator) ............................... 383

A
abasep (Function) .......................... 418
abs (Function) ............................. 47
absboxchar (Option variable) ............. 140
absint (Function) .......................... 314
absolute_real_time (Function) .......... 466
acos (Function) ........................... 214
acosh (Function) .......................... 214
acot (Function) ........................... 214
acoth (Function) ........................... 214
acs (Function) ............................ 214
acsch (Function) ........................... 214
activate (Function) ........................ 173
activecontexts (System variable) ....... 173
adapt_depth (Graphic option) .......... 668
adapt_depth (Plot option) ............... 129
add_edge (Function) ...................... 792
add_edges (Function) ..................... 792
add_vertex (Function) .................... 793
add_vertices (Function) ................. 793
addcol (Function) ........................ 328
additive (Keyword) ....................... 47
addmatrices (Function) ................. 837
addr (Function) ........................... 328
adim (Variable) ........................... 417
adjacency_matrix (Function) .......... 777
adjjoin (Function) ........................ 505
adjoint (Function) ....................... 328
af (Function) ............................. 418
aform (Variable) .......................... 417
agd (Function) ............................ 903
airy_ai (Function) ......................... 224
airy_b (Function) .......................... 224
airy_dai (Function) ....................... 224
airy dbi (Function) ...................... 225
alg&type (Function) ...................... 417
algebraic (Option variable) .......... 179
algebep (Option variable) ............. 169
algeexact (Option variable) .......... 285
algvar (Function) ......................... 285
alias (Function) ........................... 21
alias (System variable) ................. 467
all (Option variable) ..................... 353
allbut (Keyword) .......................... 47
allroots (Function) ...................... 287
allsym (Option variable) ............... 368
algebraic (Declaration) ................. 467
alphacharp (Function) ................... 937
alphanumeric (Function) ................. 937
amortization (Function) ................. 760
and (Operator) ............................ 46
annuity_fv (Function) ................... 760
annuity_pv (Function) ................... 760
antid (Function) ........................ 251
antidiff (Function) ...................... 252
AntiDifference (Function) .............. 956
antisymmetric (Declaration) .......... 48
append (Function) ....................... 493
appendfile (Function) ................... 493
apply (Function) .......................... 538
apply1 (Function) ......................... 475
apply2 (Function) ......................... 475
apply1 (Function) ......................... 475
apropos (Function) ........................ 11
args (Function) ........................... 467
arit_amortization (Function) .......... 761
arithmetic (Function) .................... 902
arithmetic (Function) .................... 902
array (Function) ........................... 317
arrayapply (Function) .................... 317
arrayinfo (Function) ..................... 317
arraymake (Function) .................... 319
arrays (System variable) ............... 320
ascii (Function) .......................... 937
asec (Function) ........................... 214
asech (Function) ........................... 215
asin (Function) ........................... 215
asinh (Function) ........................... 215
askexp (System variable) ............... 101
askinteger (Function) .................... 101
asksign (Function) ....................... 101
assoc (Function) .......................... 493
assoc_legendre_p (Function) ........... 880
assoc_legendre_q (Function) .......... 881
assume (Function) ........................ 173
assume_external_byte_order (Function) 868
assume_pos (Option variable) .......... 174
assume_pos_pre (Option variable) ..... 175
assumescalar (Option variable) ..... 174
assymbol (Variable) ...................... 417
asympt (Function) ......................... 581
atan (Function) ........................... 71
atan2 (Function) .......................... 215
atanh (Function) .......................... 215
atensimp (Function) .................... 417
atom (Function) ........................... 493
Appendix A: Function and Variable Index 963

atomgrad (Property) 252
atrigi (Package) 215
atvalue (Function) 252, 253
augcoefmatrix (Function) 328
augmented_lagrangian_method (Function) 583
av (Function) 418
average_degree (Function) 777
axes (Plot option) 129
axis_3d (Graphic option) 668
axis_bottom (Graphic option) 668
axis_left (Graphic option) 669
axis_right (Graphic option) 669
axis_top (Graphic option) 669
azimut (Plot option) 129

B
background_color (Graphic option) 669
backsubt (Option variable) 288
backtrace (Function) 561
bars (Graphic object) 705
barsplot (Function) 617
bashindices (Function) 320
batch (Function) 141
batchload (Function) 142
bc2 (Function) 303
bdvac (Function) 405
belln (Function) 505
benefit_cost (Function) 763
berlefact (Option variable) 180
bern (Function) 433
bernpoly (Function) 433
bessel_i (Function) 222
bessel_j (Function) 221
bessel_k (Function) 222
bessel_y (Function) 222
besselzero (Option variable) 223
beta (Function) 227
beta_args_sum_to_integer (Option variable) 235
beta_expand (Option variable) 234
beta_incomplete (Function) 228
beta_incomplete_generalized (Function) 232
beta_incomplete_regularized (Function) 231
bezout (Function) 180
bf_find_root (Function) 312
bf_fmin_cobyla (Function) 588
bfalroots (Function) 288
bffac (Function) 169
bfzeta (Function) 433
bfloat (Function) 169
bfloatp (Function) 169
bfpsi (Function) 169
bfpsio (Function) 169
bfstorat (Option variable) 169
bftrunc (Option variable) 169
bfzeta (Function) 433
biconnected_components (Function) 777
bimetric (Function) 405
binomial (Function) 433
bipartition (Function) 778
block (Function) 539
blockmatrixp (Function) 837
bode_gain (Function) 585
bode_phase (Function) 586
border (Graphic option) 669
bothcoef (Function) 180
boundaries_array (Global variable) 721
box (Function) 72
box (Plot option) 130
boxchar (Option variable) 73
boxplot (Function) 619
break (Function) 540
breakup (Option variable) 288
bug_report (Function) 8
build_info (Function) 8
buildq (Function) 534
burn (Function) 434

c

cabs (Function) 48
canform (Function) 369
canten (Function) 368
cardinality (Function) 506
carg (Function) 73
cartan (Function) 253
cartesian_product (Function) 506
catch (Function) 540
caucy_matrix (Function) 328
cauclysum (Option variable) 419
cbffac (Function) 170
cbrange (Graphic option) 670
cbtics (Graphic option) 670
cdf_bernoulli (Function) 658
cdf_beta (Function) 645
cdf_binomial (Function) 655
cdf_cauchy (Function) 654
cdf_chi2 (Function) 636
cdf_continuous_uniform (Function) 646
cdf_discrete_uniform (Function) 660
cdf_exp (Function) 641
cdf_f (Function) 639
cdf_gamma (Function) 644
cdf_geometric (Function) 659
cdf_gumbel (Function) 654
cdf_hypgeometric (Function) 661
cdf_laplace (Function) 653
cdf_logistic (Function) 647
cdf_lognormal (Function) 643
cdf_negative_binomial (Function) 662
cdf_noncentral_chi2 (Function) 638
cdf_noncentral_student_t (Function) 634
cdf_normal (Function) 631
cdf_paret (Function) 648
cdf_poisson (Function) 656
cdf_rank_sum (Function) .................................. 930
cdf_rayleigh (Function) .................................. 650
cdf_signed_rank (Function) ................................. 929
cdf_student_t (Function) .................................. 632
cdf_weibull (Function) .................................... 649
cdisplay (Function) ........................................ 406
ceiling (Function) ......................................... 48
central_moment (Function) ................................. 606
cequal (Function) ........................................... 937
cequalignore (Function) .................................... 937
cf (Function) ............................................... 434
cfdisrep (Function) ......................................... 435
cfexpand (Function) .......................................... 435
cflength (Option variable) ................................. 436
cframe_flag (Option variable) .............................. 410
cgeodesic (Function) ....................................... 405
cgreaterp (Function) ........................................ 937
cgreaterpiignore (Function) ................................. 937
changename (Function) ..................................... 359
changevar (Function) ....................................... 263
chaosgame (Function) ....................................... 731
char (Function) .............................................. 938
charfun (Function) .......................................... 48
charfun2 (Function) ......................................... 815
charlist (Function) ......................................... 938
charp (Function) ............................................ 937
charpoly (Function) ......................................... 329
chebyshev_t (Function) ..................................... 881
chebyshev_u (Function) ..................................... 881
check_overlaps (Function) ................................. 352
checkdiv (Function) ......................................... 405
cholesky (Function) ......................................... 838
christof (Function) .......................................... 394
chromatic_index (Function) ................................. 778
chromatic_number (Function) ................................. 778
cint (Function) ............................................... 937
circulant_graph (Function) ................................. 772
clear_edge_weight (Function) ............................... 778
clear_rules (Function) ....................................... 490
clear_vertex_label (Function) ............................... 779
clebsch_graph (Function) .................................... 773
cless (Function) ............................................. 937
clessignore (Function) ....................................... 937
close (Function) ............................................ 934
closefile (Function) .......................................... 142
cmetric (Function) .......................................... 391
cnonsset_flag (Option variable) ............................. 411
coff (Function) ............................................. 180
coeffmatrix (Function) ...................................... 390
cograd (Function) ............................................ 404
col (Function) .............................................. 330
collapse (Function) ......................................... 142
collectterms (Function) ..................................... 901
color (Graphic option) ....................................... 671
color (Plot option) .......................................... 130
colorbox (Graphic option) ................................. 671
colorbox (Plot option) ....................................... 130
columnop (Function) ........................................ 837
columns (Graphic option) ................................... 672
columnspace (Function) ..................................... 837
columnswap (Function) ...................................... 837
columnvector (Function) .................................... 330
combination (Function) ..................................... 903
combine (Function) ......................................... 181
commutative (Declaration) ................................. 49
comp2pui (Function) ........................................ 443
compareTo (Function) ...................................... 49
compile (Function) ......................................... 540
compile (Function) ......................................... 540
compile_file (Function) ..................................... 559
complement_graph (Function) ............................... 773
cdf_bipartite_graph (Function) ............................. 773
cdf_graph (Function) ........................................ 773
components (Function) ..................................... 362
concat (Function) .......................................... 368
concat (Function) .......................................... 142
conjugate (Function) ....................................... 330
constderiv (Function) ....................................... 372
connect_vertices (Function) ............................... 793
connected_components (Function) ........................ 779
cos (Function) ............................................. 493
constant (Special operator) ................................. 73
constantp (Function) ....................................... 73
constituent (Function) ...................................... 937
context (Option variable) .................................. 176
dcontext (Option variable) ................................. 176
dcontinuos_freq (Function) ................................. 601
dcontortion (Function) .................................... 402
dcontour (Function) ......................................... 672
dcontour_level (Function) .................................. 672
dcontour_plot (Function) .................................... 114
dcontract (Function) ........................................ 362
dcontract (Function) ........................................ 447
dcontract_edge (Function) .................................. 793
dcontraflag (Function) ...................................... 404
contrib_ode (Function) ...................................... 593
dconvert (Function) .......................................... 948
dcoord (Function) ........................................... 372
dcopy (Function) ............................................ 837
dcopy_graph (Function) ...................................... 772
dcopylist (Function) ......................................... 493
dcopymatrix (Function) ...................................... 331
dcor (Function) ............................................. 613
dcos (Function) ............................................. 215
dcos (Function) ............................................. 215
dcosflag (Function) ........................................... 315
dcot (Function) ............................................. 215
doth (Function) ............................................. 215
dcov (Function) ............................................. 612
dcov (Function) ............................................. 612
dcov (Function) ............................................. 375
dcovect (Function) .......................................... 330
dcovers (Function) .......................................... 903
dcreate_graph (Function) ................................... 771
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create_list</code></td>
<td>494</td>
</tr>
<tr>
<td><code>csc</code></td>
<td>215</td>
</tr>
<tr>
<td><code>csh</code></td>
<td>215</td>
</tr>
<tr>
<td><code>csetup</code></td>
<td>391</td>
</tr>
<tr>
<td><code>ct_coords</code> (Option variable)</td>
<td>413</td>
</tr>
<tr>
<td><code>ct_coordsys</code> (Function)</td>
<td>391</td>
</tr>
<tr>
<td><code>ctaylor</code></td>
<td>396</td>
</tr>
<tr>
<td><code>ctapply</code> (Option variable)</td>
<td>411</td>
</tr>
<tr>
<td><code>ctaypt</code> (Option variable)</td>
<td>411</td>
</tr>
<tr>
<td><code>ctayswitch</code> (Option variable)</td>
<td>411</td>
</tr>
<tr>
<td><code>ctayvar</code> (Option variable)</td>
<td>411</td>
</tr>
<tr>
<td><code>ctorsion_flag</code> (Option variable)</td>
<td>411</td>
</tr>
<tr>
<td><code>cttransform</code> (Function)</td>
<td>265</td>
</tr>
<tr>
<td><code>cttranspose</code> (Option variable)</td>
<td>838</td>
</tr>
<tr>
<td><code>ctrsgimp</code> (Option variable)</td>
<td>410</td>
</tr>
<tr>
<td><code>cube_graph</code> (Function)</td>
<td>773</td>
</tr>
<tr>
<td><code>cuboctahedron_graph</code> (Function)</td>
<td>773</td>
</tr>
<tr>
<td><code>cumlisp</code> (Function)</td>
<td>938</td>
</tr>
<tr>
<td><code>current_let_rule_package</code> (Option variable)</td>
<td>733</td>
</tr>
<tr>
<td><code>cv</code> (Function)</td>
<td>606</td>
</tr>
<tr>
<td><code>cycle_digraph</code> (Function)</td>
<td>773</td>
</tr>
<tr>
<td><code>cycle_graph</code> (Function)</td>
<td>773</td>
</tr>
<tr>
<td><code>cylindrical</code> (Graphic object)</td>
<td>705</td>
</tr>
<tr>
<td><code>data_file_name</code> (Graphic option)</td>
<td>673</td>
</tr>
<tr>
<td><code>days360</code> (Function)</td>
<td>759</td>
</tr>
<tr>
<td><code>dblint</code> (Function)</td>
<td>264</td>
</tr>
<tr>
<td><code>deactivate</code> (Function)</td>
<td>176</td>
</tr>
<tr>
<td><code>debugmode</code> (Option variable)</td>
<td>21</td>
</tr>
<tr>
<td><code>declare</code> (Function)</td>
<td>74</td>
</tr>
<tr>
<td><code>declare_constvalue</code> (Function)</td>
<td>749</td>
</tr>
<tr>
<td><code>declare_dimensions</code> (Function)</td>
<td>753</td>
</tr>
<tr>
<td><code>declare_fundamental_dimensions</code> (Function)</td>
<td>753</td>
</tr>
<tr>
<td><code>declare_fundamental_units</code> (Function)</td>
<td>754</td>
</tr>
<tr>
<td><code>declare_qty</code> (Function)</td>
<td>751</td>
</tr>
<tr>
<td><code>declare_translated</code> (Function)</td>
<td>411</td>
</tr>
<tr>
<td><code>declare_unit_conversion</code> (Function)</td>
<td>559</td>
</tr>
<tr>
<td><code>declare_units</code> (Function)</td>
<td>750</td>
</tr>
<tr>
<td><code>declare_weights</code> (Function)</td>
<td>352</td>
</tr>
<tr>
<td><code>decsym</code> (Function)</td>
<td>368</td>
</tr>
<tr>
<td><code>default_let_rule_package</code> (Option variable)</td>
<td>476</td>
</tr>
<tr>
<td><code>defcon</code> (Function)</td>
<td>361</td>
</tr>
<tr>
<td><code>define</code> (Function)</td>
<td>541</td>
</tr>
<tr>
<td><code>define_variable</code> (Function)</td>
<td>542</td>
</tr>
<tr>
<td><code>defint</code> (Function)</td>
<td>265</td>
</tr>
<tr>
<td><code>defmatch</code> (Function)</td>
<td>476</td>
</tr>
<tr>
<td><code>defrule</code> (Function)</td>
<td>478</td>
</tr>
<tr>
<td><code>defstruct</code> (Function)</td>
<td>529</td>
</tr>
<tr>
<td><code>deftheory</code> (Function)</td>
<td>419</td>
</tr>
<tr>
<td><code>degree_sequence</code> (Function)</td>
<td>779</td>
</tr>
<tr>
<td><code>del</code> (Function)</td>
<td>254</td>
</tr>
<tr>
<td><code>delay</code> (Graphic option)</td>
<td>674</td>
</tr>
<tr>
<td><code>delete</code> (Function)</td>
<td>494</td>
</tr>
<tr>
<td><code>deleten</code> (Function)</td>
<td>410</td>
</tr>
<tr>
<td><code>delta</code> (Function)</td>
<td>254</td>
</tr>
<tr>
<td><code>demo</code> (Function)</td>
<td>11</td>
</tr>
<tr>
<td><code>demoivre</code> (Option variable)</td>
<td>101</td>
</tr>
<tr>
<td><code>denom</code> (Function)</td>
<td>182</td>
</tr>
<tr>
<td><code>dependencies</code> (System variable)</td>
<td>254</td>
</tr>
<tr>
<td><code>depends</code> (Function)</td>
<td>254</td>
</tr>
<tr>
<td><code>derivabbrev</code> (Option variable)</td>
<td>255</td>
</tr>
<tr>
<td><code>derivdegree</code> (Function)</td>
<td>255</td>
</tr>
<tr>
<td><code>derivlist</code> (Function)</td>
<td>256</td>
</tr>
<tr>
<td><code>derivsubst</code> (Option variable)</td>
<td>256</td>
</tr>
<tr>
<td><code>describe</code> (Function)</td>
<td>13</td>
</tr>
<tr>
<td><code>detsolve</code> (Function)</td>
<td>303</td>
</tr>
<tr>
<td><code>determinant</code> (Function)</td>
<td>331</td>
</tr>
<tr>
<td><code>detout</code> (Option variable)</td>
<td>331</td>
</tr>
<tr>
<td><code>diag</code> (Function)</td>
<td>821</td>
</tr>
<tr>
<td><code>diag_matrix</code> (Function)</td>
<td>821</td>
</tr>
<tr>
<td><code>diagmatrix</code> (Function)</td>
<td>838</td>
</tr>
<tr>
<td><code>diagmatrixp</code> (Function)</td>
<td>331</td>
</tr>
<tr>
<td><code>diagnostic</code> (Option variable)</td>
<td>405</td>
</tr>
<tr>
<td><code>diameter</code> (Function)</td>
<td>779</td>
</tr>
<tr>
<td><code>diff</code> (Function)</td>
<td>256, 309</td>
</tr>
<tr>
<td><code>diff</code> (Special symbol)</td>
<td>257</td>
</tr>
<tr>
<td><code>digtcharp</code> (Function)</td>
<td>938</td>
</tr>
<tr>
<td><code>dim</code> (Option variable)</td>
<td>410</td>
</tr>
<tr>
<td><code>dimacs_export</code> (Function)</td>
<td>794</td>
</tr>
<tr>
<td><code>dimacs_import</code> (Function)</td>
<td>794</td>
</tr>
<tr>
<td><code>dimension</code> (Function)</td>
<td>290</td>
</tr>
<tr>
<td><code>dimensionless</code> (Function)</td>
<td>756</td>
</tr>
<tr>
<td><code>dimensions</code> (Function)</td>
<td>754</td>
</tr>
<tr>
<td><code>dimensions</code> (Graphic option)</td>
<td>674</td>
</tr>
<tr>
<td><code>dimensions_as_list</code> (Function)</td>
<td>754</td>
</tr>
<tr>
<td><code>direct</code> (Function)</td>
<td>448</td>
</tr>
<tr>
<td><code>discrete_freq</code> (Function)</td>
<td>601</td>
</tr>
<tr>
<td><code>disjoin</code> (Function)</td>
<td>506</td>
</tr>
<tr>
<td><code>disjointp</code> (Function)</td>
<td>507</td>
</tr>
<tr>
<td><code>disolate</code> (Function)</td>
<td>79</td>
</tr>
<tr>
<td><code>disp</code> (Function)</td>
<td>143</td>
</tr>
<tr>
<td><code>dispcon</code> (Function)</td>
<td>143</td>
</tr>
<tr>
<td><code>disflag</code> (Option variable)</td>
<td>290</td>
</tr>
<tr>
<td><code>disform</code> (Function)</td>
<td>79</td>
</tr>
<tr>
<td><code>disfun</code> (Function)</td>
<td>544</td>
</tr>
<tr>
<td><code>disjordan</code> (Function)</td>
<td>622</td>
</tr>
<tr>
<td><code>display</code> (Function)</td>
<td>143</td>
</tr>
<tr>
<td><code>display_format_internal</code> (Option variable)</td>
<td>144</td>
</tr>
<tr>
<td><code>display2d</code> (Option variable)</td>
<td>144</td>
</tr>
<tr>
<td><code>disrule</code> (Function)</td>
<td>478</td>
</tr>
<tr>
<td><code>disterms</code> (Function)</td>
<td>144</td>
</tr>
<tr>
<td><code>distrib</code> (Function)</td>
<td>80</td>
</tr>
<tr>
<td><code>distribute_over</code> (Option variable)</td>
<td>101</td>
</tr>
</tbody>
</table>
divide (Function) ........................................ 182
divisors (Function) ..................................... 507
divsum (Function) ....................................... 436
dkummer_m (Function) .................................. 595
dkummer_u (Function) .................................. 595
dlange (Function) ........................................ 823
do (Special operator) ................................... 562
doallxops (Option variable) ......................... 332
dodecahedron_graph (Function) ..................... 773
domain (Option variable) .............................. 102
domexpt (Option variable) ......................... 332
domxops (Option variable) ............................ 332
domxctimes (Option variable) ..................... 332
dontfactor (Option variable) ....................... 332
doscmxops (Option variable) ...................... 333
doscmxplus (Option variable) ..................... 333
dot0nascsimp (Option variable) .................. 333
dot0simp (Option variable) ........................... 333
dot1simp (Option variable) ............................ 333
dotassoc (Option variable) ............................ 333
dotconstrules (Option variable) .................. 333
dotdistrib (Option variable) .......................... 333
dotexp_tsimp (Option variable) .................. 333
dotident (Option variable) ............................ 334
dotproduct (Function) .................................. 838
dotscrules (Option variable) ...................... 334
dot simp (Function) ..................................... 352
dpart (Function) ......................................... 80
draw (Function) ......................................... 666
draw_file (Function) .................................... 667
draw_graph (Function) .................................. 795
draw_graph_program (Option variable) .......... 797
draw2d (Function) ....................................... 666
draw2d (Function) ....................................... 667
drawdf (Function) ....................................... 727
dscalar (Function) ....................................... 257, 404
ele2polyomne (Function) ............................. 452
ele2zpol (Function) ..................................... 443
elem (Function) .......................................... 444
elementp (Function) .................................... 508
elevation (Plot option) ............................... 130
elevation_grid (Graphic object) ................. 705
eliminate (Function) ................................. 182
eclipse (Graphic object) .............................. 706
e elliptic_e (Function) ................................. 246
e elliptic_ec (Function) ............................... 247
e elliptic_eu (Function) ............................... 246
e elliptic_f (Function) ................................. 246
e elliptic_kc (Function) ............................... 247
e elliptic_pi (Function) ............................... 247
ematrix (Function) ...................................... 337
empty_graph (Function) .............................. 773
emptyp (Function) ....................................... 508
endcons (Function) ...................................... 495
enhanced3d (Graphic option) ..................... 675
entermatrix (Function) ............................... 337
entertensor (Function) ............................... 359
entier (Function) ........................................ 49
epsilon_lp (Option variable) .................... 897
equal (Function) ......................................... 49
equalp (Function) ........................................ 314
equiv_classes (Function) ............................. 508
erf (Function) ........................................... 237
erf_generalized (Function) ....................... 237
erf_representation (Option variable) .......... 237
erfc (Function) .......................................... 237
errflag (Option variable) ......................... 265
erfi (Function) .......................................... 237
errcatch (Function) ..................................... 564
error (Function) ......................................... 564
error (System variable) ............................. 565
error_size (Option variable) .................... 145
erro_sym (Option variable) ....................... 145
error_type (Graphic option) ...................... 677
errormsg (Function) .................................... 565
errormsg (Option variable) ....................... 565
errors (Graphic object) ............................. 706
euler (Function) ......................................... 436
ev (Function) ............................................ 21
ev_point (Global variable) ....................... 958
eval (Special symbol) .................................. 24
eval_string (Function) ................................. 939
evenp (Function) .......................................... 52
every (Function) ......................................... 508
evflag (Property) ........................................ 24
evfun (Property) ......................................... 25
evolution (Function) .................................... 731
evolution2d (Function) ............................... 732
evundiff (Function) ..................................... 371
example (Function) ....................................... 14
exp (Function) ............................................ 80
expand (Function) ....................................... 102
expand (Function) ....................................... 104
expand_denom (Option variable) ............... 104

E

echelon (Function) ....................................... 334
edge_color (draw_graph_option) ................. 798
edge_coloring (draw_graph_option) .......... 799
edge_coloring (Function) ............................ 779
edge_connectivity (Function) .................... 780
edge_partition (draw_graph_option) .......... 799
edge_type (draw_graph_option) ................. 798
edge_width (draw_graph_option) ............... 798
edges (Function) ......................................... 780
eigens_by_jacobi (Function) ..................... 838
eigenvalues (Function) ............................... 334
eigenvectors (Function) .............................. 335
eighth (Function) ......................................... 405
einstein (Function) ...................................... 395
eivals (Function) ......................................... 334
eivects (Function) ......................................... 335
elapsed_real_time (Function) .................... 466
elapsed_run_time (Function) ..................... 466
ele2comp (Function) ..................................... 444
expandwrt_factorized (Function) .................. 104
expintegral_chichi (Function) .................. 236
expintegral_ci (Function) .................. 236
expintegral_e (Function) .................. 236
expintegral_e1 (Function) .................. 236
expintegral_ei (Function) .................. 236
expintegral_li (Function) .................. 236
expintegral_shi (Function) .................. 236
expintegral_sii (Function) .................. 236
expintegrate (Option variable) .................. 237
expintegrate (Option variable) .................. 237
explicit (Option variable) .................. 237
explicit (Graphic object) ................. 707
explode (Function) .................. 447
expo (Option variable) .................. 104
exponentialize (Function) .................. 105
exponentialize (Option variable) .................. 105
expon (Function) .................. 903
extdiff (Function) .................. 384
extract_linear_equations (Function) .................. 352
extremal_subset (Function) .................. 509
ezgcd (Function) .................. 182
file_name (Graphic option) .................. 677
file_output_append (Option variable) .................. 141
file_search (Function) .................. 146
file_search (Option variable) .................. 147
file_search (Option variable) .................. 147
file_search_maxima (Option variable) .................. 147
file_search_tests (Option variable) .................. 147
file_search_usage (Option variable) .................. 147
file_type (Function) .................. 147
file_type_lisp (Option variable) .................. 147
file_type_maxima (Option variable) .................. 148
filename_merge (Function) .................. 146
fill_color (Graphic option) .................. 677
fill_density (Graphic option) .................. 678
fillarray (Function) .................. 320
filled_graph (Graphic option) .................. 678
find_root (Function) .................. 312
find_root_abs (Option variable) .................. 312
find_root_error (Option variable) .................. 312
find_root_rel (Option variable) .................. 312
findode (Function) .................. 403
first (Function) .................. 495
fix (Function) .................. 52
fixed_vertices (draw_graph option) .................. 800
flatten (Function) .................. 510
flength (Function) .................. 934
float (Function) .................. 170
float2bf (Option variable) .................. 170
floatump (Function) .................. 170
floor (Function) .................. 51
flower_snark (Function) .................. 773
flush (Function) .................. 371
flushderiv (Function) .................. 374
flushd (Function) .................. 372
flushd (Function) .................. 372
min_cobyla (Function) .................. 587
font (Graphic option) .................. 678
font_size (Graphic option) .................. 680
fort (Function) .................. 566
forget (Function) .................. 177
fortdent (Option variable) .................. 310
fortran (Function) .................. 310
fortspaces (Option variable) .................. 311
fourcos (Function) .................. 315
fourexpand (Function) .................. 315
fourier (Function) .................. 314
fourint (Function) .................. 315
fourintcos (Function) .................. 315
fourintsin (Function) .................. 315
foursimp (Function) .................. 314
foursin (Function) .................. 315
fourth (Function) .................. 495
fposition (Function) .................. 935
fposition (Function) .................. 935
fpprec (Option variable) .................. 170
fpprintprec (Option variable) .................. 170
frame_bracket (Function) .................. 399
freeo (Function) .................. 81

F

f90 (Function) .................. 757
faceexpand (Option variable) .................. 183
facsum (Function) .................. 900
facsum_combine (Global variable) .................. 900
factcomb (Function) .................. 183
factlim (Option variable) .................. 105
factor (Function) .................. 105
factorfacsum (Function) .................. 901
factorflag (Option variable) .................. 185
factorial (Function) .................. 437
factorial_expand (Option variable) .................. 437
factorout (Function) .................. 185
factors (Function) .................. 186
facts (Function) .................. 177
false (Constant) .................. 205
fast_central_elements (Function) .................. 352
fast_linsolve (Function) .................. 351
fasttimes (Function) .................. 186
fb (Variable) .................. 412
feature (Declaration) .................. 463
featurep (Function) .................. 464
features (Declaration) .................. 177
fernfal (Function) .................. 766
fft (Function) .................. 308
fib (Function) .................. 437
fibotphi (Function) .................. 437
fifth (Function) .................. 495
file_name (draw_graph option) .................. 799
Appendix A: Function and Variable Index

H
halfangles (Option variable) ........................................ 216
hamilton_cycle (Function) ......................................... 782
hamilton_path (Function) ........................................... 782
hankel (Function) ................................................... 839
hankel_1 (Function) ................................................ 222
hankel_2 (Function) ................................................ 223
harmonic (Function) ............................................... 902
harmonic_mean (Function) .................................... 609
hay (Function) ...................................................... 903
head_angle (draw_graph option) ............................. 799
head_angle (Graphic option) ................................. 680
head_both (Graphic option) ................................... 681
head_length (draw_graph option) ......................... 799
head_length (Graphic option) ................................ 681
head_type (Graphic option) .................................. 682
heaxedron_graph (Function) ................................ 774
hermite (Function) .................................................. 881
hessian (Function) ............................................... 840
hgfred (Function) .................................................. 241
hilbert_matrix (Function) .................................... 840
hilbertmap (Function) ......................................... 767
hipov (Function) ................................................... 189
histogram (Function) ............................................ 615
hodge (Function) ................................................... 385
horner (Function) ................................................. 311
hypergeometric (Function) .................................. 238
hypergeometric_representation (Option variable) 237

I
ibase (Option variable) ....................................... 150
ic (Function) ....................................................... 304
ic1 (Function) ..................................................... 304
ic2 (Function) ..................................................... 304
icci (Variable) ..................................................... 379
icci2 (Variable) ................................................... 379
ichr1 (Function) ................................................... 374
ichr2 (Function) ................................................... 375
icosahedron_graph (Function) .......................... 774
icosidodecahedron_graph (Function) ..................... 774
icosahedron_graph (Function) ............................. 774
icounter (Option variable) .................................... 365
ic (Function) ....................................................... 375
ident (Function) .................................................... 339
identfor (Function) .............................................. 840
identity (Function) .............................................. 511
idiff (Function) ................................................... 370
idim (Function) .................................................... 374
idummys (Function) ............................................. 365
idummys (Option variable) .................................... 365
ieq (Function) ..................................................... 292
ieqnp (Function) ................................................... 292
if (Special operator) ............................................ 566
ifactors (Function) .............................................. 437
if (Variable) ...................................................... 378
ifc1 (Variable) ..................................................... 379
ifc2 (Variable) ..................................................... 379
ifg (Variable) ........................................................ 380
ifgi (Variable) ..................................................... 380
ifr (Variable) ...................................................... 380
iframe_bracket_form (Option variable) ............. 380
iframes (Function) ............................................ 378
iftri (Variable) ..................................................... 380
ifs (Function) ..................................................... 732
igeodesic_coords (Function) ................................ 376
igeode_flag (Option variable) ............................ 385
ikt1 (Variable) ..................................................... 381
ikt2 (Variable) ..................................................... 381
ilt (Function) ...................................................... 265
image (Graphic object) ...................................... 708
imagpart (Function) ............................................. 374
inmetric (Function) ............................................. 374
integrate_use_rootsof (Option variable) ......... 271
integrate_constant (System variable) ............... 269
integration_constant (System variable) ............ 269
integration_constant_counter (System variable) .... 270
inr (Function) ..................................................... 85
inprod (Function) ............................................... 340
integrate (Function) ............................................ 266
int analyzing (Option variable) ......................... 266
integer_partitions (Function) ................................. 511
integerp (Function) ............................................. 469
integrate (Function) ............................................ 266
integrate_use_rootsof (Option variable) .......... 271
intersection (Function) ........................................ 512
intersection (Function) ........................................ 512
interval (Function) ............................................. 881
intfacin (Option variable) .................................. 189
intopois (Function) ............................................. 431
intosum (Function) ............................................. 105
invmod (Function) .............................................. 438
inv (Variable) ..................................................... 405
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>invariant2</td>
<td>405</td>
</tr>
<tr>
<td>inverse_fft</td>
<td>307</td>
</tr>
<tr>
<td>inverse_jacobi_cd</td>
<td>246</td>
</tr>
<tr>
<td>inverse_jacobi_cn</td>
<td>245</td>
</tr>
<tr>
<td>inverse_jacobi_cs</td>
<td>245</td>
</tr>
<tr>
<td>inverse_jacobi_dc</td>
<td>246</td>
</tr>
<tr>
<td>inverse_jacobi_dn</td>
<td>245</td>
</tr>
<tr>
<td>inverse_jacobi_ds</td>
<td>246</td>
</tr>
<tr>
<td>inverse_jacobi_nc</td>
<td>246</td>
</tr>
<tr>
<td>inverse_jacobi_nd</td>
<td>246</td>
</tr>
<tr>
<td>inverse_jacobi_ns</td>
<td>245</td>
</tr>
<tr>
<td>inverse_jacobi_sc</td>
<td>245</td>
</tr>
<tr>
<td>jacobi (Function)</td>
<td>438</td>
</tr>
<tr>
<td>jacobi_cd</td>
<td>245</td>
</tr>
<tr>
<td>jacobi_cn</td>
<td>245</td>
</tr>
<tr>
<td>jacobi_cs</td>
<td>245</td>
</tr>
<tr>
<td>jacobi_dc</td>
<td>245</td>
</tr>
<tr>
<td>jacobi_dn</td>
<td>244</td>
</tr>
<tr>
<td>jacobi_ds</td>
<td>244</td>
</tr>
<tr>
<td>jacobi_nc</td>
<td>245</td>
</tr>
<tr>
<td>jacobi_nd</td>
<td>245</td>
</tr>
<tr>
<td>jacobi_ns</td>
<td>245</td>
</tr>
<tr>
<td>jacobi_p</td>
<td>881</td>
</tr>
<tr>
<td>jacobi_sc</td>
<td>245</td>
</tr>
<tr>
<td>jacobi_sd</td>
<td>245</td>
</tr>
<tr>
<td>jacobi_sn</td>
<td>244</td>
</tr>
<tr>
<td>jacobian</td>
<td>840</td>
</tr>
<tr>
<td>JF</td>
<td>621</td>
</tr>
<tr>
<td>join (Function)</td>
<td>496</td>
</tr>
<tr>
<td>jordan (Function)</td>
<td>622</td>
</tr>
<tr>
<td>julia (Function)</td>
<td>732</td>
</tr>
<tr>
<td>julia_parameter</td>
<td>766</td>
</tr>
<tr>
<td>julia_set (Function)</td>
<td>766</td>
</tr>
<tr>
<td>julia_sin</td>
<td>766</td>
</tr>
</tbody>
</table>

**K**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>kdel (Function)</td>
<td>365</td>
</tr>
<tr>
<td>kdelta (Function)</td>
<td>365</td>
</tr>
<tr>
<td>keepfloat</td>
<td>190</td>
</tr>
<tr>
<td>key (Graphic option)</td>
<td>682</td>
</tr>
<tr>
<td>kill (Function)</td>
<td>26, 27</td>
</tr>
<tr>
<td>killcontext (Function)</td>
<td>177</td>
</tr>
<tr>
<td>kInvariant</td>
<td>412</td>
</tr>
<tr>
<td>kostka (Function)</td>
<td>451</td>
</tr>
<tr>
<td>kron_delta (Function)</td>
<td>513</td>
</tr>
<tr>
<td>kroncker_product (Function)</td>
<td>841</td>
</tr>
<tr>
<td>kt (Variable)</td>
<td>413</td>
</tr>
<tr>
<td>kummer_m</td>
<td>595</td>
</tr>
<tr>
<td>kummer_u</td>
<td>595</td>
</tr>
<tr>
<td>kurtosis (Function)</td>
<td>610</td>
</tr>
<tr>
<td>kurtosis_bernoulli</td>
<td>659</td>
</tr>
<tr>
<td>kurtosis_beta</td>
<td>646</td>
</tr>
<tr>
<td>kurtosis_binomial</td>
<td>656</td>
</tr>
<tr>
<td>kurtosis_chi2</td>
<td>637</td>
</tr>
<tr>
<td>kurtosis_continuous_uniform</td>
<td>647</td>
</tr>
<tr>
<td>kurtosis_discrete_uniform</td>
<td>661</td>
</tr>
<tr>
<td>kurtosis_exp</td>
<td>642</td>
</tr>
<tr>
<td>kurtosis_f</td>
<td>640</td>
</tr>
<tr>
<td>kurtosis_gamma</td>
<td>645</td>
</tr>
<tr>
<td>kurtosis_geometric</td>
<td>660</td>
</tr>
<tr>
<td>kurtosis_gumbel</td>
<td>655</td>
</tr>
<tr>
<td>kurtosis_hypergeometric</td>
<td>662</td>
</tr>
<tr>
<td>kurtosis_laplace</td>
<td>653</td>
</tr>
<tr>
<td>kurtosis_logistic</td>
<td>648</td>
</tr>
<tr>
<td>kurtosis_lognormal</td>
<td>644</td>
</tr>
<tr>
<td>kurtosis_negative_binomial</td>
<td>663</td>
</tr>
<tr>
<td>kurtosis_noncentral_chi2</td>
<td>639</td>
</tr>
<tr>
<td>kurtosis_noncentral_student_t</td>
<td>635</td>
</tr>
<tr>
<td>kurtosis_normal</td>
<td>632</td>
</tr>
<tr>
<td>kurtosis_pareto</td>
<td>649</td>
</tr>
<tr>
<td>kurtosis_poisson</td>
<td>657</td>
</tr>
<tr>
<td>kurtosis_rayleigh</td>
<td>652</td>
</tr>
<tr>
<td>kurtosis_student_t</td>
<td>633</td>
</tr>
<tr>
<td>kurtosis_weibull</td>
<td>649</td>
</tr>
</tbody>
</table>
L

label (Graphic object) ........................................ 710
label_alignment (draw_graph option) ....................... 797
label_alignment (Graphic option) .......................... 683
label_orientation (Graphic option) ......................... 683
labels (Function) ........................................... 27
labels (System variable) ..................................... 27
lagrange (Function) ........................................ 813
laguerre (Function) .......................................... 881
lambda (Function) ........................................... 548
lambert_w (Function) .......................................... 242
laplace (Function) ........................................... 259
laplacian_matrix (Function) .................................. 785
lassociative (Declaration) ..................................... 105
last (Function) ............................................... 496
lbfgs (Function) ............................................. 827
lbfgs_ncorrections (Variable) ............................... 832
lbfgs_nfeval_max (Variable) ................................. 832
lc_1 (Function) ............................................... 367
lc_u (Function) ............................................... 367
lc2zdk (Function) ............................................ 366
lcharg (Function) ............................................. 938
lc (Function) ................................................ 438
ld (Function) ................................................ 271
ldiag (Function) ............................................... 151
ldisplay (Function) ........................................... 151
legend (Plot option) ......................................... 130
legendre_p (Function) ......................................... 881
legendre_q (Function) ......................................... 881
leinstein (Function) ........................................... 395
length (Function) ............................................. 496
let (Function) ................................................. 479
let_rule_packages (Option variable) ...................... 481
letrat (Option variable) ..................................... 480
letrules (Function) ........................................... 480
letsimp (Function) ............................................ 480
levi_civita (Function) ........................................ 386
lf (Variable) ................................................ 412
lfreeof (Function) ............................................ 87
lg (Variable) ................................................. 412
lgtreillis (Function) ......................................... 451
lhosallim (Option variable) .................................. 249
lacs (Function) ............................................... 292
li (Function) ................................................ 209
liediff (Function) ............................................ 370
limit (Function) ............................................... 249
linsubst (Option variable) .................................. 249
linstated (Function) .......................................... 833
line_graph (Function) ........................................ 775
line_type (Graphic option) ................................. 684
line_width (Graphic option) ................................ 684
linear (Declaration) .......................................... 105
linear (Function) ............................................. 902
linear_program (Function) ................................... 897
linear_solver (Global variable) ............................. 897
linearinterpol (Function) .................................... 815
linechar (Option variable) .................................. 152
linel (Option variable) ...................................... 152
linenum (System variable) ................................... 28
linsolve (Function) ........................................... 293
linsolve_params (Option variable) ......................... 294
linsolvewarn (Option variable) ............................. 294
lispdisp (Option variable) .................................. 152
list_correlations (Function) .................................. 614
list_matrix_entries (Function) .............................. 340
list_nc_monomials (Function) ................................ 353
listarith (Option variable) ................................... 496
listarray (Function) ........................................... 321
listconstvars (Option variable) ............................. 86
listdummyvars (Option variable) ............................ 86
listify (Function) ............................................ 514
listoftens (Function) ......................................... 359
listofvars (Function) .......................................... 86
listp (Function) .............................................. 497, 841
lm (Function) ................................................ 54
lim (Function) ................................................. 54
lmxchar (Option variable) .................................... 340
load (Function) ................................................ 153
load_pathname (System variable) ............................ 153
loadfile (Function) .......................................... 154
loadprint (Option variable) .................................. 154
local (Function) .............................................. 550
locate_matrix_entry (Function) .............................. 841
log (Function) ................................................ 210
log_gamma (Function) ......................................... 226
logabs (Option variable) ..................................... 210
logand (Function) ............................................. 901
logarc (Function) .............................................. 211
logarc (Option variable) ..................................... 211
logcb (Graphic option) ........................................ 684
logconcoefp (Option variable) ............................... 211
logcontract (Function) ........................................ 211
logexpand (Option variable) .................................. 211
lognlint (Option variable) ................................... 211
lognumerator (Option variable) .............................. 212
logor (Function) ............................................... 902
logsimp (Option variable) ..................................... 212
logx (Graphic option) ......................................... 685
logx (Plot option) ............................................. 131
logxor (Function) ............................................. 902
logy (Graphic option) ......................................... 685
logy (Plot option) ............................................. 131
logz (Graphic option) ......................................... 685
lopov (Function) ............................................... 87
lorentz_gauge (Function) ..................................... 376
locovercasep (Function) ....................................... 938
lpart (Function) ............................................... 87
lratsubst (Function) ........................................... 190
lreduce (Function) ............................................ 515
lriem (Variable) ............................................... 412
lriemann (Function) .......................................... 395
lsquares_estimates (Function) ............................... 849
lsquares_estimates_approximate (Function) ............... 849
lsquares_estimates_exact (Function) ....................... 850
lsquares_mse (Function) ...................................... 852
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lsquares_residual_mae</td>
<td>854</td>
</tr>
<tr>
<td>lsquares_residuals</td>
<td>853</td>
</tr>
<tr>
<td>ls (Function)</td>
<td>98</td>
</tr>
<tr>
<td>ltreillis (Function)</td>
<td>451</td>
</tr>
<tr>
<td>lu_backsub (Function)</td>
<td>841</td>
</tr>
<tr>
<td>lu_factor (Function)</td>
<td>841</td>
</tr>
<tr>
<td>max_flow (Function)</td>
<td>786</td>
</tr>
<tr>
<td>max_independent_set (Function)</td>
<td>786</td>
</tr>
<tr>
<td>max_matching (Function)</td>
<td>787</td>
</tr>
<tr>
<td>MAX_0RD (Global variable)</td>
<td>957</td>
</tr>
<tr>
<td>maxapplydepth (Option variable)</td>
<td>106</td>
</tr>
<tr>
<td>maxapplyheight (Option variable)</td>
<td>106</td>
</tr>
<tr>
<td>maxima_temptdir (System variable)</td>
<td>464</td>
</tr>
<tr>
<td>maxima_userdir (System variable)</td>
<td>464</td>
</tr>
<tr>
<td>maximize_lp (Function)</td>
<td>897</td>
</tr>
<tr>
<td>maxnegex (Option variable)</td>
<td>106</td>
</tr>
<tr>
<td>maxposex (Option variable)</td>
<td>106</td>
</tr>
<tr>
<td>maxpsifracdenom (Option variable)</td>
<td>235</td>
</tr>
<tr>
<td>maxpsifracnum (Option variable)</td>
<td>235</td>
</tr>
<tr>
<td>maxpsinegint (Option variable)</td>
<td>235</td>
</tr>
<tr>
<td>maxpsiposint (Option variable)</td>
<td>235</td>
</tr>
<tr>
<td>maxtayorder (Option variable)</td>
<td>420</td>
</tr>
<tr>
<td>maybe (Function)</td>
<td>54</td>
</tr>
<tr>
<td>mean (Function)</td>
<td>604</td>
</tr>
<tr>
<td>mean_bernoulli (Function)</td>
<td>658</td>
</tr>
<tr>
<td>mean_beta (Function)</td>
<td>646</td>
</tr>
<tr>
<td>mean_binomial (Function)</td>
<td>656</td>
</tr>
<tr>
<td>mean_chi2 (Function)</td>
<td>636</td>
</tr>
<tr>
<td>mean_continuous_uniform (Function)</td>
<td>646</td>
</tr>
<tr>
<td>mean_deviation (Function)</td>
<td>608</td>
</tr>
<tr>
<td>mean_discrete_uniform (Function)</td>
<td>660</td>
</tr>
<tr>
<td>mean_exp (Function)</td>
<td>641</td>
</tr>
<tr>
<td>mean_f (Function)</td>
<td>639</td>
</tr>
<tr>
<td>mean_gamma (Function)</td>
<td>644</td>
</tr>
<tr>
<td>mean_hypergeometric (Function)</td>
<td>660</td>
</tr>
<tr>
<td>mean_gumbel (Function)</td>
<td>654</td>
</tr>
<tr>
<td>mean_laplace (Function)</td>
<td>653</td>
</tr>
<tr>
<td>mean_logistic (Function)</td>
<td>647</td>
</tr>
<tr>
<td>mean_lognormal (Function)</td>
<td>643</td>
</tr>
<tr>
<td>mean_negative_binomial (Function)</td>
<td>663</td>
</tr>
<tr>
<td>mean_noncentral_chisquare (Function)</td>
<td>638</td>
</tr>
<tr>
<td>mean_noncentral_student_t (Function)</td>
<td>634</td>
</tr>
<tr>
<td>mean_normal (Function)</td>
<td>632</td>
</tr>
<tr>
<td>mean_pareto (Function)</td>
<td>648</td>
</tr>
<tr>
<td>mean_poisson (Function)</td>
<td>657</td>
</tr>
<tr>
<td>mean_rayleigh (Function)</td>
<td>651</td>
</tr>
<tr>
<td>mean_student_t (Function)</td>
<td>633</td>
</tr>
<tr>
<td>mean_weibull (Function)</td>
<td>649</td>
</tr>
<tr>
<td>median (Function)</td>
<td>608</td>
</tr>
<tr>
<td>median_deviation (Function)</td>
<td>609</td>
</tr>
<tr>
<td>member (Function)</td>
<td>497</td>
</tr>
<tr>
<td>mesh (Graphic object)</td>
<td>711</td>
</tr>
<tr>
<td>mesh_lines_color (Plot option)</td>
<td>131</td>
</tr>
<tr>
<td>method (System variable)</td>
<td>594</td>
</tr>
<tr>
<td>metricexpandall (Function)</td>
<td>953</td>
</tr>
<tr>
<td>min (Function)</td>
<td>55</td>
</tr>
<tr>
<td>min_degree (Function)</td>
<td>787</td>
</tr>
<tr>
<td>min_edge_cut (Function)</td>
<td>787</td>
</tr>
<tr>
<td>min_vertex_cover (Function)</td>
<td>787</td>
</tr>
<tr>
<td>min_vertex_cut (Function)</td>
<td>787</td>
</tr>
<tr>
<td>minf (Constant)</td>
<td>205</td>
</tr>
<tr>
<td>minfactorial (Function)</td>
<td>438</td>
</tr>
<tr>
<td>minimalPoly (Function)</td>
<td>623</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>read_nested_list</td>
<td>867</td>
</tr>
<tr>
<td>read_xpm</td>
<td>720</td>
</tr>
<tr>
<td>readline</td>
<td>936</td>
</tr>
<tr>
<td>readonly</td>
<td>158</td>
</tr>
<tr>
<td>realonly (Option variable)</td>
<td>295</td>
</tr>
<tr>
<td>realpart</td>
<td>95</td>
</tr>
<tr>
<td>realroots</td>
<td>295</td>
</tr>
<tr>
<td>reelarray</td>
<td>324</td>
</tr>
<tr>
<td>rectangle (Graphic object)</td>
<td>717</td>
</tr>
<tr>
<td>rectform</td>
<td>95</td>
</tr>
<tr>
<td>recttopolar (Function)</td>
<td>307</td>
</tr>
<tr>
<td>redef (Function)</td>
<td>370</td>
</tr>
<tr>
<td>redraw (draw_graph option)</td>
<td>799</td>
</tr>
<tr>
<td>reduce_consts (Function)</td>
<td>905</td>
</tr>
<tr>
<td>reduce_order (Function)</td>
<td>909</td>
</tr>
<tr>
<td>refcheck (Option variable)</td>
<td>575</td>
</tr>
<tr>
<td>region (Graphic object)</td>
<td>717</td>
</tr>
<tr>
<td>region_boundaries (Function)</td>
<td>723</td>
</tr>
<tr>
<td>region_boundaries_plus (Function)</td>
<td>723</td>
</tr>
<tr>
<td>rem</td>
<td>471</td>
</tr>
<tr>
<td>remainder (Function)</td>
<td>201</td>
</tr>
<tr>
<td>remarray</td>
<td>324</td>
</tr>
<tr>
<td>rembox</td>
<td>95</td>
</tr>
<tr>
<td>remcomp</td>
<td>364</td>
</tr>
<tr>
<td>remcon</td>
<td>362</td>
</tr>
<tr>
<td>remcoord (Function)</td>
<td>372</td>
</tr>
<tr>
<td>remfun (Function)</td>
<td>314</td>
</tr>
<tr>
<td>remfunction (Function)</td>
<td>30</td>
</tr>
<tr>
<td>remlet (Function)</td>
<td>485</td>
</tr>
<tr>
<td>remove (Function)</td>
<td>471, 472</td>
</tr>
<tr>
<td>remove constvalue (Function)</td>
<td>749</td>
</tr>
<tr>
<td>remove_dimensions (Function)</td>
<td>753</td>
</tr>
<tr>
<td>remove_edge (Function)</td>
<td>794</td>
</tr>
<tr>
<td>remove fundamental dimensions (Function)</td>
<td>753</td>
</tr>
<tr>
<td>remove fundamental units (Function)</td>
<td>754</td>
</tr>
<tr>
<td>remove_vertex (Function)</td>
<td>794</td>
</tr>
<tr>
<td>rempart</td>
<td>901</td>
</tr>
<tr>
<td>remrule (Function)</td>
<td>485</td>
</tr>
<tr>
<td>remsym (Function)</td>
<td>369</td>
</tr>
<tr>
<td>remvalue (Function)</td>
<td>472</td>
</tr>
<tr>
<td>rename (Function)</td>
<td>360</td>
</tr>
<tr>
<td>reset (Function)</td>
<td>30</td>
</tr>
<tr>
<td>residue (Function)</td>
<td>272</td>
</tr>
<tr>
<td>resolvante</td>
<td>453</td>
</tr>
<tr>
<td>resolvante_alterneel (Function)</td>
<td>457</td>
</tr>
<tr>
<td>resolvante_bipartite (Function)</td>
<td>457</td>
</tr>
<tr>
<td>resolvante_diedrale (Function)</td>
<td>457</td>
</tr>
<tr>
<td>resolvante_klein (Function)</td>
<td>457</td>
</tr>
<tr>
<td>resolvante_klein3 (Function)</td>
<td>458</td>
</tr>
<tr>
<td>resolvante_produit_sym (Function)</td>
<td>458</td>
</tr>
<tr>
<td>resolvante_unitaire (Function)</td>
<td>458</td>
</tr>
<tr>
<td>resolvante_vierer (Function)</td>
<td>458</td>
</tr>
<tr>
<td>rest (Function)</td>
<td>499</td>
</tr>
<tr>
<td>resultant (Function)</td>
<td>201</td>
</tr>
<tr>
<td>resultant (Variable)</td>
<td>201</td>
</tr>
<tr>
<td>return (Function)</td>
<td>568</td>
</tr>
<tr>
<td>reveal (Function)</td>
<td>158</td>
</tr>
<tr>
<td>reverse (Function)</td>
<td>499</td>
</tr>
<tr>
<td>revert (Function)</td>
<td>424</td>
</tr>
<tr>
<td>revert2 (Function)</td>
<td>425</td>
</tr>
<tr>
<td>rgb2level (Function)</td>
<td>720</td>
</tr>
<tr>
<td>rhs (Function)</td>
<td>296</td>
</tr>
<tr>
<td>ric (Variable)</td>
<td>412</td>
</tr>
<tr>
<td>ricci (Function)</td>
<td>394</td>
</tr>
<tr>
<td>riemann (Function)</td>
<td>395</td>
</tr>
<tr>
<td>rinv (Function)</td>
<td>412</td>
</tr>
<tr>
<td>risch (Function)</td>
<td>272</td>
</tr>
<tr>
<td>rk (Function)</td>
<td>733</td>
</tr>
<tr>
<td>rmxchar (Option variable)</td>
<td>159</td>
</tr>
<tr>
<td>rncombine (Function)</td>
<td>472</td>
</tr>
<tr>
<td>romberg (Function)</td>
<td>893</td>
</tr>
<tr>
<td>rombergabs (Option variable)</td>
<td>894</td>
</tr>
<tr>
<td>rombergtol (Option variable)</td>
<td>895</td>
</tr>
<tr>
<td>rombergmin (Option variable)</td>
<td>895</td>
</tr>
<tr>
<td>rombergtol (Option variable)</td>
<td>895</td>
</tr>
<tr>
<td>room (Function)</td>
<td>464</td>
</tr>
<tr>
<td>rootscnomode (Option variable)</td>
<td>297</td>
</tr>
<tr>
<td>rootscontract (Function)</td>
<td>297</td>
</tr>
<tr>
<td>rootsepsilon (Option variable)</td>
<td>298</td>
</tr>
<tr>
<td>round (Function)</td>
<td>58</td>
</tr>
<tr>
<td>row (Function)</td>
<td>347</td>
</tr>
<tr>
<td>rowop (Function)</td>
<td>846</td>
</tr>
<tr>
<td>rowswap (Function)</td>
<td>846</td>
</tr>
<tr>
<td>rreduce (Function)</td>
<td>519</td>
</tr>
<tr>
<td>run_testsuite (Function)</td>
<td>7</td>
</tr>
<tr>
<td>run viewer (Plot option)</td>
<td>132</td>
</tr>
</tbody>
</table>

S

save (Function) | 159 |
<p>| savedef (Option variable) | 160 |
| savefactors (Option variable) | 21 |
| saving (Function) | 762 |
| scalarmatrixxp (Option variable) | 347 |
| scalarxp (Function) | 473 |
| scaled_bessel_i (Function) | 224 |
| scaled_bessel_i0 (Function) | 224 |
| scaled_bessel_ii (Function) | 224 |
| scaledfactors (Function) | 347 |
| scanmap (Function) | 568 |
| scatterplot (Function) | 616 |
| schur2comp (Function) | 446 |
| sconc (Function) | 143 |
| scopy (Function) | 939 |
| scsimp (Function) | 109 |
| scvature (Function) | 395 |
| sdowncase (Function) | 939 |
| sec (Function) | 217 |
| sech (Function) | 217 |
| second (Function) | 499 |
| sequal (Function) | 939 |
| sequalignore (Function) | 939 |
| set draw defaults (Function) | 667 |
| set edge weight (Function) | 789 |</p>
<table>
<thead>
<tr>
<th>Function/Variable</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_partitions (Function)</td>
<td>521</td>
</tr>
<tr>
<td>set_plot_option (Function)</td>
<td>128</td>
</tr>
<tr>
<td>set_random_state (Function)</td>
<td>56</td>
</tr>
<tr>
<td>set_tex_environment (Function)</td>
<td>165</td>
</tr>
<tr>
<td>set_tex_environment_default (Function)</td>
<td>166</td>
</tr>
<tr>
<td>set_up_dot_simplifications (Function)</td>
<td>351</td>
</tr>
<tr>
<td>set_vertex_label (Function)</td>
<td>789</td>
</tr>
<tr>
<td>setcheck (Option variable)</td>
<td>576</td>
</tr>
<tr>
<td>setcheckbreak (Option variable)</td>
<td>576</td>
</tr>
<tr>
<td>setdifference (Function)</td>
<td>520</td>
</tr>
<tr>
<td>setelmx (Function)</td>
<td>347</td>
</tr>
<tr>
<td>setequalp (Function)</td>
<td>520</td>
</tr>
<tr>
<td>setp (Function)</td>
<td>521</td>
</tr>
<tr>
<td>setunits (Function)</td>
<td>946</td>
</tr>
<tr>
<td>setup_autoload (Function)</td>
<td>473</td>
</tr>
<tr>
<td>setval (System variable)</td>
<td>576</td>
</tr>
<tr>
<td>seventh (Function)</td>
<td>499</td>
</tr>
<tr>
<td>sexplode (Function)</td>
<td>939</td>
</tr>
<tr>
<td>sf (Function)</td>
<td>417</td>
</tr>
<tr>
<td>shortest_path (Function)</td>
<td>790</td>
</tr>
<tr>
<td>shortest_weighted_path (Function)</td>
<td>790</td>
</tr>
<tr>
<td>show (Function)</td>
<td>160</td>
</tr>
<tr>
<td>show_edge_color (draw_graph option)</td>
<td>799</td>
</tr>
<tr>
<td>show_edge_type (draw_graph option)</td>
<td>799</td>
</tr>
<tr>
<td>show_edge_width (draw_graph option)</td>
<td>799</td>
</tr>
<tr>
<td>show_edges (draw_graph option)</td>
<td>798</td>
</tr>
<tr>
<td>show_id (draw_graph option)</td>
<td>797</td>
</tr>
<tr>
<td>show_label (draw_graph option)</td>
<td>797</td>
</tr>
<tr>
<td>show_vertex_color (draw_graph option)</td>
<td>798</td>
</tr>
<tr>
<td>show_vertex_size (draw_graph option)</td>
<td>798</td>
</tr>
<tr>
<td>show_vertices (draw_graph option)</td>
<td>798</td>
</tr>
<tr>
<td>show_weight (draw_graph option)</td>
<td>797</td>
</tr>
<tr>
<td>showcamps (Function)</td>
<td>364</td>
</tr>
<tr>
<td>showcamps (Function)</td>
<td>364</td>
</tr>
<tr>
<td>showcontract (Function)</td>
<td>160</td>
</tr>
<tr>
<td>showtime (Option variable)</td>
<td>30</td>
</tr>
<tr>
<td>sierpinskia (Function)</td>
<td>765</td>
</tr>
<tr>
<td>sierpinskimap (Function)</td>
<td>767</td>
</tr>
<tr>
<td>sign (Function)</td>
<td>58</td>
</tr>
<tr>
<td>signum (Function)</td>
<td>58</td>
</tr>
<tr>
<td>similaritytransform (Function)</td>
<td>347</td>
</tr>
<tr>
<td>simp (Option variable)</td>
<td>109</td>
</tr>
<tr>
<td>simple_linear_regression (Function)</td>
<td>928</td>
</tr>
<tr>
<td>simplified_output (Global variable)</td>
<td>957</td>
</tr>
<tr>
<td>simplify_products (Option variable)</td>
<td>910</td>
</tr>
<tr>
<td>simplify_sum (Function)</td>
<td>910</td>
</tr>
<tr>
<td>simploge (Function)</td>
<td>940</td>
</tr>
<tr>
<td>simppower (Function)</td>
<td>940</td>
</tr>
<tr>
<td>simptanh (Function)</td>
<td>940</td>
</tr>
<tr>
<td>sin (Function)</td>
<td>347</td>
</tr>
<tr>
<td>sinh (Function)</td>
<td>217</td>
</tr>
<tr>
<td>sinpiflag (Option variable)</td>
<td>314</td>
</tr>
<tr>
<td>sins (Option variable)</td>
<td>317</td>
</tr>
<tr>
<td>sins (Function)</td>
<td>940</td>
</tr>
<tr>
<td>sinvectomy (Function)</td>
<td>940</td>
</tr>
<tr>
<td>sixth (Function)</td>
<td>499</td>
</tr>
<tr>
<td>skewness (Function)</td>
<td>610</td>
</tr>
</tbody>
</table>

- **skewness_bernoulli (Function)**: Page 659
- **skewness_beta (Function)**: Page 646
- **skewness_binomial (Function)**: Page 656
- **skewness_chi2 (Function)**: Page 637
- **skewness_continuous_uniform (Function)**: Page 647
- **skewness_discrete_uniform (Function)**: Page 661
- **skewness_exp (Function)**: Page 642
- **skewness_f (Function)**: Page 640
- **skewness_gama (Function)**: Page 645
- **skewness_geometric (Function)**: Page 660
- **skewness_gumbel (Function)**: Page 655
- **skewness_hypergeometric (Function)**: Page 662
- **skewness_laplace (Function)**: Page 653
- **skewness_logistic (Function)**: Page 648
- **skewness_lognormal (Function)**: Page 644
- **skewness_negative_binomial (Function)**: Page 663
- **skewness_noncentral_chi2 (Function)**: Page 639
- **skewness_noncentral_student_t (Function)**: Page 635
- **small_rhombicosidodecahedron_graph (Function)**: Page 776
- **small_rhombicuboctahedron_graph (Function)**: Page 776
- **smax (Function)**: Page 607
- **smin (Function)**: Page 607
- **smismatch (Function)**: Page 940
- **snowmap (Function)**: Page 767
- **snub_cuboctahedron_graph (Function)**: Page 776
- **snub_dodecahedron_graph (Function)**: Page 776
- **solve (Function)**: Page 298
- **solve_rec (Function)**: Page 910
- **solve_rec_rat (Function)**: Page 911
- **solvedecomposes (Option variable)**: Page 301
- **solveexplicit (Option variable)**: Page 301
- **solvefactors (Option variable)**: Page 301
- **solvefun (Option variable)**: Page 301
- **solvetricvar (Option variable)**: Page 302
- **svalues (Function)**: Page 522
- **some (Function)**: Page 453
- **sort (Function)**: Page 59
- **space (Variable)**: Page 938
- **sparse (Option variable)**: Page 348
- **sparsef6_dec (Function)**: Page 795
- **sparsef6_dec (Function)**: Page 795
- **sparsef6_export (Function)**: Page 795
- **sparsef6_import (Function)**: Page 795
- **specint (Function)**: Page 239
- **spherical (Graphic object)**: Page 717
- **spherical_bessel_j (Function)**: Page 883
<table>
<thead>
<tr>
<th>Function/Variable</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>spherical_bessel_y (Function)</td>
<td>883</td>
</tr>
<tr>
<td>spherical_hankel1 (Function)</td>
<td>883</td>
</tr>
<tr>
<td>spherical_hankel2 (Function)</td>
<td>883</td>
</tr>
<tr>
<td>spherical_harmonic (Function)</td>
<td>884</td>
</tr>
<tr>
<td>spherical_to_xyz (System function)</td>
<td>129</td>
</tr>
<tr>
<td>split (Function)</td>
<td>537</td>
</tr>
<tr>
<td>sposition (Function)</td>
<td>941</td>
</tr>
<tr>
<td>spring_embedding_depth (draw_graph option)</td>
<td>799</td>
</tr>
<tr>
<td>sprint (Function)</td>
<td>936</td>
</tr>
<tr>
<td>sqfr (Function)</td>
<td>201</td>
</tr>
<tr>
<td>sqrt (Function)</td>
<td>60</td>
</tr>
<tr>
<td>sqrtdenest (Function)</td>
<td>906</td>
</tr>
<tr>
<td>sqrtdispflag (Option variable)</td>
<td>60</td>
</tr>
<tr>
<td>sremove (Function)</td>
<td>941</td>
</tr>
<tr>
<td>sremovefirst (Function)</td>
<td>941</td>
</tr>
<tr>
<td>sreverse (Function)</td>
<td>941</td>
</tr>
<tr>
<td>ssearch (Function)</td>
<td>941</td>
</tr>
<tr>
<td>ssort (Function)</td>
<td>942</td>
</tr>
<tr>
<td>status (Function)</td>
<td>464</td>
</tr>
<tr>
<td>ssbest (Function)</td>
<td>942</td>
</tr>
<tr>
<td>ssbestfirst (Function)</td>
<td>942</td>
</tr>
<tr>
<td>staircase (Function)</td>
<td>734</td>
</tr>
<tr>
<td>stdisp (Option variable)</td>
<td>160</td>
</tr>
<tr>
<td>stats_num (Option variable)</td>
<td>915</td>
</tr>
<tr>
<td>status (Function)</td>
<td>465</td>
</tr>
<tr>
<td>std (Function)</td>
<td>605</td>
</tr>
<tr>
<td>std_bernoulli (Function)</td>
<td>658</td>
</tr>
<tr>
<td>std_beta (Function)</td>
<td>646</td>
</tr>
<tr>
<td>std_binomial (Function)</td>
<td>656</td>
</tr>
<tr>
<td>std_chisquare (Function)</td>
<td>637</td>
</tr>
<tr>
<td>std_continuous_uniform (Function)</td>
<td>647</td>
</tr>
<tr>
<td>std_discrete_uniform (Function)</td>
<td>661</td>
</tr>
<tr>
<td>std_exp (Function)</td>
<td>642</td>
</tr>
<tr>
<td>std_f (Function)</td>
<td>640</td>
</tr>
<tr>
<td>std_gamma (Function)</td>
<td>645</td>
</tr>
<tr>
<td>std_geometric (Function)</td>
<td>660</td>
</tr>
<tr>
<td>std_gumbel (Function)</td>
<td>655</td>
</tr>
<tr>
<td>std_hypergeometric (Function)</td>
<td>662</td>
</tr>
<tr>
<td>std_laplace (Function)</td>
<td>653</td>
</tr>
<tr>
<td>std_logistic (Function)</td>
<td>648</td>
</tr>
<tr>
<td>std_lognormal (Function)</td>
<td>644</td>
</tr>
<tr>
<td>std_negative_binomial (Function)</td>
<td>663</td>
</tr>
<tr>
<td>std_noncentral_chisquare (Function)</td>
<td>638</td>
</tr>
<tr>
<td>std_noncentral_student_t (Function)</td>
<td>635</td>
</tr>
<tr>
<td>std_normal (Function)</td>
<td>632</td>
</tr>
<tr>
<td>std_pareto (Function)</td>
<td>648</td>
</tr>
<tr>
<td>std_poisson (Function)</td>
<td>657</td>
</tr>
<tr>
<td>std_rayleigh (Function)</td>
<td>651</td>
</tr>
<tr>
<td>std_student_t (Function)</td>
<td>633</td>
</tr>
<tr>
<td>std_weibull (Function)</td>
<td>649</td>
</tr>
<tr>
<td>std1 (Function)</td>
<td>605</td>
</tr>
<tr>
<td>stirling (Function)</td>
<td>931</td>
</tr>
<tr>
<td>stirling1 (Function)</td>
<td>523</td>
</tr>
<tr>
<td>stirling2 (Function)</td>
<td>524</td>
</tr>
<tr>
<td>strim (Function)</td>
<td>942</td>
</tr>
<tr>
<td>strl (Function)</td>
<td>942</td>
</tr>
<tr>
<td>strm (Function)</td>
<td>942</td>
</tr>
<tr>
<td>strmr (Function)</td>
<td>943</td>
</tr>
<tr>
<td>string (Function)</td>
<td>160</td>
</tr>
<tr>
<td>stringdisp (Option variable)</td>
<td>160</td>
</tr>
<tr>
<td>stringout (Function)</td>
<td>161</td>
</tr>
<tr>
<td>stringp (Function)</td>
<td>938</td>
</tr>
<tr>
<td>strong_components (Function)</td>
<td>790</td>
</tr>
<tr>
<td>structures (Global variable)</td>
<td>529</td>
</tr>
<tr>
<td>struve_h (Function)</td>
<td>238</td>
</tr>
<tr>
<td>struve_l (Function)</td>
<td>238</td>
</tr>
<tr>
<td>style (Plot option)</td>
<td>132</td>
</tr>
<tr>
<td>sublis (Function)</td>
<td>60</td>
</tr>
<tr>
<td>sublis_apply_lambda (Option variable)</td>
<td>60</td>
</tr>
<tr>
<td>sublist (Function)</td>
<td>60</td>
</tr>
<tr>
<td>sublist_indices (Function)</td>
<td>499</td>
</tr>
<tr>
<td>submatrix (Function)</td>
<td>348</td>
</tr>
<tr>
<td>subnumsimp (Option variable)</td>
<td>61</td>
</tr>
<tr>
<td>subsample (Function)</td>
<td>601</td>
</tr>
<tr>
<td>subset (Function)</td>
<td>525</td>
</tr>
<tr>
<td>subseq (Function)</td>
<td>525</td>
</tr>
<tr>
<td>subst (Function)</td>
<td>61</td>
</tr>
<tr>
<td>substpart (Function)</td>
<td>63</td>
</tr>
<tr>
<td>substring (Function)</td>
<td>943</td>
</tr>
<tr>
<td>subvar (Function)</td>
<td>324</td>
</tr>
<tr>
<td>subvarp (Function)</td>
<td>63</td>
</tr>
<tr>
<td>sum (Function)</td>
<td>96</td>
</tr>
<tr>
<td>suncontract (Function)</td>
<td>110</td>
</tr>
<tr>
<td>sunexpand (Option variable)</td>
<td>110</td>
</tr>
<tr>
<td>summand_to_rec (Function)</td>
<td>912</td>
</tr>
<tr>
<td>sumsplitfact (Option variable)</td>
<td>111</td>
</tr>
<tr>
<td>supcase (Function)</td>
<td>943</td>
</tr>
<tr>
<td>supcontext (Function)</td>
<td>178</td>
</tr>
<tr>
<td>surface_hide (Graphic option)</td>
<td>689</td>
</tr>
<tr>
<td>symalp (Function)</td>
<td>63</td>
</tr>
<tr>
<td>sym_difference (Function)</td>
<td>525</td>
</tr>
<tr>
<td>symmetric (Declaration)</td>
<td>111</td>
</tr>
<tr>
<td>symmetricp (Function)</td>
<td>405</td>
</tr>
<tr>
<td>system (Function)</td>
<td>166</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>t (Plot option)</td>
<td>133</td>
</tr>
<tr>
<td>tab (Variable)</td>
<td>938</td>
</tr>
<tr>
<td>take_channel (Function)</td>
<td>721</td>
</tr>
<tr>
<td>take_inference (Function)</td>
<td>914</td>
</tr>
<tr>
<td>tan (Function)</td>
<td>217</td>
</tr>
<tr>
<td>tanh (Function)</td>
<td>217</td>
</tr>
<tr>
<td>taylor (Function)</td>
<td>425</td>
</tr>
<tr>
<td>taylor_logexpand (Option variable)</td>
<td>429</td>
</tr>
<tr>
<td>taylor_order_coefficients (Option variable)</td>
<td>429</td>
</tr>
<tr>
<td>taylor_simplifier (Function)</td>
<td>429</td>
</tr>
<tr>
<td>taylor_truncate_polynomials (Option variable)</td>
<td>429</td>
</tr>
<tr>
<td>taylordepth (Option variable)</td>
<td>428</td>
</tr>
<tr>
<td>taylorinfo (Function)</td>
<td>428</td>
</tr>
<tr>
<td>taylorlp (Function)</td>
<td>429</td>
</tr>
<tr>
<td>taytort (Function)</td>
<td>430</td>
</tr>
</tbody>
</table>
Appendix A: Function and Variable Index

uniteigenvectors (Function) .................................. 348
uniteigenvectors (Option variable) ...................... 348
units (Function) .................................................. 750
univector (Function) ............................................. 349
unknown (Function) .............................................. 111
unless (Special operator) .................................... 569
unordered (Function) .......................................... 99
unsum (Function) ................................................ 430
untellrat (Function) ............................................ 203
untimer (Function) .............................................. 577
untrace (Function) .............................................. 579
uppercasep (Function) ........................................ 938
uric (Variable) .................................................. 412
urici (Function) ................................................ 395
urie (Variable) .................................................. 412
urieumann (Function) ......................................... 396
use_fast_arrays (Option variable) ....................... 325
user_preamble (Graphic option) ......................... 693
usersetunits (Optional variable) ......................... 950
uvect (Function) .............................................. 349

V
values (System variable) ...................................... 31
vandermonde_matrix (Function) ......................... 847
var (Function) .................................................. 604
var_beroulli (Function) ..................................... 658
var_beta (Function) ............................................ 646
var_binomial (Function) ..................................... 656
var_chisq (Function) ......................................... 637
var_continuous_uniform (Function) ...................... 647
var_discrete_uniform (Function) ......................... 661
var_exp (Function) ............................................ 641
var_f (Function) ................................................ 640
var_gamma (Function) ........................................ 645
var_geometric (Function) .................................... 660
var_gumbel (Function) ....................................... 655
var_hypergeometric (Function) ............................ 662
var_laplace (Function) ....................................... 653
var_logistic (Function) ...................................... 647
var_lognormal (Function) .................................... 644
var_negative_binomial (Function) ....................... 663
var_noncentral_chisq (Function) ......................... 638
var_noncentral_student_t (Function) .................... 634
var_normal (Function) ........................................ 632
var_pareto (Function) ........................................ 648
var_poisson (Function) ....................................... 657
var_rayleigh (Function) ...................................... 651
var_student_t (Function) .................................... 633
var_weibull (Function) ....................................... 649
var1 (Function) ................................................. 605
vect_cross (Option variable) .............................. 350
vector (Graphic object) ...................................... 718
vectorpotential (Function) .................................. 63
vectorsimp (Function) ....................................... 349
verify (Function) ............................................. 999
verbose (Option variable) .................................. 430
vers (Function) ............................................... 903
vertex_color (draw_graph_option) ....................... 798
vertex_coloring (draw_graph_option) ................... 798
vertex_coloring (Function) ................................. 792
vertex_connectivity (Function) ......................... 790
vertex_degree (Function) .................................... 791
vertex_distance (Function) .................................. 791
vertex_eccentricity (Function) ......................... 791
vertex_in_degree (Function) ............................... 791
vertex_out_degree (Function) .............................. 791
vertex_partition (draw_graph_option) .................. 798
vertex_size (draw_graph_option) ......................... 798
vertex_type (draw_graph_option) ......................... 798
vertices (Function) .......................................... 792
vertices_to_cycle (Function) ............................. 800
vertices_to_path (Function) .............................. 800
view (Graphic option) ....................................... 693

W
warnings (Global variable) ................................ 957
weyl (Function) ............................................... 396
weyl (Variable) ................................................ 412
wheel_graph (Function) ...................................... 777
while (Special operator) .................................... 569
wiener_index (Function) ..................................... 792
with_stdout (Function) ...................................... 167
write_binary_data (Function) .............................. 869
write_data (Function) ........................................ 867
writefile (Function) ......................................... 167
wronskian (Function) ....................................... 901

X
x (Plot option) .................................................. 133
x_voxel (Graphic option) ..................................... 693
xaxis (Graphic option) ....................................... 693
xaxis_color (Graphic option) ............................... 694
xaxis_secondary (Graphic option) ......................... 694
xaxis_type (Graphic option) ............................... 694
xaxis_width (Graphic option) .............................. 695
xlabel (Graphic option) ..................................... 695
xlabel (Plot option) ......................................... 133
xrange (Graphic option) ..................................... 695
xrange_secondary (Graphic option) ...................... 696
xreduce (Function) .......................................... 527
xthru (Function) ............................................. 63
xtics (Graphic option) ....................................... 696
xtics_axis (Graphic option) ............................... 697
xtics_rotate (Graphic option) ............................. 697
xtics_rotate_secondary (Graphic option) ............... 697
xtics_secondary (Graphic option) ......................... 697
xtics_secondary_axis (Graphic option) ................. 697
xu_grid (Graphic option) .................................... 698
xy_file (Graphic option) ..................................... 698
xplane (Graphic option) ..................................... 698
<table>
<thead>
<tr>
<th>Option</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>y (Plot option)</td>
<td>133</td>
</tr>
<tr>
<td>y_voxel (Graphic option)</td>
<td>702</td>
</tr>
<tr>
<td>yaxis (Graphic option)</td>
<td>702</td>
</tr>
<tr>
<td>yaxis_color (Graphic option)</td>
<td>703</td>
</tr>
<tr>
<td>yaxis_secondary (Graphic option)</td>
<td>703</td>
</tr>
<tr>
<td>yaxis_type (Graphic option)</td>
<td>703</td>
</tr>
<tr>
<td>yaxis_width (Graphic option)</td>
<td>703</td>
</tr>
<tr>
<td>ylabel (Graphic option)</td>
<td>703</td>
</tr>
<tr>
<td>ylabel (Plot option)</td>
<td>703</td>
</tr>
<tr>
<td>yrange (Graphic option)</td>
<td>703</td>
</tr>
<tr>
<td>yrange_secondary (Graphic option)</td>
<td>703</td>
</tr>
<tr>
<td>ytics (Graphic option)</td>
<td>703</td>
</tr>
<tr>
<td>ytics_axis (Graphic option)</td>
<td>703</td>
</tr>
<tr>
<td>ytics_rotate (Graphic option)</td>
<td>703</td>
</tr>
<tr>
<td>ytics_rotate_secondary (Graphic option)</td>
<td>703</td>
</tr>
<tr>
<td>ytics_secondary (Graphic option)</td>
<td>703</td>
</tr>
<tr>
<td>ytics_secondary_axis (Graphic option)</td>
<td>703</td>
</tr>
<tr>
<td>yv_grid (Graphic option)</td>
<td>704</td>
</tr>
<tr>
<td>z (Plot option)</td>
<td>134</td>
</tr>
<tr>
<td>z_voxel (Graphic option)</td>
<td>704</td>
</tr>
<tr>
<td>zaxis (Graphic option)</td>
<td>704</td>
</tr>
<tr>
<td>zaxis_color (Graphic option)</td>
<td>704</td>
</tr>
<tr>
<td>zaxis_type (Graphic option)</td>
<td>704</td>
</tr>
<tr>
<td>zaxis_width (Graphic option)</td>
<td>704</td>
</tr>
<tr>
<td>Zeilberger (Function)</td>
<td>704</td>
</tr>
<tr>
<td>zeroa (Constant)</td>
<td>704</td>
</tr>
<tr>
<td>zerobern (Option variable)</td>
<td>704</td>
</tr>
<tr>
<td>zeroequiv (Function)</td>
<td>704</td>
</tr>
<tr>
<td>zerofor (Function)</td>
<td>704</td>
</tr>
<tr>
<td>zeromatrix (Function)</td>
<td>704</td>
</tr>
<tr>
<td>zeromatrixp (Function)</td>
<td>704</td>
</tr>
<tr>
<td>zeta (Function)</td>
<td>704</td>
</tr>
<tr>
<td>zeta%pi (Option variable)</td>
<td>704</td>
</tr>
<tr>
<td>zlabel (Graphic option)</td>
<td>704</td>
</tr>
<tr>
<td>zlabel (Plot option)</td>
<td>704</td>
</tr>
<tr>
<td>zlange (Function)</td>
<td>704</td>
</tr>
<tr>
<td>zrange (Graphic option)</td>
<td>704</td>
</tr>
<tr>
<td>ztics (Graphic option)</td>
<td>704</td>
</tr>
<tr>
<td>ztics_axis (Graphic option)</td>
<td>704</td>
</tr>
<tr>
<td>ztics_rotate (Graphic option)</td>
<td>704</td>
</tr>
</tbody>
</table>