Maxima Manual

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

Table of Contents

1 Introduction to Maxima

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

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

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

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

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

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

```
Examples:

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

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

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

(%o1)                                true
```

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

```
(%i1) u: expand ((x + y)^6);
       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)
```

Maxima knows about complex numbers and numerical constants:

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

Maxima can do differential and integral calculus:

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

Maxima can solve linear systems and cubic equations:

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

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

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

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

Maxima can generate plots of one or more functions:

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

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

```
                                      2                    2
(%i4) kill(labels);
(%o0)                           done
(%i1) plot2d (sin(x)/x, [x, -20, 20]);
(%o1)
(%i2) plot2d ([atan(x), erf(x), tanh(x)], [x, -5, 5]);
(%o2)
(%i3) plot3d (sin(sqrt(x^2 + y^2))/sqrt(x^2 + y^2), [x, -12, 12],
      [y, -12, 12]);
(%o3)
```

2 Bug Detection and Reporting

2.1 Functions and Variables for Bug Detection and Reporting

run_testsuite () Function
run_testsuite (*boolean*) Function
run_testsuite (*boolean, boolean*) Function
run_testsuite (*boolean, boolean, list*) Function

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

`run_testsuite ()` displays only tests that do not pass.

`run_testsuite (true)` displays tests that are marked as known bugs, as well as failures.

`run_testsuite (true, true)` displays all tests.

If the optional third argument is given, a subset of the tests is run. The subset of the tests to run is given as a list of the names of the tests. The complete set of tests is specified by `testsuite_files`.

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

`run_testsuite` returns `done`.

testsuite_files Option variable

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

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

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

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

bug_report () Function

Prints out Maxima and Lisp version numbers, and gives a link to the Maxima project bug report web page. The version information is the same as reported by `build_info`.

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

`bug_report` returns an empty string "".

build_info () Function

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

`build_info` returns an empty string "".

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, while Lisp is not. 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 `$Foo` all correspond to Maxima `foo`.
2. A Lisp identifier which is all uppercase or all lowercase and enclosed in vertical bars corresponds to a Maxima identifier with case reversed. That is, uppercase is changed to lowercase and lowercase to uppercase. E.g., Lisp `|$FOO|` and `|$foo|` correspond to Maxima `foo` and `FOO`, respectively.
3. A Lisp identifier which is mixed uppercase and lowercase and enclosed in vertical bars corresponds to a Maxima identifier with the same case. E.g., Lisp `|$Foo|` corresponds to Maxima `Foo`.

The `#$` Lisp macro allows the use of Maxima expressions in Lisp code. `#$`*expr*`$` expands to a Lisp expression equivalent to the Maxima expression *expr*.

```
(msetq $foo #$[x, y]$)
```

This has the same effect as entering

```
(%i1) foo: [x, y];
```

The Lisp function `displa` prints an expression in Maxima format.

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

Functions defined in Maxima are not ordinary Lisp functions. The Lisp function `mfuncall` calls a Maxima function. For example:

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

Some Lisp functions are shadowed in the Maxima package, namely the following.

`complement`, `continue`, `//`, `float`, `functionp`, `array`, `exp`, `listen`, `signum`, `atan`, `asin`, `acos`, `asinh`, `acosh`, `atanh`, `tanh`, `cosh`, `sinh`, `tan`, `break`, and `gcd`.

3.2 Garbage Collection

Symbolic computation tends to create a good deal of garbage, 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.

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 `?` command (i.e., the `describe` function). The user's manual is viewed as `info` hypertext by the `info` viewer program and as a web page by any ordinary web browser.

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

```
(%i1) example (integrate);
```

yields

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

                                          ratsimp(f - diff(u, x)))
(%i3) test(sin(x))
(%o3)                          0
(%i4) test(1/(x+1))
(%o4)                          0
(%i5) test(1/(x^2+1))
(%o5)                          0
```

and additional output.

3.4 Functions and Variables for Help

demo (*filename*) Function

Evaluates Maxima expressions in *filename* and displays the results. `demo` pauses after evaluating each expression and continues after the user enters a carriage return. (If running in Xmaxima, `demo` may need to see a semicolon `;` followed by a carriage return.)

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

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

Example:

```
(%i1) demo ("disol");

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


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


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

(%t5)                          g + f

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


_
(%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)

_
                             z         x
(%i7)               exp1 : ----- + ---------
                            y + x   2 (y + x)
                           z         x
(%o7)                     ----- + ---------
                          y + x   2 (y + x)


_
(%i8)                      combine(exp1)
                           z         x
```

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

_
(%i9)                      rncombine(%)
                              2 z + x
(%o9)                         ---------
                              2 (y + x)

_
                                 d   c   b   a
(%i10)                 exp2 : - + - + - + -
                                 3   3   2   2
                              d   c   b   a
(%o10)                        - + - + - + -
                              3   3   2   2

_
(%i11)                          combine(exp2)
                       2 d + 2 c + 3 (b + a)
(%o11)                 ---------------------
                                 6

_
(%i12)                        rncombine(exp2)
                       2 d + 2 c + 3 b + 3 a
(%o12)                 ---------------------
                                 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 8.

Example:

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

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

 -- Option variable: integrate_use_rootsof
     Default value: 'false'

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

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

example (*topic*) Function
example () Function

`example (`*topic*`)` displays some examples of *topic*, which is a symbol (not a string). 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 variable `manual_demo`, which defaults to `"manual.demo"`.

`example` quotes its argument. `example` returns `done` unless there is an error 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 (coeff);
(%i4) coeff(b+tan(x)+2*a*tan(x) = 3+5*tan(x),tan(x))
(%o4)                        2 a + 1 = 5
(%i5) coeff(1+x*%e^x+y,x,0)
(%o5)                           y + 1
(%o5)                           done
```

4 Command Line

4.1 Introduction to Command Line

'

Operator

The single quote operator ' prevents evaluation.

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

Applied to a function call, the single quote prevents evaluation of the function call, although the arguments of the function are still evaluated (if evaluation is not otherwise prevented). The result is the noun form of the function call.

Applied to a parenthesized expression, the single quote prevents evaluation of all symbols and function calls in the expression. E.g., `'(f(x))` means do not evaluate the expression `f(x)`. `'f(x)` (with the single quote applied to `f` instead of `f(x)`) means return the noun form of `f` applied to `[x]`.

The single quote does not prevent simplification.

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

See also the quote-quote operator '' and `nouns`.

Examples:

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

```
(%i1) aa: 1024;
(%o1)                          1024
(%i2) aa^2;
(%o2)                         1048576
(%i3) 'aa^2;
                                 2
(%o3)                           aa
(%i4) ''%;
(%o4)                         1048576
```

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

```
(%i1) x0: 5;
(%o1)                            5
(%i2) x1: 7;
(%o2)                            7
(%i3) integrate (x^2, x, x0, x1);
                                218
(%o3)                           ---
                                 3
(%i4) 'integrate (x^2, x, x0, x1);
                               7
                              /
                              [   2
(%o4)                         I  x  dx
```

```
                                ]
                                /
                                 5
(%i5) %, nouns;
                                   218
(%o5)                              ---
                                    3
```

Applied to a parenthesized expression, the single quote prevents evaluation of all symbols and function calls in the expression.

```
(%i1) aa: 1024;
(%o1)                          1024
(%i2) bb: 19;
(%o2)                           19
(%i3) sqrt(aa) + bb;
(%o3)                           51
(%i4) '(sqrt(aa) + bb);
(%o4)                     bb + sqrt(aa)
(%i5) ''%;
(%o5)                           51
```

The single quote does not prevent simplification.

```
(%i1) sin (17 * %pi) + cos (17 * %pi);
(%o1)                           - 1
(%i2) '(sin (17 * %pi) + cos (17 * %pi));
(%o2)                           - 1
```

”

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.

```
(%i1) expand ((a + b)^3);
                     3        2      2      3
(%o1)               b  + 3 a b  + 3 a  b + a
```

```
(%i2) [_, ''_];
                          3    3        2      2      3
(%o2)     [expand((b + a) ), b  + 3 a b  + 3 a  b + a ]
(%i3) [%i1, ''%i1];
                          3    3        2      2      3
(%o3)     [expand((b + a) ), b  + 3 a b  + 3 a  b + a ]
(%i4) [aa : cc, bb : dd, cc : 17, dd : 29];
(%o4)                     [cc, dd, 17, 29]
(%i5) foo_1 (x) := aa - bb * x;
(%o5)                  foo_1(x) := aa - bb x
(%i6) foo_1 (10);
(%o6)                        cc - 10 dd
(%i7) ''%;
(%o7)                           - 273
(%i8) ''(foo_1 (10));
(%o8)                           - 273
(%i9) foo_2 (x) := ''aa - ''bb * x;
(%o9)                  foo_2(x) := cc - dd x
(%i10) foo_2 (10);
(%o10)                          - 273
(%i11) [x0 : x1, x1 : x2, x2 : x3];
(%o11)                       [x1, x2, x3]
(%i12) x0;
(%o12)                           x1
(%i13) ''x0;
(%o13)                           x2
(%i14) '' ''x0;
(%o14)                           x3
```

Applied to the operator of an expression, quote-quote changes the operator from a noun to a verb (if it is not already a verb).

```
(%i1) sin (1);
(%o1)                          sin(1)
(%i2) ''sin (1);
(%o2)                     0.8414709848079
(%i3) declare (foo, noun);
(%o3)                           done
(%i4) foo (x) := x - 1729;
(%o4)                  ''foo(x) := x - 1729
(%i5) foo (100);
(%o5)                         foo(100)
(%i6) ''foo (100);
(%o6)                           - 1629
```

The quote-quote operator is applied by the input parser; it is not stored as part of a parsed input expression.

```
(%i1) [aa : bb, cc : dd, bb : 1234, dd : 5678];
(%o1)                  [bb, dd, 1234, 5678]
(%i2) aa + cc;
(%o2)                           dd + bb
```

```
(%i3) display (_, op (_), args (_));
                           _ = cc + aa

                        op(cc + aa) = +

                     args(cc + aa) = [cc, aa]

(%o3)                           done
(%i4) ''(aa + cc);
(%o4)                           6912
(%i5) display (_, op (_), args (_));
                           _ = dd + bb

                        op(dd + bb) = +

                     args(dd + bb) = [dd, bb]

(%o5)                           done
```

Quote-quote causes evaluation when evaluation is otherwise suppressed, such as in function definitions, lambda expressions, and expressions quoted by single quote '.

```
(%i1) foo_1a (x) := ''(integrate (log (x), x));
(%o1)               foo_1a(x) := x log(x) - x
(%i2) foo_1b (x) := integrate (log (x), x);
(%o2)           foo_1b(x) := integrate(log(x), x)
(%i3) dispfun (foo_1a, foo_1b);
(%t3)               foo_1a(x) := x log(x) - x

(%t4)           foo_1b(x) := integrate(log(x), x)

(%o4)                         [%t3, %t4]
(%i4) integrate (log (x), x);
(%o4)                        x log(x) - x
(%i5) foo_2a (x) := ''%;
(%o5)               foo_2a(x) := x log(x) - x
(%i6) foo_2b (x) := %;
(%o6)                     foo_2b(x) := %
(%i7) dispfun (foo_2a, foo_2b);
(%t7)               foo_2a(x) := x log(x) - x

(%t8)                     foo_2b(x) := %

(%o8)                       [%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], cos(u))
(%i10) '(sum (a[k], k, 1, 3) + sum (b[k], k, 1, 3));
(%o10)         sum(b , k, 1, 3) + sum(a , k, 1, 3)
                     k                  k
```

```
(%i11) ’(’’(sum (a[k], k, 1, 3)) + ’’(sum (b[k], k, 1, 3)));
(%o11)             b  + a  + b  + a  + b  + a
                    3    3    2    2    1    1
```

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 determinant kept outside of the inverse rather than dividing through each element.
   - `diff` causes all differentiations indicated in *expr* to be performed.
   - `derivlist (`*x*`,` *y*`,` *z*`, ...)` causes only differentiations with respect to the indicated variables.
   - `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 numervals 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`, `float`, and `pred` 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 `eval` in the arguments, steps (3) and (4) are repeated.

Examples

```
(%i1) sin(x) + cos(y) + (w+1)^2 + 'diff (sin(w), w);
                                      d                    2
(%o1)              cos(y) + sin(x) + -- (sin(w)) + (w + 1)
                                      dw
(%i2) ev (%, sin, expand, diff, x=2, y=1);
                             2
(%o2)           cos(w) + w  + 2 w + cos(1) + 1.909297426825682
```

An alternate top level syntax has been provided for `ev`, whereby one may just type in its arguments, without the `ev()`. That is, one may write simply

expr, *arg_1*, ..., *arg_n*

This is not permitted as part of another expression, e.g., in functions, blocks, etc.

Notice the parallel binding process in the following example.

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

                                         1
(%t7)                              y = - -
                                         5

                                        6
(%t8)                             x = -
                                        5
(%o8)                           [[%t7, %t8]]
(%i8) %o6, %o8;
(%o8)                             - 4 = - 4
(%i9) x + 1/x > gamma (1/2);
                                    1
(%o9)                         x + - > sqrt(%pi)
                                    x
(%i10) %, numer, x=1/2;
(%o10)                    2.5 > 1.772453850905516
(%i11) %, pred;
(%o11)                                 true
```

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`, `exponentiate`, `exptisolate`, `factorflag`, `float`, `halfangles`, `infeval`, `isolate_wrt_times`, `keepfloat`, `letrat`, `listarith`, `logabs`, `logarc`, `logexpand`, `lognegint`, `lognumer`, `m1pbranch`, `numer_pbranch`, `programmode`, `radexpand`, `ratalgdenom`, `ratfac`, `ratmx`, `ratsimpexpons`, `simp`, `simpsum`, `sumexpand`, and `trigexpand`.

Examples:

```
(%i1) sin (1/2);
                                 1
(%o1)                        sin(-)
                                 2
(%i2) sin (1/2), float;
(%o2)                   0.479425538604203
(%i3) sin (1/2), float=true;
(%o3)                   0.479425538604203
(%i4) simp : false;
(%o4)                         false
(%i5) 1 + 1;
(%o5)                         1 + 1
(%i6) 1 + 1, simp;
(%o6)                           2
(%i7) simp : true;
(%o7)                         true
(%i8) sum (1/k^2, k, 1, inf);
                             inf
                             ====
                             \     1
(%o8)                         >    --
                             /      2
                             ====  k
                             k = 1
(%i9) sum (1/k^2, k, 1, inf), simpsum;
                                 2
                               %pi
(%o9)                          ----
                                6
(%i10) declare (aa, evflag);
(%o10)                        done
(%i11) if aa = true then YES else NO;
(%o11)                         NO
(%i12) if aa = true then YES else NO, aa;
(%o12)                         YES
```

evfun Property

When a function F has the `evfun` property, the expressions `ev(`*expr*`, `F`)` and *expr*`, `F (at the interactive prompt) are equivalent to F`(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;
                              3
(%o1)                        x  - 1
(%i2) x^3 - 1, factor;
                                2
(%o2)                  (x - 1) (x  + x + 1)
(%i3) factor (x^3 - 1);
                                2
(%o3)                  (x - 1) (x  + x + 1)
(%i4) cos(4 * x) / sin(x)^4;
                             cos(4 x)
(%o4)                        --------
                                4
                             sin (x)
(%i5) cos(4 * x) / sin(x)^4, trigexpand;
                 4           2       2         4
              sin (x) - 6 cos (x) sin (x) + cos (x)
(%o5)         ------------------------------------
                                4
                             sin (x)
(%i6) cos(4 * x) / sin(x)^4, trigexpand, ratexpand;
                           2         4
                      6 cos (x)   cos (x)
(%o6)               - --------- + ------- + 1
                           2          4
                       sin (x)    sin (x)
(%i7) ratexpand (trigexpand (cos(4 * x) / sin(x)^4));
                           2         4
                      6 cos (x)   cos (x)
(%o7)               - --------- + ------- + 1
                           2          4
                       sin (x)    sin (x)
(%i8) declare ([F, G], evfun);
(%o8)                         done
(%i9) (aa : bb, bb : cc, cc : dd);
(%o9)                          dd
(%i10) aa;
(%o10)                         bb
```

```
(%i11) aa, F;
(%o11)                          F(cc)
(%i12) F (aa);
(%o12)                          F(bb)
(%i13) F (ev (aa));
(%o13)                          F(cc)
(%i14) aa, F, G;
(%o14)                        G(F(cc))
(%i15) G (F (ev (aa)));
(%o15)                        G(F(cc))
```

infeval Option variable

Enables "infinite evaluation" mode. `ev` repeatedly evaluates an expression until it stops changing. To prevent a variable, say `X`, from being evaluated away in this mode, simply include `X='X` as an argument to `ev`. Of course expressions such as `ev (X, X=X+1, infeval)` will generate an infinite loop.

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

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

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

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

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

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

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

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

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

`kill (`*symbol*`)` unbinds all properties of *symbol*. In contrast, `remvalue`, `remfunction`, `remarray`, and `remrule` unbind a specific property.

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

labels (*symbol*) Function
labels System variable

Returns the list of input, output, or intermediate expression labels which begin with *symbol*. Typically *symbol* is the value of `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 *symbol*, `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.

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

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

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

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

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

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

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

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

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

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

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

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

printprops (*a*, *i*) Function
printprops ([*a_1*, ..., *a_n*], *i*) Function
printprops (*all*, *i*) Function

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`.

prompt Option variable

Default value: `_`

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

quit () Function

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

To stop a lengthy computation, type `control-C`. The default action is to return to the Maxima prompt. If `*debugger-hook*` is `nil`, `control-C` opens the Lisp debugger. See also `debugging`.

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

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.

reset () Function

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

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

showtime Option variable

Default value: `false`

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

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

See also `timer`.

sstatus (*feature*, *package*) Function

Sets the status of *feature* in *package*. After `sstatus (`*feature*`,` *package*`)` is executed, `status (`*feature*`,` *package*`)` returns `true`. This can be useful for package writers, to keep track of what features they have loaded in.

to_lisp () Function

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

values System variable

Initial value: `[]`

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

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

`+` Operator
`-` Operator
`*` Operator
`/` Operator
`^` Operator

The symbols `+ * /` and `^` represent addition, multiplication, division, and exponentiation, respectively. The names of these operators are `"+" "*" "/"` and `"^"`, which may appear where the name of a function or operator is required.

The symbols `+` and `-` represent unary addition and negation, respectively, and the names of these operators are `"+"` and `"-"`, respectively.

Subtraction `a - b` is represented within Maxima as addition, `a + (- b)`. Expressions such as `a + (- b)` are displayed as subtraction. Maxima recognizes `"-"` only as the name of the unary negation operator, and not as the name of the binary subtraction operator.

Division `a / b` is represented within Maxima as multiplication, `a * b^(- 1)`. Expressions such as `a * b^(- 1)` are displayed as division. Maxima recognizes `"/"` as the name of the division operator.

Addition and multiplication are n-ary, commutative operators. Division and exponentiation are binary, noncommutative operators.

Maxima sorts the operands of commutative operators to construct a canonical representation. For internal storage, the ordering is determined by `orderlessp`. For display, the ordering for addition is determined by `ordergreatp`, and for multiplication, it is the same as the internal ordering.

Arithmetic computations are carried out on literal numbers (integers, rationals, ordinary floats, and bigfloats). Except for exponentiation, all arithmetic operations on numbers are simplified to numbers. Exponentiation is simplified to a number if either operand is an ordinary float or bigfloat or if the result is an exact integer or rational; otherwise an exponentiation may be simplified to `sqrt` or another exponentiation or left unchanged.

Floating-point contagion applies to arithmetic computations: if any operand is a bigfloat, the result is a bigfloat; otherwise, if any operand is an ordinary float, the result is an ordinary float; otherwise, the operands are rationals or integers and the result is a rational or integer.

Arithmetic computations are a simplification, not an evaluation. Thus arithmetic is carried out in quoted (but simplified) expressions.

Arithmetic operations are applied element-by-element to lists when the global flag `listarith` is `true`, and always applied element-by-element to matrices. When one operand is a list or matrix and another is an operand of some other type, the other operand is combined with each of the elements of the list or matrix.

Examples:

Addition and multiplication are n-ary, commutative operators. Maxima sorts the operands to construct a canonical representation. The names of these operators are `"+"` and `"*"`.

```
(%i1) c + g + d + a + b + e + f;
(%o1)                g + f + e + d + c + b + a
(%i2) [op (%), args (%)];
(%o2)               [+, [g, f, e, d, c, b, a]]
(%i3) c * g * d * a * b * e * f;
(%o3)                       a b c d e f g
(%i4) [op (%), args (%)];
(%o4)               [*, [a, b, c, d, e, f, g]]
(%i5) apply ("+", [a, 8, x, 2, 9, x, x, a]);
(%o5)                     3 x + 2 a + 19
```

```
(%i6) apply ("*", [a, 8, x, 2, 9, x, x, a]);
                                  2  3
(%o6)                        144 a  x
```

Division and exponentiation are binary, noncommutative operators. The names of these operators are "/" and "^".

```
(%i1) [a / b, a ^ b];
                                 a   b
(%o1)                           [-, a ]
                                 b
(%i2) [map (op, %), map (args, %)];
(%o2)               [[/, ^], [[a, b], [a, b]]]
(%i3) [apply ("/", [a, b]), apply ("^", [a, b])];
                                 a   b
(%o3)                           [-, a ]
                                 b
```

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

```
(%i1) [inpart (a - b, 0), inpart (a - b, 1), inpart (a - b, 2)];
(%o1)                         [+, a, - b]
(%i2) [inpart (a / b, 0), inpart (a / b, 1), inpart (a / b, 2)];
                                    1
(%o2)                         [*, a, -]
                                    b
```

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

```
(%i1) 17 + b - (1/2)*29 + 11^(2/4);
                                        5
(%o1)                   b + sqrt(11) + -
                                        2
(%i2) [17 + 29, 17 + 29.0, 17 + 29b0];
(%o2)                    [46, 46.0, 4.6b1]
```

Arithmetic computations are a simplification, not an evaluation.

```
(%i1) simp : false;
(%o1)                           false
(%i2) '(17 + 29*11/7 - 5^3);
                                29 11    3
(%o2)                    17 + ----- - 5
                                  7
(%i3) simp : true;
(%o3)                           true
(%i4) '(17 + 29*11/7 - 5^3);
                                  437
(%o4)                           - ---
                                    7
```

Arithmetic is carried out element-by-element for lists (depending on `listarith`) and matrices.

```
(%i1) matrix ([a, x], [h, u]) - matrix ([1, 2], [3, 4]);
                          [ a - 1  x - 2 ]
```

```
(%o1)                    [             ]
                         [ h - 3  u - 4 ]
(%i2) 5 * matrix ([a, x], [h, u]);
                           [ 5 a  5 x ]
(%o2)                      [          ]
                           [ 5 h  5 u ]
(%i3) listarith : false;
(%o3)                          false
(%i4) [a, c, m, t] / [1, 7, 2, 9];
                           [a, c, m, t]
(%o4)                      ------------
                           [1, 7, 2, 9]
(%i5) [a, c, m, t] ^ x;
                                      x
(%o5)                      [a, c, m, t]
(%i6) listarith : true;
(%o6)                          true
(%i7) [a, c, m, t] / [1, 7, 2, 9];
                               c  m  t
(%o7)                      [a, -, -, -]
                               7  2  9
(%i8) [a, c, m, t] ^ x;
                            x   x   x   x
(%o8)                    [a , c , m , t ]
```

**** Operator

Exponentiation operator. Maxima recognizes `**` as the same operator as `^` in input, and it is displayed as `^` in 1-dimensional output, or by placing the exponent as a superscript in 2-dimensional output.

The `fortran` function displays the exponentiation operator as `**`, whether it was input as `**` or `^`.

Examples:

```
(%i1) is (a**b = a^b);
(%o1)                          true
(%i2) x**y + x^z;
                               z    y
(%o2)                         x  + x
(%i3) string (x**y + x^z);
(%o3)                         x^z+x^y
(%i4) fortran (x**y + x^z);
      x**z+x**y
(%o4)                          done
```

5.6 Relational operators

< Operator
<= Operator
>= Operator
> Operator

The symbols `<` `<=` `>=` and `>` represent less than, less than or equal, greater than or equal, and greater than, respectively. The names of these operators are `"<"` `"<="` `">="` and `">"`, which may appear where the name of a function or operator is required.

These relational operators are all binary operators; constructs such as `a < b < c` are not recognized by Maxima.

Relational expressions are evaluated to Boolean values by the functions `is` and `maybe`, and the programming constructs `if`, `while`, and `unless`. Relational expressions are not otherwise evaluated or simplified to Boolean values, although the arguments of relational expressions are evaluated (when evaluation is not otherwise prevented by quotation).

When a relational expression cannot be evaluated to `true` or `false`, the behavior of `is` and `if` are governed by the global flag `prederror`. When `prederror` is `true`, `is` and `if` trigger an error. When `prederror` is `false`, `is` returns `unknown`, and `if` returns a partially-evaluated conditional expression.

`maybe` always behaves as if `prederror` were `false`, and `while` and `unless` always behave as if `prederror` were `true`.

Relational operators do not distribute over lists or other aggregates.

See also `= # equal` and `notequal`.

Examples:

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

```
(%i1) [x, y, z] : [123, 456, 789];
(%o1)                     [123, 456, 789]
(%i2) is (x < y);
(%o2)                          true
(%i3) maybe (y > z);
(%o3)                          false
(%i4) if x >= z then 1 else 0;
(%o4)                            0
(%i5) block ([S], S : 0, for i:1 while i <= 100 do S : S + i, return (S));
(%o5)                          5050
```

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

```
(%o1)                     [123, 456, 789]
(%i2) [x < y, y <= z, z >= y, y > z];
(%o2)    [123 < 456, 456 <= 789, 789 >= 456, 456 > 789]
(%i3) map (is, %);
(%o3)               [true, true, true, false]
```

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 (including integer, rational, and real numbers) 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 floating point number 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)). If x is anything else, x! is not simplified.

The variables factlim, minfactorial, and factcomb control the simplification of expressions containing factorials.

The functions gamma, bffac, and cbffac are varieties of the gamma function. makegamma substitutes gamma for factorials and related functions.

See also binomial.

The factorial of an integer, half-integer, or floating point argument is simplified unless the operand is greater than factlim.

```
(%i1) factlim : 10;
(%o1)                          10
(%i2) [0!, (7/2)!, 4.77!, 8!, 20!];
              105 sqrt(%pi)
(%o2)   [1, -------------, 81.44668037931199, 40320, 20!]
                    16
```

The factorial of a complex number, known constant, or general expression is not simplified. Even so it may be possible simplify the factorial after evaluating the operand.

```
(%i1) [(%i + 1)!, %pi!, %e!, (cos(1) + sin(1))!];
(%o1)      [(%i + 1)!, %pi!, %e!, (sin(1) + cos(1))!]
(%i2) ev (%, numer, %enumer);
(%o2) [(%i + 1)!, 7.188082728976037, 4.260820476357,
```

```
                                                    1.227580202486819]
```

The factorial of an unbound symbol is not simplified.

```
(%i1) kill (foo);
(%o1)                          done
(%i2) foo!;
(%o2)                          foo!
```

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

```
(%i1) '([0!, (7/2)!, 4.77!, 8!, 20!]);
                105 sqrt(%pi)
(%o1) [1, -------------, 81.44668037931199, 40320,
                16
                                                  2432902008176640000]
```

!! Operator

The double factorial operator.

For an integer, float, or rational number `n`, `n!!` evaluates to the product `n (n-2) (n-4) (n-6) ... (n - 2 (k-1))` where `k` is equal to `entier (n/2)`, that is, the largest integer less than or equal to `n/2`. Note that this definition does not coincide with other published definitions for arguments which are not integers.

For an even (or odd) integer `n`, `n!!` evaluates to the product of all the consecutive even (or odd) integers from 2 (or 1) through `n` inclusive.

For an argument `n` which is not an integer, float, or rational, `n!!` yields a noun form `genfact (n, n/2, 2)`.

Operator

Represents the negation of syntactic equality `=`.

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

Examples:

```
(%i1) a = b;
(%o1)                          a = b
(%i2) is (a = b);
(%o2)                          false
(%i3) a # b;
(%o3)                          a # b
(%i4) not a = b;
(%o4)                          true
(%i5) is (a # b);
(%o5)                          true
(%i6) is (not a = b);
(%o6)                          true
```

. Operator

The dot operator, for matrix (non-commutative) multiplication. When "." is used in this way, spaces should be left on both sides of it, e.g. `A . B`. This distinguishes it plainly from a decimal point in a floating point number.

See also `dot`, `dot0nscsimp`, `dot0simp`, `dot1simp`, `dotassoc`, `dotconstrules`, `dotdistrib`, `dotexptsimp`, `dotident`, and `dotscrules`.

: 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;
(%o1)                           a
(%i2) a : 123;
(%o2)                          123
(%i3) a;
(%o3)                          123
```

Assignment to an element of a list.

```
(%i1) b : [1, 2, 3];
(%o1)                       [1, 2, 3]
(%i2) b[3] : 456;
(%o2)                          456
(%i3) b;
(%o3)                     [1, 2, 456]
```

Assignment creates an undeclared array.

```
(%i1) c[99] : 789;
(%o1)                          789
(%i2) c[99];
(%o2)                          789
(%i3) c;
(%o3)                           c
```

```
(%i4) arrayinfo (c);
(%o4)                    [hashed, 1, [99]]
(%i5) listarray (c);
(%o5)                           [789]
```

Multiple assignment.

```
(%i1) [a, b, c] : [45, 67, 89];
(%o1)                      [45, 67, 89]
(%i2) a;
(%o2)                           45
(%i3) b;
(%o3)                           67
(%i4) c;
(%o4)                           89
```

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

```
(%i1) [a, b] : [33, 55];
(%o1)                         [33, 55]
(%i2) [a, b] : [b, a];
(%o2)                         [55, 33]
(%i3) a;
(%o3)                            55
(%i4) b;
(%o4)                            33
```

:: Operator

Assignment operator.

`::` is the same as `:` (which see) except that `::` evaluates its left-hand side as well as its right-hand side.

Examples:

```
(%i1) x : 'foo;
(%o1)                           foo
(%i2) x :: 123;
(%o2)                           123
(%i3) foo;
(%o3)                           123
(%i4) x : '[a, b, c];
(%o4)                         [a, b, c]
(%i5) x :: [11, 22, 33];
(%o5)                      [11, 22, 33]
(%i6) a;
(%o6)                           11
(%i7) b;
(%o7)                           22
(%i8) c;
(%o8)                           33
```

::= Operator

Macro function definition operator. `::=` defines a function (called a "macro" for historical reasons) which quotes its arguments, and the expression which it returns (called the "macro expansion") is evaluated in the context from which the macro was called. A macro function is otherwise the same as an ordinary function.

`macroexpand` returns a macro expansion (without evaluating it). `macroexpand (foo (x))` followed by `''%` is equivalent to `foo (x)` when `foo` is a macro function.

`::=` puts the name of the new macro function onto the global list `macros`. `kill`, `remove`, and `remfunction` unbind macro function definitions and remove names from `macros`.

`fundef` or `dispfun` return a macro function definition or assign it to a label, respectively.

Macro functions commonly contain `buildq` and `splice` expressions to construct an expression, which is then evaluated.

Examples

A macro function quotes its arguments, so message (1) shows `y - z`, not the value of `y - z`. The macro expansion (the quoted expression `'(print ("(2) x is equal to", x))` is evaluated in the context from which the macro was called, printing message (2).

```
(%i1) x: %pi;
(%o1)                          %pi
(%i2) y: 1234;
(%o2)                         1234
(%i3) z: 1729 * w;
(%o3)                       1729 w
(%i4) printq1 (x) ::= block (print ("(1) x is equal to", x),
      '(print ("(2) x is equal to", x)));
(%o4) printq1(x) ::= block(print("(1) x is equal to", x),
                                '(print("(2) x is equal to", x)))
(%i5) printq1 (y - z);
(1) x is equal to y - z
(2) x is equal to %pi
(%o5)                          %pi
```

An ordinary function evaluates is arguments, so message (1) shows the value of `y - z`. The return value is not evaluated, so message (2) is not printed until the explicit evaluation `''%`.

```
(%i1) x: %pi;
(%o1)                          %pi
(%i2) y: 1234;
(%o2)                         1234
(%i3) z: 1729 * w;
(%o3)                       1729 w
(%i4) printe1 (x) := block (print ("(1) x is equal to", x),
      '(print ("(2) x is equal to", x)));
(%o4) printe1(x) := block(print("(1) x is equal to", x),
                                '(print("(2) x is equal to", x)))
```

```
(%i5) printe1 (y - z);
(1) x is equal to 1234 - 1729 w
(%o5)              print((2) x is equal to, x)
(%i6) ''%;
(2) x is equal to %pi
(%o6)                           %pi
```

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

```
(%i1) x: %pi;
(%o1)                           %pi
(%i2) y: 1234;
(%o2)                          1234
(%i3) z: 1729 * w;
(%o3)                        1729 w
(%i4) g (x) ::= buildq ([x], print ("x is equal to", x));
(%o4)    g(x) ::= buildq([x], print("x is equal to", x))
(%i5) macroexpand (g (y - z));
(%o5)              print(x is equal to, y - z)
(%i6) ''%;
x is equal to 1234 - 1729 w
(%o6)                     1234 - 1729 w
(%i7) g (y - z);
x is equal to 1234 - 1729 w
(%o7)                     1234 - 1729 w
```

`:=` Operator

The function definition operator. *f*`(`*x_1*`, ..., `*x_n*`) :=` *expr* defines a function named *f* with arguments *x_1*, ..., *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*, ..., *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 limit the scope of `f` to `g`.

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
                        x, y
```

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
```

=

Operator

The equation operator.

An expression *a* `=` *b*, by itself, represents an unevaluated equation, which might or might not hold. Unevaluated equations may appear as arguments to `solve` and `algsys` or some other functions.

The function `is` evaluates `=` to a Boolean value. `is(`*a* `=` *b*`)` evaluates *a* `=` *b* to `true` when *a* and *b* are identical. That is, *a* and *b* are atoms which are identical, or they are not atoms and their operators are identical and their arguments are identical. Otherwise, `is(`*a* `=` *b*`)` evaluates to `false`; it never evaluates to `unknown`. When `is(`*a* `=` *b*`)` is `true`, *a* and *b* are said to be syntactically equal, in contrast to equivalent expressions, for which `is(equal(`*a*`,` *b*`))` is `true`. Expressions can be equivalent and not syntactically equal.

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

In addition to `is`, some other operators evaluate `=` and `#` to `true` or `false`, namely `if`, `and`, `or`, and `not`.

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

`rhs` and `lhs` return the right-hand and left-hand sides, respectively, of an equation or inequation.

See also `equal` and `notequal`.

Examples:

An expression *a* `=` *b*, by itself, represents an unevaluated equation, which might or might not hold.

```
(%i1) eq_1 : a * x - 5 * y = 17;
(%o1)                      a x - 5 y = 17
(%i2) eq_2 : b * x + 3 * y = 29;
(%o2)                      3 y + b x = 29
(%i3) solve ([eq_1, eq_2], [x, y]);
                         196         29 a - 17 b
(%o3)           [[x = ---------, y = -----------]]
                       5 b + 3 a       5 b + 3 a
(%i4) subst (%, [eq_1, eq_2]);
          196 a     5 (29 a - 17 b)
(%o4) [--------- - --------------- = 17,
        5 b + 3 a      5 b + 3 a
                                    196 b     3 (29 a - 17 b)
                                  --------- + --------------- = 29]
                                  5 b + 3 a      5 b + 3 a
(%i5) ratsimp (%);
(%o5)                    [17 = 17, 29 = 29]
```

`is`(*a* = *b*) evaluates *a* = *b* to `true` when *a* and *b* are syntactically equal (that is, identical). Expressions can be equivalent and not syntactically equal.

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

Some operators evaluate = and # to `true` or `false`.

```
(%i1) if expand ((x + y)^2) = x^2 + 2 * x * y + y^2 then FOO else
      BAR;
(%o1)                             FOO
(%i2) eq_3 : 2 * x = 3 * x;
(%o2)                          2 x = 3 x
(%i3) eq_4 : exp (2) = %e^2;
                                 2     2
(%o3)                           %e  = %e
(%i4) [eq_3 and eq_4, eq_3 or eq_4, not eq_3];
(%o4)                     [false, true, true]
```

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 Operator

The logical conjunction operator. `and` is an n-ary infix operator; its operands are Boolean expressions, and its result is a Boolean value.

`and` forces evaluation (like `is`) of one or more operands, and may force evaluation of all operands.

Operands are evaluated in the order in which they appear. `and` evaluates only as many of its operands as necessary to determine the result. If any operand is `false`, the result is `false` and no further operands are evaluated.

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

`and` is not commutative: `a and b` might not be equal to `b and a` due to the treatment of indeterminate operands.

or Operator

The logical disjunction operator. `or` is an n-ary infix operator; its operands are Boolean expressions, and its result is a Boolean value.

`or` forces evaluation (like `is`) of one or more operands, and may force evaluation of all operands.

Operands are evaluated in the order in which they appear. `or` evaluates only as many of its operands as necessary to determine the result. If any operand is `true`, the result is `true` and no further operands are evaluated.

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

`or` is not commutative: `a or b` might not be equal to `b or a` due to the treatment of indeterminate operands.

not Operator

The logical negation operator. `not` is a prefix operator; its operand is a Boolean expression, and its result is a Boolean value.

`not` forces evaluation (like `is`) of its operand.

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

abs (*expr*) Function

Returns the absolute value *expr*. If *expr* is complex, returns the complex modulus of *expr*.

additive Keyword

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 Keyword

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

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

while

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

`allbut` is also recognized by `kill`.

```
(%i1) [aa : 11, bb : 22, cc : 33, dd : 44, ee : 55];
(%o1)                  [11, 22, 33, 44, 55]
(%i2) kill (allbut (cc, dd));
(%o0)                            done
(%i1) [aa, bb, cc, dd];
(%o1)                     [aa, bb, 33, 44]
```

`kill(allbut(`*a_1*`, `*a_2*`, ...))` has the effect of `kill(all)` except that it does not kill the symbols *a_1*, *a_2*, ...

antisymmetric Declaration

If `declare(h,antisymmetric)` is done, this tells the simplifier that `h` is antisymmetric. E.g. `h(x,z,y)` will simplify to `- h(x, y, z)`. That is, it will give (-1)^n times the result given by `symmetric` or `commutative`, where n is the number of interchanges of two arguments necessary to convert it to that form.

cabs (*expr*) Function

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

ceiling (*x*) Function

When *x* is a real number, return the least integer that is greater than or equal to *x*.

If *x* is a constant expression (`10 * %pi`, for example), `ceiling` evaluates *x* using big floating point numbers, and applies `ceiling` to the resulting big float. Because `ceiling` uses floating point evaluation, it's possible, although unlikely, that `ceiling` could return an erroneous value for constant inputs. To guard against errors, the floating point evaluation is done using three values for `fpprec`.

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

```
(%i1) ceiling (ceiling (x));
(%o1)                       ceiling(x)
(%i2) ceiling (floor (x));
(%o2)                        floor(x)
(%i3) declare (n, integer)$
(%i4) [ceiling (n), ceiling (abs (n)), ceiling (max (n, 6))];
(%o4)                 [n, abs(n), max(n, 6)]
(%i5) assume (x > 0, x < 1)$
(%i6) ceiling (x);
(%o6)                             1
(%i7) tex (ceiling (a));
$$\left \lceil a \right \rceil$$
(%o7)                            false
```

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

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

```
(%i1) declare (f, integervalued)$
(%i2) floor (f(x));
(%o2)                            f(x)
(%i3) ceiling (f(x) - 1);
(%o3)                         f(x) - 1
```

charfun (*p*) Function

Return 0 when the predicate *p* evaluates to `false`; return 1 when the predicate evaluates to `true`. When the predicate evaluates to something other than `true` or `false` (unknown), return a noun form.

Examples:

```
(%i1) charfun (x < 1);
(%o1)                       charfun(x < 1)
(%i2) subst (x = -1, %);
(%o2)                             1
(%i3) e : charfun ('"and" (-1 < x, x < 1))$
(%i4) [subst (x = -1, e), subst (x = 0, e), subst (x = 1, e)];
(%o4)                          [0, 1, 0]
```

commutative Declaration

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`.

compare (*x*, *y*) Function

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:

```
(%i1) compare (1, 2);
(%o1)                           <
(%i2) compare (1, x);
(%o2)                        unknown
(%i3) compare (%i, %i);
(%o3)                           =
(%i4) compare (%i, %i + 1);
(%o4)                     notcomparable
(%i5) compare (1/x, 0);
(%o5)                           #
(%i6) compare (x, abs(x));
(%o6)                          <=
```

The function `compare` doesn't try to determine whether the real domains of its arguments are nonempty; thus

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

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

entier (*x*) Function

Returns the largest integer less than or equal to *x* where *x* is numeric. `fix` (as in `fixnum`) is a synonym for this, so `fix(x)` is precisely the same.

equal (*a*, *b*) Function

Represents equivalence, that is, equal value.

By itself, `equal` does not evaluate or simplify. The function `is` attempts to evaluate `equal` to a Boolean value. `is(equal(a, b))` returns `true` (or `false`) if and only if *a* and *b* are equal (or not equal) for all possible values of their variables, as determined by evaluating `ratsimp(a - b)`; if `ratsimp` returns 0, the two expressions are considered equivalent. Two expressions may be equivalent even if they are not syntactically equal (i.e., identical).

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

In addition to `is`, some other operators evaluate `equal` and `notequal` to `true` or `false`, namely `if`, `and`, `or`, and `not`.

The negation of `equal` is `notequal`.

Examples:

By itself, `equal` does not evaluate or simplify.

```
(%i1) equal (x^2 - 1, (x + 1) * (x - 1));
                          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)
```

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

```
(%i1) ratsimp (x^2 - 1 - (x + 1) * (x - 1));
(%o1)                           0
(%i2) is (equal (x^2 - 1, (x + 1) * (x - 1)));
(%o2)                          true
(%i3) is (x^2 - 1 = (x + 1) * (x - 1));
(%o3)                          false
(%i4) ratsimp (x - (x + 1));
(%o4)                           - 1
(%i5) is (equal (x, x + 1));
(%o5)                          false
(%i6) is (x = x + 1);
(%o6)                          false
(%i7) ratsimp (x - y);
(%o7)                          x - y
(%i8) is (equal (x, y));
(%o8)                         unknown
(%i9) is (x = y);
(%o9)                          false
```

When `is` fails to reduce `equal` to `true` or `false`, the result is governed by the global flag `prederror`.

```
(%i1) [aa : x^2 + 2*x + 1, bb : x^2 - 2*x - 1];
                     2             2
(%o1)              [x  + 2 x + 1, x  - 2 x - 1]
(%i2) ratsimp (aa - bb);
(%o2)                         4 x + 2
(%i3) prederror : true;
(%o3)                          true
(%i4) is (equal (aa, bb));
Maxima was unable to evaluate the predicate:
       2             2
equal(x  + 2 x + 1, x  - 2 x - 1)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i5) prederror : false;
(%o5)                          false
(%i6) is (equal (aa, bb));
(%o6)                         unknown
```

Some operators evaluate `equal` and `notequal` to `true` or `false`.

```
(%i1) if equal (y, y - 1) then FOO else BAR;
(%o1)                           BAR
(%i2) eq_1 : equal (x, x + 1);
(%o2)                      equal(x, x + 1)
(%i3) eq_2 : equal (y^2 + 2*y + 1, (y + 1)^2);
                          2                   2
(%o3)              equal(y  + 2 y + 1, (y + 1) )
(%i4) [eq_1 and eq_2, eq_1 or eq_2, not eq_1];
```

```
(%o4)                    [false, true, true]
```

Because `not` *expr* causes evaluation of *expr*, `not equal(`*a*`,` *b*`)` is equivalent to `is(notequal(`*a*`,` *b*`))`.

```
(%i1) [notequal (2*z, 2*z - 1), not equal (2*z, 2*z - 1)];
(%o1)            [notequal(2 z, 2 z - 1), true]
(%i2) is (notequal (2*z, 2*z - 1));
(%o2)                          true
```

floor (*x*) Function

When *x* is a real number, return the largest integer that is less than or equal to *x*.

If *x* is a constant expression (`10 * %pi`, for example), `floor` evaluates *x* using big floating point numbers, and applies `floor` to the resulting big float. Because `floor` uses floating point evaluation, it's possible, although unlikely, that `floor` could return an erroneous value for constant inputs. To guard against errors, the floating point evaluation is done using three values for `fpprec`.

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

```
(%i1) floor (ceiling (x));
(%o1)                       ceiling(x)
(%i2) floor (floor (x));
(%o2)                        floor(x)
(%i3) declare (n, integer)$
(%i4) [floor (n), floor (abs (n)), floor (min (n, 6))];
(%o4)                 [n, abs(n), min(n, 6)]
(%i5) assume (x > 0, x < 1)$
(%i6) floor (x);
(%o6)                            0
(%i7) tex (floor (a));
$$\left \lfloor a \right \rfloor$$
(%o7)                          false
```

The function `floor` does not automatically map over lists or matrices. Finally, for all inputs that are manifestly complex, `floor` returns a noun form.

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

```
(%i1) declare (f, integervalued)$
(%i2) floor (f(x));
(%o2)                          f(x)
(%i3) ceiling (f(x) - 1);
(%o3)                       f(x) - 1
```

notequal (*a*, *b*) Function

Represents the negation of `equal(`*a*`,` *b*`)`.

Examples:

```
(%i1) equal (a, b);
(%o1)                       equal(a, b)
```

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

eval Operator

As an argument in a call to `ev (expr)`, `eval` causes an extra evaluation of *expr*. See `ev`.

evenp (*expr*) Function

Returns `true` if *expr* is an even integer. `false` is returned in all other cases.

fix (*x*) Function

A synonym for `entier (x)`.

fullmap (*f, expr_1, ...*) Function

Similar to `map`, but `fullmap` keeps mapping down all subexpressions until the main operators are no longer the same.

`fullmap` is used by the Maxima simplifier for certain matrix manipulations; thus, Maxima sometimes generates an error message concerning `fullmap` even though `fullmap` was not explicitly called by the user.

Examples:

```
(%i1) a + b * c;
(%o1)                           b c + a
(%i2) fullmap (g, %);
(%o2)                     g(b) g(c) + g(a)
(%i3) map (g, %th(2));
(%o3)                       g(b c) + g(a)
```

fullmapl (*f, list_1, ...*) Function

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*) Function

Attempts to determine whether the predicate *expr* is provable from the facts in the `assume` database.

If the predicate is provably `true` or `false`, `is` returns `true` or `false`, respectively. Otherwise, the return value is governed by the global flag `prederror`. When `prederror` is `true`, `is` complains with an error message. Otherwise, `is` returns `unknown`.

`ev(`*expr*`, pred)` (which can be written *expr*`, pred` at the interactive prompt) is equivalent to `is(`*expr*`)`.

See also `assume`, `facts`, and `maybe`.

Examples:

`is` causes evaluation of predicates.

```
(%i1) %pi > %e;
(%o1)                         %pi > %e
(%i2) is (%pi > %e);
(%o2)                           true
```

`is` attempts to derive predicates from the `assume` database.

```
(%i1) assume (a > b);
(%o1)                         [a > b]
(%i2) assume (b > c);
(%o2)                         [b > c]
(%i3) is (a < b);
(%o3)                          false
(%i4) is (a > c);
(%o4)                          true
(%i5) is (equal (a, c));
(%o5)                          false
```

If `is` can neither prove nor disprove a predicate from the `assume` database, the global flag `prederror` governs the behavior of `is`.

```
(%i1) assume (a > b);
(%o1)                         [a > b]
(%i2) prederror: true$
(%i3) is (a > 0);
Maxima was unable to evaluate the predicate:
a > 0
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i4) prederror: false$
(%i5) is (a > 0);
(%o5)                        unknown
```

maybe (*expr*) Function

Attempts to determine whether the predicate *expr* is provable from the facts in the `assume` database.

If the predicate is provably `true` or `false`, `maybe` returns `true` or `false`, respectively. Otherwise, `maybe` returns `unknown`.

`maybe` is functionally equivalent to `is` with `prederror: false`, but the result is computed without actually assigning a value to `prederror`.

See also `assume`, `facts`, and `is`.

Examples:

```
(%i1) maybe (x > 0);
(%o1)                         unknown
(%i2) assume (x > 1);
(%o2)                         [x > 1]
(%i3) maybe (x > 0);
(%o3)                          true
```

isqrt (*x*) Function

Returns the "integer square root" of the absolute value of *x*, which is an integer.

lmax (*L*) Function

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*) Function

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*) Function

Return a simplified value for the maximum of the expressions *x_1* through *x_n*. When `get (trylevel, maxmin)`, is 2 or greater, `max` uses the simplification `max (e, -e) --> |e|`. When `get (trylevel, maxmin)` is 3 or greater, *max* tries to eliminate expressions that are between two other arguments; for example, `max (x, 2*x, 3*x) -> max (x, 3*x)`. To set the value of `trylevel` to 2, use `put (trylevel, 2, maxmin)`.

min (*x_1*, ..., *x_n*) Function

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

polymod (*p*) Function
polymod (*p*, *m*) Function

Converts the polynomial *p* to a modular representation with respect to the current modulus which is the value of the variable `modulus`.

`polymod (`*p*`,` *m*`)` specifies a modulus *m* to be used instead of the current value of `modulus`.

See `modulus`.

mod (*x*, *y*) Function

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

To find the principal argument (a number in the interval `(-%pi, %pi]`) of a complex number, use the function *x* `|->` `%pi - mod (%pi -` *x*`, 2*%pi)`, where *x* is an argument.

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

For nonnumerical arguments *x* or *y*, `mod` knows several simplification rules:

```
(%i1) mod (x, 0);
(%o1)                          x
(%i2) mod (a*x, a*y);
(%o2)                      a mod(x, y)
(%i3) mod (0, x);
(%o3)                          0
```

oddp (*expr*) Function

is `true` if *expr* is an odd integer. `false` is returned in all other cases.

pred Operator

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

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

Copies *s* to the random number generator state.

`set_random_state` always returns `done`.

random (*x*) Function

Returns a pseudorandom number. If *x* is an integer, `random (x)` returns an integer from 0 through *x* `-` `1` inclusive. If *x* is a floating point number, `random (x)` returns a nonnegative floating point number less than *x*. `random` complains with an error if *x* is neither an integer nor a float, or if *x* is not positive.

The functions `make_random_state` and `set_random_state` maintain the state of the random number generator.

The Maxima random number generator is an implementation of the Mersenne twister MT 19937.

Examples:

```
(%i1) s1: make_random_state (654321)$
(%i2) set_random_state (s1);
(%o2)                          done
(%i3) random (1000);
(%o3)                           768
(%i4) random (9573684);
(%o4)                         7657880
(%i5) random (2^75);
(%o5)                 11804491615036831636390
(%i6) s2: make_random_state (false)$
(%i7) random (1.0);
(%o7)                    .2310127244107132
(%i8) random (10.0);
(%o8)                    4.394553645870825
(%i9) random (100.0);
(%o9)                    32.28666704056853
(%i10) set_random_state (s2);
(%o10)                         done
(%i11) random (1.0);
(%o11)                   .2310127244107132
(%i12) random (10.0);
(%o12)                   4.394553645870825
(%i13) random (100.0);
(%o13)                   32.28666704056853
```

rationalize (*expr*) Function

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

```
(%o2)                           --
                                10
(%i3) fpprec : 5$
(%i4) rationalize (0.1b0);
                              209715
(%o4)                         -------
                              2097152
(%i5) fpprec : 20$
(%i6) rationalize (0.1b0);
                     236118324143482260685
(%o6)                ----------------------
                     236118324143482606848
(%i7) rationalize (sin (0.1*x + 5.6));
                                x    28
(%o7)                      sin(-- + --)
                                10   5
```

Example use:

```
(%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));
(%o1) unitfrac(r) := block([uf : [], q],
if not ratnump(r) then
   error("The input to 'unitfrac' must be a rational number"),
                                    1
while r # 0 do (uf : cons(q : ----------, uf), r : r - q),
                                       1
                                  ceiling(-)
                                       r
reverse(uf))
(%i2) unitfrac (9/10);
                             1  1  1
(%o2)                       [-, -, --]
                             2  3  15
(%i3) apply ("+", %);
                                9
(%o3)                           --
                                10
(%i4) unitfrac (-9/10);
                                   1
(%o4)                       [- 1, --]
                                   10
(%i5) apply ("+", %);
                                 9
(%o5)                          - --
                                 10
```

```
(%i6) unitfrac (36/37);
                          1  1  1  1    1
(%o6)                    [-, -, -, --, ----]
                          2  3  8  69  6808
(%i7) apply ("+", %);
                                36
(%o7)                           --
                                37
```

round (*x*) Function

When *x* is a real number, returns the closest integer to *x*. Multiples of 1/2 are rounded to the nearest even integer. Evaluation of *x* is similar to `floor` and `ceiling`.

sign (*expr*) Function

Attempts to determine the sign of *expr* on the basis of the facts in the current data base. It returns one of the following answers: `pos` (positive), `neg` (negative), `zero`, `pz` (positive or zero), `nz` (negative or zero), `pn` (positive or negative), or `pnz` (positive, negative, or zero, i.e. nothing known).

signum (*x*) Function

For numeric *x*, returns 0 if *x* is 0, otherwise returns -1 or +1 as *x* is less than or greater than 0, respectively.

If *x* is not numeric then a simplified but equivalent form is returned. For example, `signum(-x)` gives `-signum(x)`.

sort (*L*, *P*) Function

sort (*L*) Function

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

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

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

`sort (`*L*`)` is equivalent to `sort (`*L*`, orderlessp)`. That is, the default sorting order is ascending, as determined by `orderlessp`. All Maxima atoms and expressions are comparable under `orderlessp`, although there are isolated examples of expressions for which `orderlessp` is not transitive; this is a bug.

Examples:

```
(%i1) sort ([11, -17, 29b0, 7.55, 3, -5/2, b + a, 9 * c,
      19 - 3 * x]);
                5
(%o1) [- 17, - -, 3, 7.55, 11, 2.9b1, b + a, 9 c, 19 - 3 x]
                2
```

```
(%i2) sort ([11, -17, 29b0, 7.55, 3, -5/2, b + a, 9*c, 19 - 3*x],
      ordergreatp);
                                                          5
(%o2) [19 - 3 x, 9 c, b + a, 2.9b1, 11, 7.55, 3, - -, - 17]
                                                          2
(%i3) sort ([%pi, 3, 4, %e, %gamma]);
(%o3)                 [3, 4, %e, %gamma, %pi]
(%i4) sort ([%pi, 3, 4, %e, %gamma], "<");
(%o4)                 [%gamma, %e, 3, %pi, 4]
(%i5) my_list: [[aa,hh,uu], [ee,cc], [zz,xx,mm,cc], [%pi,%e]];
(%o5) [[aa, hh, uu], [ee, cc], [zz, xx, mm, cc], [%pi, %e]]
(%i6) sort (my_list);
(%o6) [[%pi, %e], [aa, hh, uu], [ee, cc], [zz, xx, mm, cc]]
(%i7) sort (my_list, lambda ([a, b], orderlessp (reverse (a),
      reverse (b))));
(%o7) [[%pi, %e], [ee, cc], [zz, xx, mm, cc], [aa, hh, uu]]
```

sqrt (*x*) Function

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

`radexpand` if `true` will cause nth 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 `radexpand` 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.

The variable `sublis_apply_lambda` controls simplification after `sublis`.

Example:

```
(%i1) sublis ([a=b, b=a], sin(a) + cos(b));
(%o1)                    sin(b) + cos(a)
```

sublist (*list*, *p*) Function

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

Example:

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

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.

subst (*a*, *b*, *c*) Function

Substitutes *a* for *b* in *c*. *b* must be an atom or a complete subexpression of *c*. For example, `x+y+z` is a complete subexpression of `2*(x+y+z)/w` while `x+y` is not. When *b* does not have these characteristics, one may sometimes use `substpart` or `ratsubst` (see below). Alternatively, if *b* is of the form `e/f` then one could use `subst (a*f, e, c)` while if *b* is of the form `e^(1/f)` then one could use `subst (a^f, e, c)`. The `subst` command also discerns the `x^y` in `x^-y` so that `subst (a, sqrt(x), 1/sqrt(x))` yields `1/a`. *a* and *b* may also be operators of an expression enclosed in double-quotes `"` or they may be function names. If one wishes to substitute for the independent variable in derivative forms then the `at` function (see below) should be used.

`subst` is an alias for `substitute`.

`subst (`*eq_1*`,` *expr*`)` or `subst ([`*eq_1*`, ..., `*eq_k*`],` *expr*`)` are other permissible forms. The *eq_i* are equations indicating substitutions to be made. For each equation, the right side will be substituted for the left in the expression *expr*.

`exptsubst` if `true` permits substitutions like `y` for `%e^x` in `%e^(a*x)` to take place.

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

Examples:

```
(%i1) subst (a, x+y, x + (x+y)^2 + y);
                                     2
(%o1)                         y + x + a
(%i2) subst (-%i, %i, a + b*%i);
(%o2)                          a - %i b
```

For further examples, do `example (subst)`.

substinpart (*x*, *expr*, *n_1*, ..., *n_k*) Function

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

Examples:

```
(%i1) x . 'diff (f(x), x, 2);
                                 2
                                d
(%o1)                    x . (--- (f(x)))
                                  2
                                 dx
(%i2) substinpart (d^2, %, 2);
                                    2
(%o2)                            x . d
(%i3) substinpart (f1, f[1](x + 1), 0);
(%o3)                          f1(x + 1)
```

If the last argument to a `part` function is a list of indices then several subexpressions are picked out, each one corresponding to an index of the list. Thus

```
(%i1) part (x + y + z, [1, 3]);
(%o1)                            z + x
```

`piece` holds the value of the last expression selected when using the `part` functions. It is set during the execution of the function and thus may be referred to in the function itself as shown below. If `partswitch` is set to `true` then `end` is returned when a selected part of an expression doesn't exist, otherwise an error message is given.

```
(%i1) expr: 27*y^3 + 54*x*y^2 + 36*x^2*y + y + 8*x^3 + x + 1;
                3         2       2           3
(%o1)     27 y  + 54 x y  + 36 x  y + y + 8 x  + x + 1
(%i2) part (expr, 2, [1, 3]);
                                 2
(%o2)                          54 y
(%i3) sqrt (piece/54);
                              abs(y)
(%i4) substpart (factor (piece), expr, [1, 2, 3, 5]);
                                3
(%o4)               (3 y + 2 x)  + y + x + 1
(%i5) expr: 1/x + y/x - 1/z;
                              1   y   1
(%o5)                       - - + - + -
                              z   x   x
(%i6) substpart (xthru (piece), expr, [2, 3]);
                             y + 1   1
(%o6)                        ----- - -
                               x     z
```

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

substpart (*x, expr, n_1, ..., n_k*) Function

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

```
(%i1) 1/(x^2 + 2);
                                1
(%o1)                         ------
                               2
                              x  + 2
(%i2) substpart (3/2, %, 2, 1, 2);
                                1
(%o2)                         --------
                               3/2
                              x    + 2
(%i3) a*x + f(b, y);
                          a x + f(b, y)
(%i4) substpart ("+", %, 1, 0);
(%o4)                     x + f(b, y) + a
```

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

subvarp (*expr*) Function

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

symbolp (*expr*) Function

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

See also Section 6.4 [Identifiers], page 61.

unorder () Function

Disables the aliasing created by the last use of the ordering commands `ordergreat` and `orderless`. `ordergreat` and `orderless` may not be used more than one time each without calling `unorder`. See also `ordergreat` and `orderless`.

Examples:

```
(%i1) unorder();
(%o1)                           []
(%i2) b*x + a^2;
                                     2
(%o2)                          b x + a
(%i3) ordergreat (a);
(%o3)                           done
(%i4) b*x + a^2;
 %th(1) - %th(3);
                                2
(%o4)                          a  + b x
(%i5) unorder();
                                 2    2
(%o5)                           a  - a
```

vectorpotential (*givencurl*) Function

Returns the vector potential of a given curl vector, in the current coordinate system. `potentialzeroloc` has a similar role as for `potential`, but the order of the left-hand sides of the equations must be a cyclic permutation of the coordinate variables.

xthru (*expr*) Function

Combines all terms of *expr* (which should be a sum) over a common denominator without expanding products and exponentiated sums as `ratsimp` does. `xthru` cancels common factors in the numerator and denominator of rational expressions but only if the factors are explicit.

Sometimes it is better to use `xthru` before `ratsimp`ing 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 `ratsimp`ed.

```
(%i1) ((x+2)^20 - 2*y)/(x+y)^20 + (x+y)^(-19) - x/(x+y)^20;
                                    20
                  1        (x + 2)   - 2 y        x
(%o1)         --------- + --------------- - ---------
                    19              20              20
```

```
                (y + x)          (y + x)          (y + x)
(%i2) xthru (%);
                                      20
                                (x + 2)   - y
(%o2)                           -------------
                                         20
                                  (y + x)
```

zeroequiv (*expr*, *v*) Function

Tests whether the expression *expr* in the variable *v* is equivalent to zero, returning `true`, `false`, or `dontknow`.

`zeroequiv` has these restrictions:

1. Do not use functions that Maxima does not know how to differentiate and evaluate.
2. If the expression has poles on the real line, there may be errors in the result (but this is unlikely to occur).
3. If the expression contains functions which are not solutions to first order differential equations (e.g. Bessel functions) there may be incorrect results.
4. The algorithm uses evaluation at randomly chosen points for carefully selected subexpressions. This is always a somewhat hazardous business, although the algorithm tries to minimize the potential for error.

For example `zeroequiv (sin(2*x) - 2*sin(x)*cos(x), x)` returns `true` and `zeroequiv (%e^x + x, x)` returns `false`. On the other hand `zeroequiv (log(a*b) - log(a) - log(b), a)` returns `dontknow` because of the presence of an extra parameter `b`.

6 Expressions

6.1 Introduction to Expressions

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

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

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

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

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

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

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

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

6.2 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 - 4*x + 13 = 0` are `2 + 3*%i` and `2 - 3*%i`. Note that simplification of products of complex expressions can be effected by expanding the product. Simplification of quotients, roots, and other functions of complex expressions can usually be accomplished by using the `realpart`, `imagpart`, `rectform`, `polarform`, `abs`, `carg` functions.

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:

```
(%i1) foo (x) := x^2;
                                      2
(%o1)                      foo(x) := x
(%i2) foo (42);
(%o2)                           1764
(%i3) 'foo (42);
(%o3)                          foo(42)
(%i4) 'foo (42), nouns;
(%o4)                           1764
(%i5) declare (bar, noun);
(%o5)                           done
(%i6) bar (x) := x/17;
                                      x
(%o6)                     ''bar(x) := --
                                      17
(%i7) bar (52);
(%o7)                          bar(52)
(%i8) bar (52), nouns;
                                 52
(%o8)                            --
                                 17
(%i9) integrate (1/x, x, 1, 42);
(%o9)                           log(42)
(%i10) 'integrate (1/x, x, 1, 42);
                                 42
                                /
                                [   1
(%o10)                          I   - dx
                                ]   x
                                /
                                 1
(%i11) ev (%, nouns);
```

```
(%o11)                          log(42)
```

6.4 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 7 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 7 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.";
(%o1)                This is a string.
(%i2) s_2 : "Embedded \"double quotes\" and backslash \\ characters.";
(%o2) Embedded "double quotes" and backslash \ characters.
(%i3) s_3 : "Embedded line termination
in this string.";
(%o3) Embedded line termination
in this string.
(%i4) s_4 : "Ignore the \
line termination \
characters in \
this string.";
(%o4) Ignore the line termination characters in this string.
(%i5) stringdisp : false;
(%o5)                           false
(%i6) s_1;
(%o6)                    This is a string.
(%i7) stringdisp : true;
(%o7)                           true
(%i8) s_1;
(%o8)                   "This is a string."
```

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 undefine existing operators, or to redefine the precedence of existing operators. An operator may be unary

prefix or unary postfix, binary infix, n-ary infix, matchfix, or nofix. "Matchfix" means a pair of symbols which enclose their argument or arguments, and "nofix" means an operator which takes no arguments. As examples of the different types of operators, there are the following.

unary prefix
: negation `- a`

unary postfix
: factorial `a!`

binary infix
: exponentiation `a^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");
(%o1)                           dd
(%i2) dd a;
(%o2)                          dd a
(%i3) "dd" (a);
(%o3)                          dd a
(%i4) infix ("<-");
(%o4)                           <-
(%i5) a <- dd b;
(%o5)                      a <- dd b
(%i6) "<-" (a, "dd" (b));
(%o6)                      a <- dd b
```

The Maxima functions which define new operators are summarized in this table, stating the default left and right binding powers (lbp and rbp, respectively). (Binding power determines operator precedence. However, since left and right binding powers can differ, binding power is somewhat more complicated than precedence.) Some of the operation definition functions take additional arguments; see the function descriptions for details.

`prefix`
: rbp=180

`postfix`
: lbp=180

`infix`
: lbp=180, rbp=180

`nary`
: lbp=180, rbp=180

`matchfix`
: (binding power not applicable)

`nofix`
: (binding power not applicable)

For comparison, here are some built-in operators and their left and right binding powers.

```
Operator   lbp     rbp

  :        180     20
  ::       180     20
  :=       180     20
  ::=      180     20
  !        160
  !!       160
  ^        140     139
  .        130     129
  *        120
  /        120     120
  +        100     100
  -        100     134
  =        80      80
  #        80      80
  >        80      80
  >=       80      80
  <        80      80
  <=       80      80
  not              70
  and      65
  or       60
  ,        10
  $        -1
  ;        -1
```

`remove` and `kill` remove operator properties from an atom. `remove ("a", op)` removes only the operator properties of *a*. `kill ("a")` removes all properties of *a*, including the operator properties. Note that the name of the operator must be enclosed in quotation marks.

```
(%i1) infix ("##");
(%o1)                           ##
(%i2) "##" (a, b) := a^b;
                                         b
(%o2)                      a ## b := a
(%i3) 5 ## 3;
(%o3)                           125
(%i4) remove ("##", op);
(%o4)                          done
(%i5) 5 ## 3;
Incorrect syntax: # is not a prefix operator
5 ##
  ^
(%i5) "##" (5, 3);
(%o5)                           125
(%i6) infix ("##");
(%o6)                           ##
```

```
(%i7) 5 ## 3;
(%o7)                          125
(%i8) kill ("##");
(%o8)                         done
(%i9) 5 ## 3;
Incorrect syntax: # is not a prefix operator
5 ##
  ^
(%i9) "##" (5, 3);
(%o9)                       ##(5, 3)
```

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 series, not parallel.

See also `atvalue`. For other functions which carry out substitutions, see also `subst` and `ev`.

Examples:

```
(%i1) atvalue (f(x,y), [x = 0, y = 1], a^2);
                                 2
(%o1)                           a
(%i2) atvalue ('diff (f(x,y), x), x = 0, 1 + y);
(%o2)                        @2 + 1
(%i3) printprops (all, atvalue);
                                 !
                  d              !
                 --- (f(@1, @2))!       = @2 + 1
                 d@1             !
                                 !@1 = 0

                                      2
                           f(0, 1) = a

(%o3)                         done
(%i4) diff (4*f(x, y)^2 - u(x, y)^2, x);
                   d                              d
(%o4)  8 f(x, y) (-- (f(x, y))) - 2 u(x, y) (-- (u(x, y)))
                   dx                             dx
(%i5) at (%, [x = 0, y = 1]);
                                          !
```

```
                2              d           !
(%o5)     16 a  - 2 u(0, 1) (-- (u(x, y))!            )
                             dx          !
                                         !x = 0, y = 1
```

box (*expr*) Function
box (*expr, a*) Function

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

`box` (*expr*, *a*) encloses *expr* in a box labelled by the symbol *a*. The label is truncated if it is longer than the width of the box.

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

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

Examples:

```
(%i1) box (a^2 + b^2);
                              """""""""
                              " 2    2"
(%o1)                         "b  + a "
                              """""""""
(%i2) a : 1234;
(%o2)                           1234
(%i3) b : c - d;
(%o3)                           c - d
(%i4) box (a^2 + b^2);
                         """""""""""""""""""""
                         "       2           "
(%o4)                    "(c - d)  + 1522756"
                         """""""""""""""""""""
(%i5) box (a^2 + b^2, term_1);
                         term_1"""""""""""""""
                         "       2           "
(%o5)                    "(c - d)  + 1522756"
                         """""""""""""""""""""
(%i6) 1729 - box (1729);
                                      """""""
(%o6)                        1729 - "1729"
                                      """""""
(%i7) boxchar: "-";
(%o7)                                 -
(%i8) box (sin(x) + cos(y));
                         -----------------
(%o8)                    -cos(y) + sin(x)-
                         -----------------
```

boxchar Option variable

Default value: "

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

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

carg (*z*) Function

Returns the complex argument of *z*. The complex argument is an angle `theta` in `(-%pi, %pi]` such that `r exp (theta %i) =` *z* where `r` is the magnitude of *z*.

`carg` is a computational function, not a simplifying function.

`carg` ignores the declaration `declare (`*x*`, complex)`, and treats *x* as a real variable. This is a bug.

See also `abs` (complex magnitude), `polarform`, `rectform`, `realpart`, and `imagpart`.

Examples:

```
(%i1) carg (1);
(%o1)                           0
(%i2) carg (1 + %i);
                               %pi
(%o2)                          ---
                                4
(%i3) carg (exp (%i));
(%o3)                           1
(%i4) carg (exp (%pi * %i));
(%o4)                          %pi
(%i5) carg (exp (3/2 * %pi * %i));
                                %pi
(%o5)                         - ---
                                 2
(%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:

```
(%i1) constantp (7 * sin(2));
(%o1)                                true
(%i2) constantp (rat (17/29));
(%o2)                                true
(%i3) constantp (%pi * sin(%e));
(%o3)                                true
(%i4) constantp (exp (x));
(%o4)                                false
(%i5) declare (x, constant);
(%o5)                                done
(%i6) constantp (exp (x));
(%o6)                                true
(%i7) constantp (foo (x) + bar (%e) + baz (2));
(%o7)                                false
(%i8)
```

declare (*a_1, p_1, a_2, p_2, ...*) Function

Assigns the atom or list of atoms *a_i* the property or list of properties *p_i*. When *a_i* and/or *p_i* 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_i* known to `ev` so that the function named by *a_i* is applied when *a_i* appears as a flag argument of `ev`. See `evfun`.

`evflag` Makes *a_i* known to the `ev` function so that *a_i* is bound to `true` during the execution of `ev` when *a_i* appears as a flag argument of `ev`. See `evflag`.

`bindtest` Tells Maxima to trigger an error when *a_i* is evaluated unbound.

`noun` Tells Maxima to parse *a_i* as a noun. The effect of this is to replace instances of *a_i* with `'`*a_i* or `nounify(`*a_i*`)`, depending on the context.

`constant` Tells Maxima to consider *a_i* a symbolic constant.

`scalar` Tells Maxima to consider *a_i* a scalar variable.

`nonscalar` Tells Maxima to consider *a_i* a nonscalar variable. The usual application is to declare a variable as a symbolic vector or matrix.

`mainvar` Tells Maxima to consider *a_i* a "main variable". A main variable succeeds all other constants and variables in the canonical ordering of Maxima expressions, as determined by `ordergreatp`.

`alphabetic` Tells Maxima to recognize all characters in *a_i* (which must be a string) as alphabetic characters.

`feature` Tells Maxima to recognize *a_i* as the name of a feature. Other atoms may then be declared to have the *a_i* property.

`rassociative`, `lassociative`
Tells Maxima to recognize *a_i* as a right-associative or left-associative function.

`nary` Tells Maxima to recognize *a_i* as an n-ary function.

The `nary` declaration is not the same as calling the `nary` function. The sole effect of `declare(foo, nary)` is to instruct the Maxima simplifier to flatten nested expressions, for example, to simplify `foo(x, foo(y, z))` to `foo(x, y, z)`.

`symmetric`, `antisymmetric`, `commutative`
Tells Maxima to recognize *a_i* as a symmetric or antisymmetric function. `commutative` is the same as `symmetric`.

`oddfun`, `evenfun`
Tells Maxima to recognize *a_i* as an odd or even function.

`outative` Tells Maxima to simplify *a_i* expressions by pulling constant factors out of the first argument.

When *a_i* has one argument, a factor is considered constant if it is a literal or declared constant.

When *a_i* has two or more arguments, a factor is considered constant if the second argument is a symbol and the factor is free of the second argument.

`multiplicative`
Tells Maxima to simplify *a_i* expressions by the substitution *a_i*`(x * y * z * ...) --> a_i(x) * a_i(y) * a_i(z) * ...`. The substitution is carried out on the first argument only.

`additive` Tells Maxima to simplify *a_i* expressions by the substitution *a_i*`(x + y + z + ...) --> a_i(x) + a_i(y) + a_i(z) + ...`. The substitution is carried out on the first argument only.

`linear` Equivalent to declaring *a_i* both `outative` and `additive`.

`integer`, `noninteger`
Tells Maxima to recognize *a_i* as an integer or noninteger variable.

`even`, `odd` Tells Maxima to recognize *a_i* as an even or odd integer variable.

`rational`, `irrational`
Tells Maxima to recognize *a_i* as a rational or irrational real variable.

`real`, `imaginary`, `complex`
Tells Maxima to recognize *a_i* as a real, pure imaginary, or complex variable.

`increasing`, `decreasing`
Tells Maxima to recognize *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;
                                     3
(%o2)                         (b + a)
(%i3) (a + b)^3, expand;
                       3        2      2      3
(%o3)                 b  + 3 a b  + 3 a  b + a
(%i4) declare (demoivre, evflag);
(%o4)                          done
(%i5) exp (a + b*%i);
                                %i b + a
(%o5)                          %e
(%i6) exp (a + b*%i), demoivre;
                         a
(%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);
                                2
(%o1)                      2 3  47 14593
(%i2) declare (factor, noun);
(%o2)                           done
(%i3) factor (12345678);
(%o3)                    factor(12345678)
(%i4) ''%, nouns;
                                2
(%o4)                      2 3  47 14593
```

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

`alphabetic` declaration.

```
(%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) e (- u) - e (u);
(%o4)                      e(- u) - e(u)
(%i5) declare (e, evenfun);
(%o5)                           done
(%i6) e (- u) - e (u);
(%o6)                             0
```

`outative` declaration.

```
(%i1) F1 (100 * x);
(%o1)                         F1(100 x)
(%i2) declare (F1, outative);
(%o2)                           done
(%i3) F1 (100 * x);
(%o3)                         100 F1(x)
(%i4) declare (zz, constant);
(%o4)                           done
(%i5) F1 (zz * y);
(%o5)                         zz F1(y)
```

`multiplicative` declaration.

```
(%i1) F2 (a * b * c);
(%o1)                         F2(a b c)
(%i2) declare (F2, multiplicative);
(%o2)                           done
(%i3) F2 (a * b * c);
(%o3)                    F2(a) F2(b) F2(c)
```

`additive` declaration.

```
(%i1) F3 (a + b + c);
(%o1)                       F3(c + b + a)
(%i2) declare (F3, additive);
(%o2)                           done
(%i3) F3 (a + b + c);
(%o3)                  F3(c) + F3(b) + F3(a)
```

`linear` declaration.

```
(%i1) 'sum (F(k) + G(k), k, 1, inf);
                         inf
                         ====
                         \
(%o1)                     >    (G(k) + F(k))
                         /
                         ====
                         k = 1
(%i2) declare (nounify (sum), linear);
(%o2)                           done
(%i3) 'sum (F(k) + G(k), k, 1, inf);
                       inf          inf
                       ====         ====
                       \            \
(%o3)                   >    G(k) +  >    F(k)
```

```
/            /
====         ====
k = 1        k = 1
```

disolate (*expr, x_1, ..., x_n*) Function

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

dispform (*expr*) Function

Returns the external representation of *expr* with respect to its main operator. This should be useful in conjunction with `part` which also deals with the external representation. Suppose *expr* is -A . Then the internal representation of *expr* is "*"(-1,A), while the external representation is "-"(A). `dispform (expr, all)` converts the entire expression (not just the top-level) to external format. For example, if `expr: sin (sqrt (x))`, then `freeof (sqrt, expr)` and `freeof (sqrt, dispform (expr))` give `true`, while `freeof (sqrt, dispform (expr, all))` gives `false`.

distrib (*expr*) Function

Distributes sums over products. It differs from `expand` in that it works at only the top level of an expression, i.e., it doesn't recurse and it is faster than `expand`. It differs from `multthru` in that it expands all sums at that level.

Examples:

```
(%i1) distrib ((a+b) * (c+d));
(%o1)                 b d + a d + b c + a c
(%i2) multthru ((a+b) * (c+d));
(%o2)                 (b + a) d + (b + a) c
(%i3) distrib (1/((a+b) * (c+d)));
                                1
(%o3)                   ---------------
                        (b + a) (d + c)
(%i4) expand (1/((a+b) * (c+d)), 1, 0);
                                1
(%o4)                   ---------------------
                        b d + a d + b c + a c
```

dpart (*expr, n_1, ..., n_k*) Function

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

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

```
"z"
"""
```

exp (*x*) Function

Represents the exponential function. Instances of `exp (x)` in input are simplified to `%e^x`; `exp` does not appear in simplified expressions.

`demoivre` if `true` causes `%e^(a + b %i)` to simplify to `%e^(a (cos(b) + %i sin(b)))` if `b` is free of `%i`. See `demoivre`.

`%emode`, when `true`, causes `%e^(%pi %i x)` to be simplified. See `%emode`.

`%enumer`, when `true` causes `%e` to be replaced by 2.718... whenever `numer` is `true`. See `%enumer`.

%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*) Function

`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. Local variables in `block` are not recognized by `freeof` as dummy variables; this is a bug.

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

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

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

```
(%i1) freeof (i, 'sum (f(i), i, 0, n));
(%o1)                          true
(%i2) freeof (x, 'integrate (x^2, x, 0, 1));
(%o2)                          true
(%i3) freeof (x, 'integrate (x^2, x));
(%o3)                          false
```

genfact (*x*, *y*, *z*) Function

Returns the generalized factorial, defined as `x (x-z) (x - 2 z) ... (x - (y - 1) z)`. Thus, for integral *x*, `genfact (x, x, 1) = x!` and `genfact (x, x/2, 2) = x!!`.

imagpart (*expr*) Function

Returns the imaginary part of the expression *expr*.

`imagpart` is a computational function, not a simplifying function.

See also `abs`, `carg`, `polarform`, `rectform`, and `realpart`.

infix (*op*) Function
infix (*op*, *lbp*, *rbp*) Function
infix (*op*, *lbp*, *rbp*, *lpos*, *rpos*, *pos*) Function

Declares *op* to be an infix operator. An infix operator is a function of two arguments, with the name of the function written between the arguments. For example, the subtraction operator `-` is an infix operator.

`infix` (*op*) declares *op* to be an infix operator with default binding powers (left and right both equal to 180) and parts of speech (left and right both equal to `any`).

`infix` (*op*, *lbp*, *rbp*) declares *op* to be an infix operator with stated left and right binding powers and default parts of speech (left and right both equal to `any`).

`infix` (*op*, *lbp*, *rbp*, *lpos*, *rpos*, *pos*) declares *op* to be an infix operator with stated left and right binding powers and parts of speech *lpos*, *rpos*, and *pos* for the left operand, the right operand, and the operator result, respectively.

"Part of speech", in reference to operator declarations, means expression type. Three types are recognized: `expr`, `clause`, and `any`, indicating an algebraic expression, a Boolean expression, or any kind of expression, respectively. Maxima can detect some syntax errors by comparing the declared part of speech to an actual expression.

The precedence of *op* with respect to other operators derives from the left and right binding powers of the operators in question. If the left and right binding powers of

op are both greater the left and right binding powers of some other operator, then *op* takes precedence over the other operator. If the binding powers are not both greater or less, some more complicated relation holds.

The associativity of *op* depends on its binding powers. Greater left binding power (*lbp*) implies an instance of *op* is evaluated before other operators to its left in an expression, while greater right binding power (*rbp*) implies an instance of *op* is evaluated before other operators to its right in an expression. Thus greater *lbp* makes *op* right-associative, while greater *rbp* makes *op* left-associative. If *lbp* is equal to *rbp*, *op* is left-associative.

See also `Syntax`.

Examples:

If the left and right binding powers of *op* are both greater the left and right binding powers of some other operator, then *op* takes precedence over the other operator.

```
(%i1) :lisp (get '$+ 'lbp)
100
(%i1) :lisp (get '$+ 'rbp)
100
(%i1) infix ("##", 101, 101);
(%o1)                           ##
(%i2) "##"(a, b) := sconcat("(", a, ",", b, ")");
(%o2)       (a ## b) := sconcat("(", a, ",", b, ")")
(%i3) 1 + a ## b + 2;
(%o3)                        (a,b) + 3
(%i4) infix ("##", 99, 99);
(%o4)                           ##
(%i5) 1 + a ## b + 2;
(%o5)                        (a+1,b+2)
```

Greater *lbp* makes *op* right-associative, while greater *rbp* makes *op* left-associative.

```
(%i1) infix ("##", 100, 99);
(%o1)                           ##
(%i2) "##"(a, b) := sconcat("(", a, ",", b, ")")$
(%i3) foo ## bar ## baz;
(%o3)                     (foo,(bar,baz))
(%i4) infix ("##", 100, 101);
(%o4)                           ##
(%i5) foo ## bar ## baz;
(%o5)                     ((foo,bar),baz)
```

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

```
(%i1) infix ("##", 100, 99, expr, expr, expr);
(%o1)                           ##
(%i2) if x ## y then 1 else 0;
Incorrect syntax: Found algebraic expression where logical expression expected
if x ## y then
             ^
(%i2) infix ("##", 100, 99, expr, expr, clause);
```

```
(%o2)                          ##
(%i3) if x ## y then 1 else 0;
(%o3)                if x ## y then 1 else 0
```

inflag Option variable

Default value: `false`

When `inflag` is `true`, functions for part extraction inspect the internal form of `expr`.

Note that the simplifier re-orders expressions. Thus `first (x + y)` returns `x` if `inflag` is `true` and `y` if `inflag` is `false`. (`first (y + x)` gives the same results.)

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

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

inpart (*expr, n_1, ..., n_k*) Function

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

Examples:

```
(%i1) x + y + w*z;
(%o1)                        w z + y + x
(%i2) inpart (%, 3, 2);
(%o2)                             z
(%i3) part (%th (2), 1, 2);
(%o3)                             z
(%i4) 'limit (f(x)^g(x+1), x, 0, minus);
                                    g(x + 1)
(%o4)                  limit   f(x)
                       x -> 0-
(%i5) inpart (%, 1, 2);
(%o5)                          g(x + 1)
```

isolate (*expr, x*) Function

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

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

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

Do `example (isolate)` for examples.

isolate_wrt_times Option variable

Default value: `false`

When `isolate_wrt_times` is `true`, `isolate` will also isolate with respect to products. E.g. compare both settings of the switch on

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

(%t2)                          2 a


(%t3)                          2 b


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

                      2
(%o4)                c  + %t3 c + %t2 c + %t4
(%i4) isolate_wrt_times: false$
(%i5) isolate (expand ((a+b+c)^2), c);
                      2
(%o5)                c  + 2 b c + 2 a c + %t4
```

listconstvars Option variable

Default value: `false`

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

listdummyvars Option variable

Default value: `true`

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

```
(%i1) listdummyvars: true$
(%i2) listofvars ('sum(f(i), i, 0, n));
(%o2)                         [i, n]
(%i3) listdummyvars: false$
(%i4) listofvars ('sum(f(i), i, 0, n));
(%o4)                           [n]
```

listofvars (*expr*) Function

Returns a list of the variables in *expr*.

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

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

lfreeof (*list, expr*) Function

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

lopow (*expr, x*) Function

Returns the lowest exponent of *x* which explicitly appears in *expr*. Thus

```
(%i1) lopow ((x+y)^2 + (x+y)^a, x+y);
(%o1)                        min(a, 2)
```

lpart (*label, expr, n_1, ..., n_k*) Function

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

multthru (*expr*) Function
multthru (*expr_1, expr_2*) Function

Multiplies a factor (which should be a sum) of *expr* by the other factors of *expr*. That is, *expr* is *f_1 f_2 ... f_n* where at least one factor, say *f_i*, is a sum of terms. Each term in that sum is multiplied by the other factors in the product. (Namely all the factors except *f_i*). `multthru` does not expand exponentiated sums. This function is the fastest way to distribute products (commutative or noncommutative) over sums. Since quotients are represented as products `multthru` can be used to divide sums by products as well.

`multthru (expr_1, expr_2)` multiplies each term in *expr_2* (which should be a sum or an equation) by *expr_1*. If *expr_1* is not itself a sum then this form is equivalent to `multthru (expr_1*expr_2)`.

```
(%i1) x/(x-y)^2 - 1/(x-y) - f(x)/(x-y)^3;
                        1         x         f(x)
(%o1)               - ----- + -------- - --------
                      x - y          2          3
                                (x - y)    (x - y)
(%i2) multthru ((x-y)^3, %);
                              2
(%o2)               - (x - y)  + x (x - y) - f(x)
(%i3) ratexpand (%);
                             2
(%o3)                      - y  + x y - f(x)
(%i4) ((a+b)^10*s^2 + 2*a*b*s + (a*b)^2)/(a*b*s^2);
```

```
                          10  2              2  2
                 (b + a)   s  + 2 a b s + a  b
(%o4)            ------------------------------
                                  2
                               a b s
(%i5) multthru (%);  /* note that this does not expand (b+a)^10 */
                                          10
                        2   a b   (b + a)
(%o5)                   - + --- + ---------
                        s    2       a b
                             s
(%i6) multthru (a.(b+c.(d+e)+f));
(%o6)            a . f + a . c . (e + d) + a . b
(%i7) expand (a.(b+c.(d+e)+f));
(%o7)         a . f + a . c . e + a . c . d + a . b
```

nounify (*f*) Function

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

nterms (*expr*) Function

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

op (*expr*) Function

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

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

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

`op` evaluates it argument.

See also `args`.

Examples:

```
(%i1) stringdisp: true$
(%i2) op (a * b * c);
(%o2)                          "*"
(%i3) op (a * b + c);
(%o3)                          "+"
(%i4) op ('sin (a + b));
```

```
(%o4)                           sin
(%i5) op (a!);
(%o5)                           "!"
(%i6) op (-a);
(%o6)                           "-"
(%i7) op ([a, b, c]);
(%o7)                           "["
(%i8) op ('(if a > b then c else d));
(%o8)                          "if"
(%i9) op ('foo (a));
(%o9)                           foo
(%i10) prefix (foo);
(%o10)                         "foo"
(%i11) op (foo a);
(%o11)                         "foo"
(%i12) op (F [x, y] (a, b, c));
(%o12)                         F
                                x, y
(%i13) op (G [u, v, w]);
(%o13)                          G
```

operatorp (*expr, op*) Function
operatorp (*expr*, [*op_1, ..., op_n*]) Function

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

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

optimize (*expr*) Function

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

optimprefix Option variable

Default value: `%`

`optimprefix` is the prefix used for generated symbols by the `optimize` command.

ordergreat (*v_1, ..., v_n*) Function
orderless (*v_1, ..., v_n*) Function

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

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

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

See also `ordergreatp`.

ordergreatp (*expr_1, expr_2*) Function
orderlessp (*expr_1, expr_2*) Function

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

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

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

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

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

For non-atomic expressions, the canonical ordering is derived from the ordering for atoms. For the built-in `+` `*` and `^` operators, the ordering is not easily summarized. For other built-in operators and all other functions and operators, expressions are ordered by their arguments (beginning with the first argument), then by the name of the operator or function. In the case of subscripted expressions, the subscripted symbol is considered the operator and the subscript is considered an argument.

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

See also `sort`.

Examples:

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

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

Effect of `ordergreat` and `orderless` functions.

```
(%i1) sort ([M, H, K, T, E, W, G, A, P, J, S]);
(%o1)           [A, E, G, H, J, K, M, P, S, T, W]
(%i2) ordergreat (S, J);
(%o2)                         done
(%i3) orderless (M, H);
(%o3)                         done
```

```
(%i4) sort ([M, H, K, T, E, W, G, A, P, J, S]);
(%o4)           [M, H, A, E, G, K, P, T, W, J, S]
```

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

```
(%i1) sort ([aa, foo, bar, bb, baz, quux, cc, dd, A1, B1, C1]);
(%o1)   [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)]
(%i2) sort ([foo(1), X[1], X[k], foo(k), 1, k]);
(%o2)            [1, foo(1), X , k, foo(k), X ]
                                1              k
```

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`.

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

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

`piece` holds the last expression selected when using the `part` functions. It is set during the execution of the function and thus may be referred to in the function itself as shown below.

If `partswitch` is set to `true` then `end` is returned when a selected part of an expression doesn't exist, otherwise an error message is given.

Example: `part (z+2*y, 2, 1)` yields 2.

`example (part)` displays additional examples.

partition (*expr, x*) Function

Returns a list of two expressions. They are (1) the factors of *expr* (if it is a product), the terms of *expr* (if it is a sum), or the list (if it is a list) which don't contain *x* and, (2) the factors, terms, or list which do.

```
(%i1) partition (2*a*x*f(x), x);
(%o1)                      [2 a, x f(x)]
(%i2) partition (a+b, x);
(%o2)                       [b + a, 0]
(%i3) partition ([a, b, f(a), c], a);
(%o3)                  [[b, c], [a, f(a)]]
```

partswitch Option variable

Default value: `false`

When `partswitch` is `true`, `end` is returned when a selected part of an expression doesn't exist, otherwise an error message is given.

pickapart (*expr*, *n*) Function

Assigns intermediate expression labels to subexpressions of *expr* at depth *n*, an integer. Subexpressions at greater or lesser depths are not assigned labels. `pickapart` returns an expression in terms of intermediate expressions equivalent to the original expression *expr*.

See also `part`, `dpart`, `lpart`, `inpart`, and `reveal`.

Examples:

```
(%i1) expr: (a+b)/2 + sin (x^2)/3 - log (1 + sqrt(x+1));
                                          2
                                       sin(x )   b + a
(%o1)       - log(sqrt(x + 1) + 1) + ------- + -----
                                         3        2
(%i2) pickapart (expr, 0);

                                          2
                                       sin(x )   b + a
(%t2)       - log(sqrt(x + 1) + 1) + ------- + -----
                                         3        2

(%o2)                           %t2
(%i3) pickapart (expr, 1);

(%t3)                - log(sqrt(x + 1) + 1)


                                 2
                              sin(x )
(%t4)                         -------
                                 3


                               b + a
(%t5)                          -----
                                 2

(%o5)                    %t5 + %t4 + %t3
(%i5) pickapart (expr, 2);

(%t6)                log(sqrt(x + 1) + 1)


                                 2
(%t7)                         sin(x )
```

```
(%t8)                          b + a

                           %t8   %t7
(%o8)                      --- + --- - %t6
                            2     3
(%i8) pickapart (expr, 3);

(%t9)                      sqrt(x + 1) + 1


                                 2
(%t10)                           x

                   b + a                sin(%t10)
(%o10)             ----- - log(%t9) + ---------
                     2                     3
(%i10) pickapart (expr, 4);

(%t11)                       sqrt(x + 1)

                        2
                   sin(x )   b + a
(%o11)             ------- + ----- - log(%t11 + 1)
                     3         2
(%i11) pickapart (expr, 5);

(%t12)                           x + 1

                    2
               sin(x )   b + a
(%o12)         ------- + ----- - log(sqrt(%t12) + 1)
                 3         2
(%i12) pickapart (expr, 6);
                  2
             sin(x )   b + a
(%o12)       ------- + ----- - log(sqrt(x + 1) + 1)
               3         2
```

piece 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 (*expr*) Function

Returns an expression `r %e^(%i theta)` equivalent to *expr*, such that `r` and `theta` are purely real.

powers (*expr, x*) Function

Gives the powers of *x* occuring in *expr*.

`load (powers)` loads this function.

product (*expr, i, i_0, i_1*) Function

Represents a product of the values of *expr* as the index *i* varies from *i_0* to *i_1*. The noun form `'product` is displayed as an uppercase letter pi.

`product` evaluates *expr* and lower and upper limits *i_0* and *i_1*, `product` quotes (does not evaluate) the index *i*.

If the upper and lower limits differ by an integer, *expr* is evaluated for each value of the index *i*, and the result is an explicit product.

Otherwise, the range of the index is indefinite. Some rules are applied to simplify the product. When the global variable `simpproduct` is `true`, additional rules are applied. In some cases, simplification yields a result which is not a product; otherwise, the result is a noun form `'product`.

See also `nouns` and `evflag`.

Examples:

```
(%i1) product (x + i*(i+1)/2, i, 1, 4);
(%o1)           (x + 1) (x + 3) (x + 6) (x + 10)
(%i2) product (i^2, i, 1, 7);
(%o2)                         25401600
(%i3) product (a[i], i, 1, 7);
(%o3)                 a  a  a  a  a  a  a
                       1  2  3  4  5  6  7
(%i4) product (a(i), i, 1, 7);
(%o4)          a(1) a(2) a(3) a(4) a(5) a(6) a(7)
(%i5) product (a(i), i, 1, n);
                             n
                           /===\
                            ! !
(%o5)                       ! !  a(i)
                            ! !
                           i = 1
(%i6) product (k, k, 1, n);
                              n
                            /===\
                             ! !
(%o6)                        ! !  k
                             ! !
                            k = 1
(%i7) product (k, k, 1, n), simpproduct;
(%o7)                         n!
(%i8) product (integrate (x^k, x, 0, 1), k, 1, n);
                             n
                           /===\
                            ! !    1
(%o8)                       ! !  -----
```

```
                                ! !  k + 1
                               k = 1
(%i9) product (if k <= 5 then a^k else b^k, k, 1, 10);
                                 15  40
(%o9)                           a   b
```

realpart (*expr*) Function

Returns the real part of *expr*. `realpart` and `imagpart` will work on expressions involving trigonometic 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
                                      h
(%i2) dpart (dpart (expr, 1, 1), 2, 2);
                          """""""    a d - b c
(%o2)                sin("%pi x") + ---------
                          """""""      """"
                                       " 2"
                                       "h "
                                       """"
(%i3) expr2: lpart (BAR, lpart (FOO, %, 1), 2);
                    FOO"""""""""""   BAR"""""""""
                    "    """""""  "   "a d - b c"
(%o3)               "sin("%pi x")" + "---------"
                    "    """""""  "   "  """"   "
                    """""""""""""""   "  " 2"   "
                                      "  "h "   "
                                      "  """"   "
                                      """""""""""
(%i4) rembox (expr2, unlabelled);
                                    BAR"""""""""
                     FOO""""""""""   "a d - b c"
```

```
(%o4)              "sin(%pi x)" + "---------"
                   """""""""""""   "    2    "
                                   "   h     "
                                   """""""""""
(%i5) rembox (expr2, FOO);
                                   BAR"""""""""
                       """""""""   "a d - b c"
(%o5)              sin("%pi x") + "---------"
                       """""""""   "  """"   "
                                   "  " 2"   "
                                   "  "h "   "
                                   "  """"   "
                                   """""""""""
(%i6) rembox (expr2, BAR);
                   FOO"""""""""""
                   "    """""""" "   a d - b c
(%o6)              "sin("%pi x")" + ---------
                   "    """""""" "     """"
                   """""""""""""""     " 2"
                                       "h "
                                       """"
(%i7) rembox (expr2);
                                   a d - b c
(%o7)                sin(%pi x) + ---------
                                       2
                                      h
```

sum (*expr, i, i_0, i_1*) Function

Represents a summation of the values of *expr* as the index *i* varies from *i_0* to *i_1*. The noun form `'sum` is displayed as an uppercase letter sigma.

`sum` evaluates its summand *expr* and lower and upper limits *i_0* and *i_1*, `sum` quotes (does not evaluate) the index *i*.

If the upper and lower limits differ by an integer, the summand *expr* is evaluated for each value of the summation index *i*, and the result is an explicit sum.

Otherwise, the range of the index is indefinite. Some rules are applied to simplify the summation. When the global variable `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) `cauchysum` is `true`, a product of summations is expressed as a Cauchy product, in which the index of the inner summation is a function of the index of the outer one, rather than varying independently.

The global variable `genindex` is the alphabetic prefix used to generate the next index of summation, when an automatically generated index is needed.

`gensumnum` is the numeric suffix used to generate the next index of summation, when an automatically generated index is needed. When `gensumnum` is `false`, an automatically-generated index is only `genindex` with no numeric suffix.

See also `sumcontract`, `intosum`, `bashindices`, `niceindices`, `nouns`, `evflag`, and `zeilberger`.

Examples:

```
(%i1) sum (i^2, i, 1, 7);
(%o1)                          140
(%i2) sum (a[i], i, 1, 7);
(%o2)           a  + a  + a  + a  + a  + a  + a
                 7    6    5    4    3    2    1
(%i3) sum (a(i), i, 1, 7);
(%o3)    a(7) + a(6) + a(5) + a(4) + a(3) + a(2) + a(1)
(%i4) sum (a(i), i, 1, n);
                             n
                            ====
                            \
(%o4)                        >    a(i)
                            /
                            ====
                            i = 1
(%i5) sum (2^i + i^2, i, 0, n);
                           n
                          ====
                          \       i    2
(%o5)                      >    (2  + i )
                          /
                          ====
                          i = 0
(%i6) sum (2^i + i^2, i, 0, n), simpsum;
                                3      2
                    n + 1   2 n  + 3 n  + n
(%o6)              2      + --------------- - 1
                                    6
(%i7) sum (1/3^i, i, 1, inf);
                             inf
                             ====
                             \     1
(%o7)                         >    --
                             /      i
                             ====  3
                             i = 1
(%i8) sum (1/3^i, i, 1, inf), simpsum;
                               1
(%o8)                          -
                               2
(%i9) sum (i^2, i, 1, 4) * sum (1/i^2, i, 1, inf);
                                inf
                                ====
                                \     1
(%o9)                        30  >    --
                                /      2
```

```
                                ====  i
                                i = 1
(%i10) sum (i^2, i, 1, 4) * sum (1/i^2, i, 1, inf), simpsum;
                                     2
(%o10)                          5 %pi
(%i11) sum (integrate (x^k, x, 0, 1), k, 1, n);
                                n
                               ====
                               \       1
(%o11)                          >    -----
                               /     k + 1
                               ====
                               k = 1
(%i12) sum (if k <= 5 then a^k else b^k, k, 1, 10);
          10    9    8    7    6    5    4    3    2
(%o12)   b   + b  + b  + b  + b  + a  + a  + a  + a  + a
```

lsum (*expr*, *x*, *L*) Function

Represents the sum of *expr* for each element *x* in *L*.

A noun form `'lsum` is returned if the argument *L* does not evaluate to a list.

Examples:

```
(%i1) lsum (x^i, i, [1, 2, 7]);
                                7    2
(%o1)                          x  + x  + x
(%i2) lsum (i^2, i, rootsof (x^3 - 1));
                      ====
                      \      2
(%o2)                  >    i
                      /
                      ====
                                    3
                      i in rootsof(x  - 1)
```

verbify (*f*) Function

Returns the verb form of the function name *f*.

See also `verb`, `noun`, and `nounify`.

Examples:

```
(%i1) verbify ('foo);
(%o1)                           foo
(%i2) :lisp $%
$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.

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 (*expr*) Function
expand (*expr, p, n*) 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`.

The `expand` flag used with `ev` causes expansion.

The file '`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")`.

expandwrt (*expr, x_1, ..., x_n*) Function

Expands expression `expr` with respect to the variables *x_1*, ..., *x_n*. All products involving the variables appear explicitly. The form returned will be free of products of sums of expressions that are not free of the variables. *x_1*, ..., *x_n* may be variables, operators, or expressions.

By default, denominators are not expanded, but this can be controlled by means of the switch `expandwrt_denom`.

This function is autoloaded from '`simplification/stopex.mac`'.

expandwrt_denom Option variable

Default value: `false`

`expandwrt_denom` controls the treatment of rational expressions by `expandwrt`. If `true`, then both the numerator and denominator of the expression will be expanded according to the arguments of `expandwrt`, but if `expandwrt_denom` is `false`, then only the numerator will be expanded in that way.

expandwrt_factored (*expr, x_1, ..., x_n*) Function

is similar to `expandwrt`, but treats expressions that are products somewhat differently. `expandwrt_factored` expands only on those factors of `expr` that contain the variables *x_1*, ..., *x_n*.

This function is autoloaded from '`simplification/stopex.mac`'.

expon Option variable

Default value: 0

`expon` is the exponent of the largest negative power which is automatically expanded (independent of calls to `expand`). For example, if `expon` is 4 then `(x+1)^(-5)` will not be automatically expanded.

exponentialize (*expr*) Function

exponentialize Option variable

The function `exponentialize (expr)` converts circular and hyperbolic functions in *expr* to exponentials, without setting the global variable `exponentialize`.

When the variable `exponentialize` is `true`, all circular and hyperbolic functions are converted to exponential form. The default value is `false`.

`demoivre` converts complex exponentials into circular functions. `exponentialize` and `demoivre` cannot both be true at the same time.

expop Option variable

Default value: 0

`expop` is the highest positive exponent which is automatically expanded. Thus `(x + 1)^3`, when typed, will be automatically expanded only if `expop` is greater than or equal to 3. If it is desired to have `(x + 1)^n` expanded where `n` is greater than `expop` then executing `expand ((x + 1)^n)` will work only if `maxposex` is not less than n.

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 (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 `(declare (x, mainvar), expand ((x+y)^4))`. (Note: Care should be taken if you elect to use the above feature. E.g., if you subtract an expression in which `x` is a `mainvar` from one in which `x` isn't a `mainvar`, resimplification e.g. with `ev (expr, simp)` may be necessary if cancellation is to occur. Also, if you save an expression in which `x` is a `mainvar`, you probably should also save `x`.)

maxapplydepth Option variable

Default value: 10000

`maxapplydepth` is the maximum depth to which `apply1` and `apply2` will delve.

maxapplyheight Option variable

Default value: 10000

`maxapplyheight` is the maximum height to which `applyb1` will reach before giving up.

maxnegex Option variable

Default value: 1000

`maxnegex` is the largest negative exponent which will be expanded by the `expand` command (see also `maxposex`).

maxposex Option variable

Default value: 1000

`maxposex` is the largest exponent which will be expanded with the `expand` command (see also `maxnegex`).

multiplicative Declaration

`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 `product (x[i], i, m, n)`.

negdistrib Option variable

Default value: `true`

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

negsumdispflag Option variable

Default value: `true`

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

noeval Special symbol

`noeval` suppresses the evaluation phase of `ev`. This is useful in conjunction with other switches and in causing expressions to be resimplified without being reevaluated.

noun Declaration

`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 Option variable

Default value: `false`

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

nouns Special symbol

`nouns` is an `evflag`. When used as an option to the `ev` command, `nouns` converts all "noun" forms occurring in the expression being `ev`'d to "verbs", i.e., evaluates them. See also `noun`, `nounify`, `verb`, and `verbify`.

numer Special symbol

`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.

numerval (*x_1, expr_1, ..., var_n, expr_n*) Function

Declares 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 System variable

`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 Option variable

Default value: `true`

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

outative Declaration

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

1. If `f` is univariate, whenever the simplifier encounters `f` applied to a product, that product will be partitioned into factors that are constant and factors that are not and the constant factors will be pulled out. E.g., `f(a*x)` will simplify to `a*f(x)` where `a` is a constant. Non-atomic constant factors will not be pulled out.
2. If `f` is a function of 2 or more arguments, outativity is defined as in the case of `sum` or `integrate`, i.e., `f (a*g(x), x)` will simplify to `a * f(g(x), x)` for `a` free of `x`.

`sum`, `integrate`, and `limit` are all `outative`.

posfun Declaration

`declare (f, posfun)` declares `f` to be a positive function. `is (f(x) > 0)` yields `true`.

radcan (*expr*) Function

Simplifies *expr*, which can contain logs, exponentials, and radicals, by converting it into a form which is canonical over a large class of expressions and a given ordering of variables; that is, all functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, `radcan` produces a regular form. Two equivalent expressions in this class do not necessarily have the same appearance, but their difference can be simplified by `radcan` to zero.

For some expressions `radcan` is quite time consuming. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial-fraction expansions of exponents.

When `%e_to_numlog` is `true`, `%e^(r*log(expr))` simplifies to `expr^r` if `r` is a rational number.

When `radexpand` is `false`, certain transformations are inhibited. `radcan (sqrt (1-x))` remains `sqrt (1-x)` and is not simplified to `%i sqrt (x-1)`. `radcan (sqrt (x^2 - 2*x + 11))` remains `sqrt (x^2 - 2*x + 1)` and is not simplified to `x - 1`.

`example (radcan)` displays some examples.

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 (16*x^2)` simplifies to `4*x`.

More particularly, consider `sqrt (x^2)`.

- If `radexpand` is `all` or `assume (x > 0)` has been executed, `sqrt(x^2)` simplifies to `x`.
- If `radexpand` is `true` and `domain` is `real` (its default), `sqrt(x^2)` simplifies to `abs(x)`.
- If `radexpand` is `false`, or `radexpand` is `true` and `domain` is `complex`, `sqrt(x^2)` is not simplified.

Note that `domain` only matters when `radexpand` is `true`.

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, rassociative)` tells the Maxima simplifier that `g` is right-associative. E.g., `g(g(a, b), g(c, d))` simplifies to `g(a, g(b, g(c, d)))`.

scsimp (*expr*, *rule_1*, ..., *rule_n*) Function

Sequential Comparative Simplification (method due to Stoute). `scsimp` attempts to simplify *expr* according to the rules *rule_1*, ..., *rule_n*. If a smaller expression is obtained, the process repeats. Otherwise after all simplifications are tried, it returns the original answer.

`example (scsimp)` displays some examples.

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:

```
(%i1) sumexpand: true$
(%i2) sum (f (i), i, 0, m) * sum (g (j), j, 0, n);
                       m      n
                      ====   ====
                      \      \
(%o2)                  >      >     f(i1) g(i2)
                      /      /
                      ====   ====
                      i1 = 0 i2 = 0
(%i3) sum (f (i), i, 0, m)^2;
                       m      m
                      ====   ====
                      \      \
(%o3)                  >      >     f(i3) f(i4)
                      /      /
                      ====   ====
                      i3 = 0 i4 = 0
```

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 (*expr*) Function

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

8 Plotting

8.1 Functions and Variables for Plotting

contour_plot (*expr*, *x_range*, *y_range*, *options*, ...) Function

Plots the contours (curves of equal value) of *expr* over the region *x_range* by *y_range*. Any additional arguments are treated the same as in `plot3d`.

`contour_plot` only works when the plot format is `gnuplot` or `gnuplot_pipes`.

See also `implicit_plot`.

Examples:

```
(%i1) contour_plot (x^2 + y^2, [x, -4, 4], [y, -4, 4]);
(%o1)
(%i2) contour_plot (sin(y) * cos(x)^2, [x, -4, 4], [y, -4, 4]);
(%o2)
(%i3) F(x, y) := x^3 + y^2;
                                      3    2
(%o3)                    F(x, y) := x  + y
(%i4) contour_plot (F, [u, -4, 4], [v, -4, 4]);
(%o4)
(%i5) contour_plot (F, [u, -4, 4], [v, -4, 4], [gnuplot_preamble,
      "set size ratio -1"]);
(%o5)
(%i6) set_plot_option ([gnuplot_preamble,
                        "set cntrparam levels 12"])$

(%i7) contour_plot (F, [u, -4, 4], [v, -4, 4]);
```

in_netmath Option variable

Default value: `false`

When `in_netmath` is `true`, `plot3d` prints OpenMath output to the console if `plot_format` is `openmath`; otherwise `in_netmath` (even if `true`) has no effect. `in_netmath` has no effect on `plot2d`.

plot2d (*expr*, *x_range*, ..., *options*, ...) Function
plot2d ([*expr_1*, ..., *expr_n*], ..., *options*, ...) Function
plot2d ([*expr_1*, ..., *expr_n*], *x_range*,..., *options*, ...) Function

Where *expr*, *expr_1*, ..., *expr_n* can be either expressions, or Maxima or Lisp functions or operators, 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.

`plot2d` plots one expression *expr* or several expressions `[`*name_1*`, ..., `*name_n*`]`. The expressions that are not of the parametic or discrete types should all depend only on one variable *var* and it will be mandatory the use of *x_range* to name that

variable and gives its minimum and maximum values, using the syntax: `[`*variable*`,` *min*`,` *max*`]`. The plot will show the horizontal axis bound by the values of *min* and *max*.

A expression to be plotted can also be given in the discrete or parametric forms. Namely, as a list starting with the word "discrete" or "parametric". The keyword *discrete* must be followed by two lists of values, both with the same length, which are the horizontal and vertical coordinates of a set of points; alternatively, the coordinates of each point can be put into a list with two values, and all the coordinates of the points should be inside another list. The keyword *parametric* must be followed by two expressions *x_expr* and *y_expr*, and a range of the form `[`*param*`,` *min*`,` *max*`]`. The two expressions must depend only on the parameter *param*, and the plot will show the path traced out by the point with coordinates (*x_expr*, *y_expr*) as *param* increases from *min* to *max*.

The range of the vertical axis is not mandatory. It is one more of the options for the command, with the syntax: `[y,` *min*`,` *max*`]`. If that option is used, the plot will show that entire range, even if the expressions do not reach all that range. Otherwise, if a vertical range is not specified by `set_plot_option`, the boundaries of the vertical axis will be set up automatically.

All other options should also be lists, starting with the name of the option. The option *xlabel* can be used to give a label for the horizontal axis; if that option is not used, the horizontal axis will be labeled with the name of the variable specified in *x_range*, or with the expression *x_expr* in the case of just one parametric expression, or it will be left blank otherwise.

A label for the vertical axis can be given with the *ylabel* option. If there is only one expression to be plotted and the *ylabel* option was not used, the vertical axis will be labeled with that expression, unless it is too large, or with the expression *y_expr* if the expression is parametric, or with the text "discrete data" if the expression is discrete.

The options `[logx]` and `[logy]` do not need any parameters. They will make the horizontal and vertical axes be scaled logarithmically.

If there are several expressions to be plotted, a legend will be written to identiy 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.

By default, the expressions are plotted as a set of line segments joining adjacent points within a set of points which is either given in the *discrete* form, or calculated automatically from the expression given, using an algorithm that automatically adapts the steps among points using as an initial estimate of the total number of points the value set with the *nticks* option. The option *style* can be used to make one of the expressions to be represented as a set of isolated points, or as points and line segments.

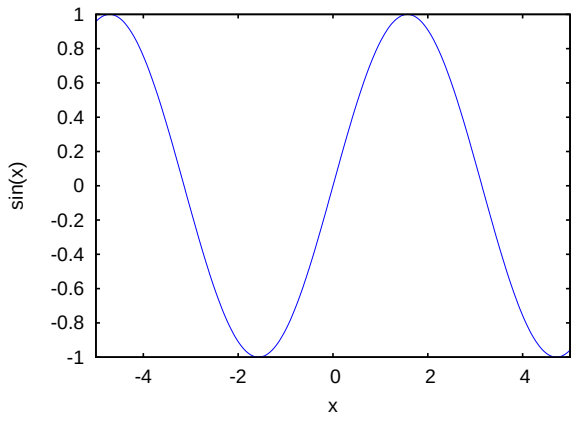
There are several global options stored in the list *plot_options* which can be modified with the function `set_plot_option`; any of those global options can be overriden with options given in the *plot2d* command.

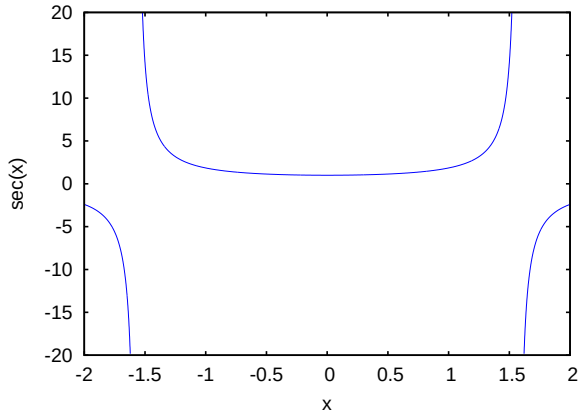
A function to be plotted may be specified as the name of a Maxima or Lisp function or operator, a Maxima lambda expression, or a general Maxima expression. If specified as a name or a lambda expression, the function must be a function of one argument.

Examples:

Plots of common functions.

```
(%i1) plot2d (sin(x), [x, -5, 5])$

(%i2) plot2d (sec(x), [x, -2, 2], [y, -20, 20], [nticks, 200])$
```

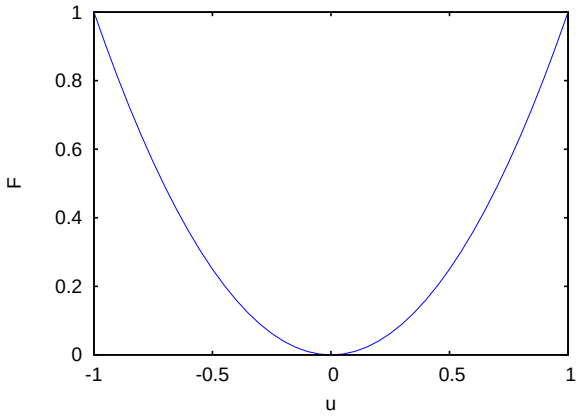


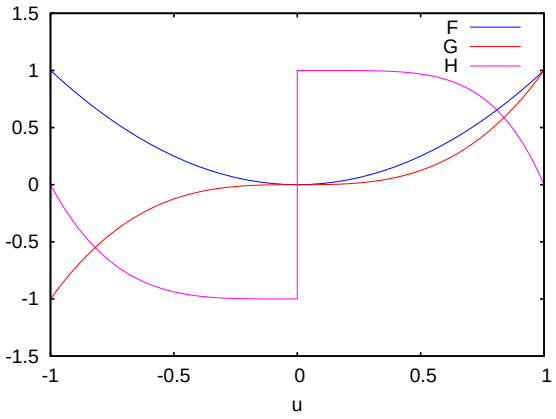


Plotting functions by name.

```
(%i3) F(x) := x^2 $

(%i4) :lisp (defun |$g| (x) (m* x x x))

$g
(%i5) H(x) := if x < 0 then x^4 - 1 else 1 - x^5 $

(%i6) plot2d (F, [u, -1, 1])$
```

```
(%i7) plot2d ([F, G, H], [u, -1, 1], [y, -1.5, 1.5])$
```

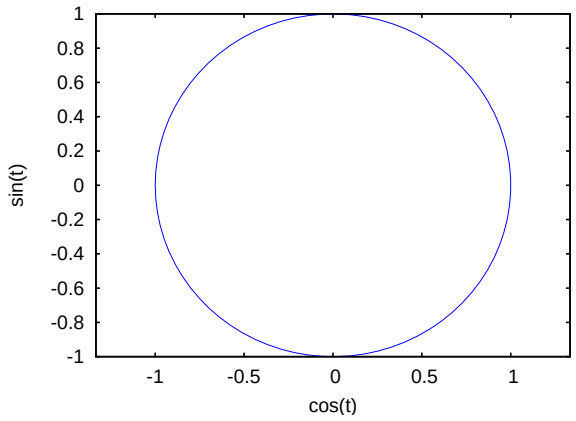




We can plot a circle using a parametric plot with a parameter t. It is not necessary to give a range for the horizontal range, since the range of the parameter t determines the domain. However, since the graph's horizontal and vertical axes lengths are in the 4 to 3 proportion, we will use the *xrange* option to obtain the same scaling in both axes:

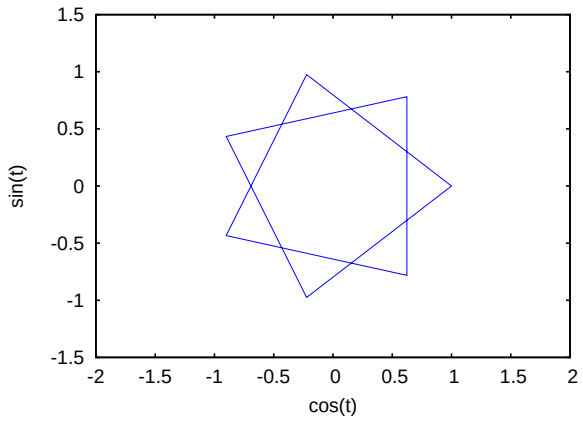
```
(%i8) plot2d ([parametric, cos(t), sin(t), [t,-%pi,%pi],
```

```
[nticks,80]], [x, -4/3, 4/3])$
```



If we repeat that plot with only 8 points and extending the range of the parameter to give two turns, we will obtain the plot of a star:

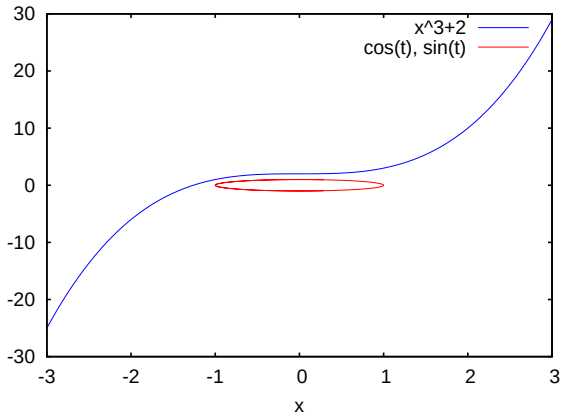
```
(%i9) plot2d ([parametric, cos(t), sin(t), [t, -%pi*2, %pi*2],
        [nticks, 8]], [x, -2, 2], [y, -1.5, 1.5])$
```



Combination of an ordinary plot of a cubic polynomial with a parametric plot of a circle:

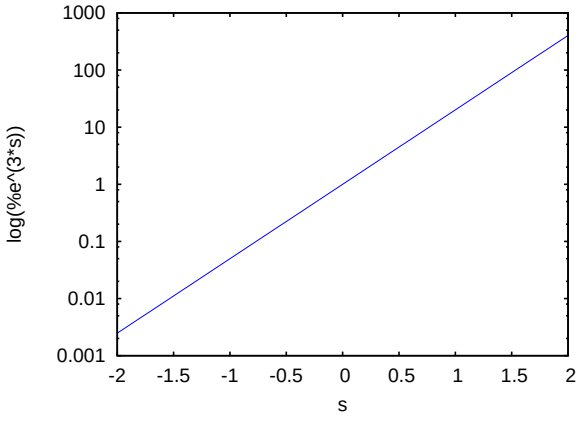
```
(%i10) plot2d ([x^3+2, [parametric, cos(t), sin(t), [t, -5, 5],
```

```
[nticks, 80]]], [x, -3, 3])$
```



Example of a logarithmic plot:

```
(%i11) plot2d (exp(3*s), [s, -2, 2], [logy])$
```

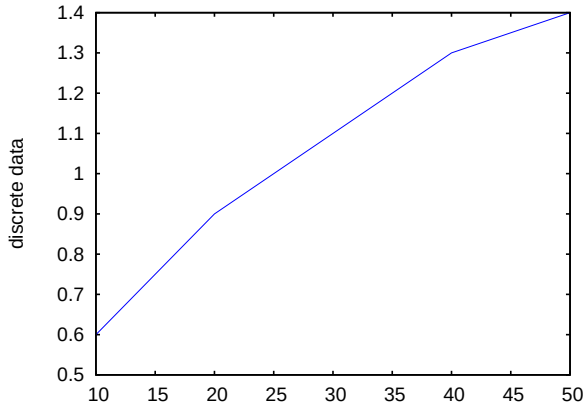


To show some examples of discrete plots, we will start by entering the coordinates of 5 points, in the two different ways that can be used:

```
(%i12) xx:[10, 20, 30, 40, 50]$
(%i13) yy:[.6, .9, 1.1, 1.3, 1.4]$
(%i14) xy:[[10,.6], [20,.9], [30,1.1], [40,1.3], [50,1.4]]$
```

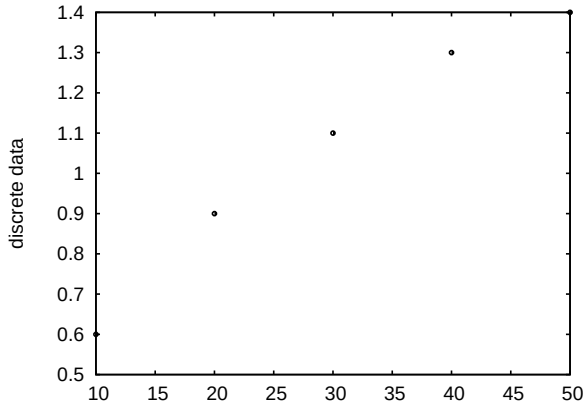
To plot those data points, joined with line segments, we use:

```
(%i15) plot2d([discrete,xx,yy])$
```



We will now show the plot with only points, and illustrating the use of the second way of giving the points coordinates:

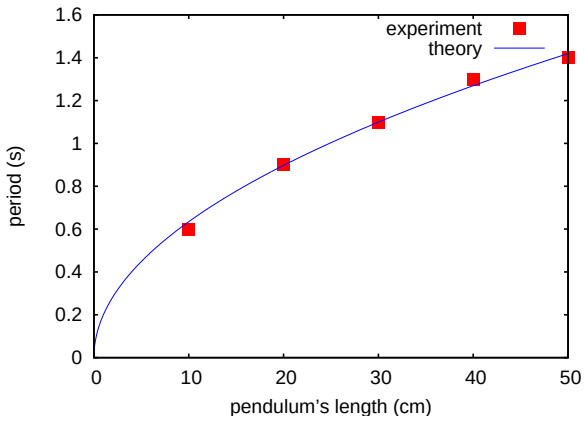
```
(%i16) plot2d([discrete, xy], [style, points])$
```



The plot of the data points can be shown together with a plot of the theoretical function that predicts the data:

```
(%i17) plot2d([[discrete,xy], 2*%pi*sqrt(1/980)], [1,0,50],
        [style, [points,5,2,6], [lines,1,1]],
        [legend,"experiment","theory"],
```

```
[xlabel,"pendulum's length (cm)"], [ylabel,"period (s)"])$
```



The meaning of the three numbers after the "points" style option are as follows; 5: radius of the points, 2: index of color used (red), 6: type of objects used (solid squares). The two numbers after the "lines" style option give the thickness of the line (1 point) and the color (1 corresponds to blue).

See also `plot_options`, which describes plotting options and has more examples.

xgraph_curves (*list*) Function

graphs the list of 'point sets' given in list by using xgraph. If the program xgraph is not installed, this command will fail.

A point set may be of the form

```
[x0, y0, x1, y1, x2, y2, ...]
```

or

```
[[x0, y0], [x1, y1], ...]
```

A point set may also contain symbols which give labels or other information.

```
xgraph_curves ([pt_set1, pt_set2, pt_set3]);
```

graph the three point sets as three curves.

```
pt_set: append (["NoLines: True", "LargePixels: true"],
                                [x0, y0, x1, y1, ...]);
```

would make the point set (and subsequent ones), have no lines between points, and to use large pixels. See the man page on xgraph for more options to specify.

```
pt_set: append ([concat ("\"", "x^2+y")], [x0, y0, x1, y1, ...]);
```

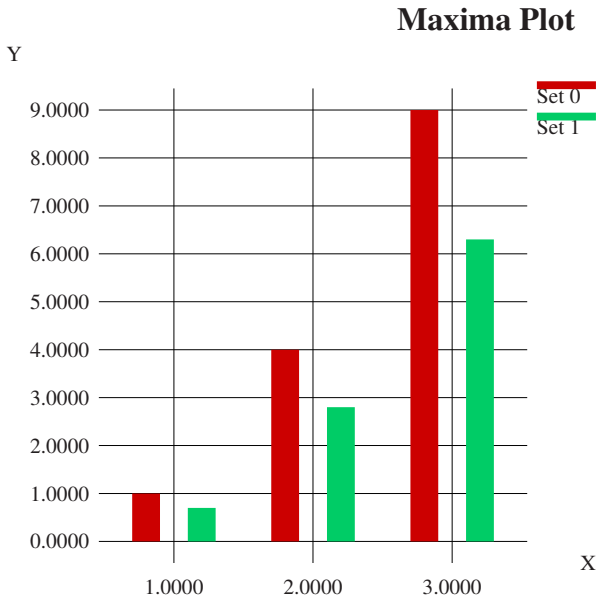
would make there be a "label" of "x^2+y" for this particular point set. The " at the beginning is what tells xgraph this is a label.

```
pt_set: append ([concat ("TitleText: Sample Data")], [x0, ...])$
```

would make the main title of the plot be "Sample Data" instead of "Maxima Plot".

To make a bar graph with bars which are 0.2 units wide, and to plot two possibly different such bar graphs:

```
(%i1) xgraph_curves ([append (["BarGraph: true", "NoLines: true",
      "BarWidth: .2"], create_list ([i - .2, i^2], i, 1, 3)),
      append (["BarGraph: true", "NoLines: true", "BarWidth: .2"],
      create_list ([i + .2, .7*i^2], i, 1, 3))]);
```



A temporary file '`xgraph-out`' is used.

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`.

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

The plot options which are recognized by `plot2d` and `plot3d` are the following:

- Option: `plot_format`

  Determines which graphic interface is used by `plot2d` and `plot3d`.

  - Value: `gnuplot` default on Windows

    Gnuplot is the most advanced plotting package among the packages available in Maxima. It requires an external gnuplot installation.

  - Value: `gnuplot_pipes` default on non-Windows platforms

    Similar to the `gnuplot` format except that communication with gnuplot is done through a pipe. It should be used to plot on screen, for plotting to files it is better to use the `gnuplot` format.

- Value: `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 Tcl/Tk.

- Value: `openmath`

  Openmath is a Tcl/Tk GUI plotting program. This format is provided by Xmaxima, which is distributed together with Maxima; in order to use this format you should install the package Xmaxima, and it will work not only from Xmaxima itself, but also from the command line and other GUI's for Maxima.

- Option: `run_viewer`

  Controls whether or not the appropriate viewer for the plot format should be run.

  - Default value: `true`

    Execute the viewer program.

  - Value: `false`

    Do not execute the viewer program.

- Option: `y`

  The vertical range of the plot.

  Example:

  ```
  [y, - 3, 3]
  ```

  Sets the vertical range to [-3, 3].

- Option: `plot_realpart`

  When `plot_realpart` is `true`, the real part of a complex value x is plotted; this is equivalent to plotting `realpart(`x`)` instead of x. Otherwise, only values with imaginary part equal to 0 are plotted, and complex values are ignored.

  Example:

  ```
  plot2d (log(x), [x, -5, 5], [plot_realpart, false]);
  plot2d (log(x), [x, -5, 5], [plot_realpart, true]);
  ```

  The default value is `false`.

- Option: `nticks`

  In plot2d, it is gives the initial number of points used by the adaptive plotting routine for plotting functions. It is also the number of points that will be shown in a parametric plot.

  Example:

  ```
  [nticks, 20]
  ```

  The default for `nticks` is 10.

- Option: `adapt_depth`

  The maximum number of splittings used by the adaptive plotting routine.

  Example:

```
[adapt_depth, 5]
```

The default for `adapt_depth` is 10.

- Option: `xlabel`

  The label for the horizontal axis in a 2d plot.

  Example:

  ```
  [xlabel, "Time in seconds"]
  ```

- Option: `ylabel`

  The label of the vertical axis in a 2d plot.

  Example:

  ```
  [ylabel, "Temperature"]
  ```

- Option: `logx`

  It makes the horizontal axis of a 2d plot to be rendered in a logarithmic scale. It does not need any aditional parameters.

- Option: `logy`

  It makes the vertical axis of a 2d plot to be rendered in a logarithmic scale. It does not need any aditional parameters.

- Option: `legend`

  The labels for the various expressions in a 2d plot with several expressions. If there are more expressions than the number of labels given, they will be repeated. If legend is followed by the word *false*, no legend 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.

  Example:

  ```
  [legend, "Set 1", "Set 2", "Set 3"]
  ```

- Option: `box`

  Currently, this option can only be followed by the word *false*, and it will be used to supress the box around the plot.

  Example:

  ```
  [box, false]
  ```

- Option: `style`

  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 aditional parameters. *lines* accepts one or two numbers: the width of the line and an integer that identifies a color. The default color codes are: 1: blue, 2: red, 3: magenta, 4: orange, 5: brown, 6: lime and 7: aqua. If you use Gnuplot with a terminal different than X11, those colors might be different; for example, if you use the option [*gnuplot_term*,*ps*], color index 4 will correspond to black, instead of orange.

points accepts one two or three parameters; the first parameter is the radius of the points, the second parameter is an integer that selects the color, using the same code used for *lines* and the third parameter is currently used only by Gnuplot and it corresponds to several objects instead of points. The default types of objects are: 1: filled circles, 2: open circles, 3: plus signs, 4: x, 5: *, 6: filled squares, 7: open squares, 8: filled triangles, 9: open triangles, 10: filled inverted triangles, 11: open inverted triangles, 12: filled lozenges and 13: open lozenges.

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

Example:

```
[style,[lines,2,3],[points,1,4,3]]
```

This will plot the first (and third, fifth, etc) expression with magenta line segments of width 2, and the second (and fourth, sixth, etc) expression with orange plus signs of size 1 (orange circles in the case of Openmath).

The default for the style option is *lines* with a width of 1, and different colors.

- Option: `grid`

  Sets the number of grid points to use in the x- and y-directions for three-dimensional plotting.

  Example:

  ```
  [grid, 50, 50]
  ```

  sets the grid to 50 by 50 points. The default grid is 30 by 30.

- Option: `transform_xy`

  Allows transformations to be applied to three-dimensional plots.

  Example:

  ```
  [transform_xy, false]
  ```

  The default `transform_xy` is `false`. If it is not `false`, it should be the output of

  ```
  make_transform([x,y,z], f1(x,y,z), f2(x,y,z), f3(x,y,z))$
  ```

  The `polar_xy` transformation is built in. It gives the same transformation as

  ```
  make_transform ([r, th, z], r*cos(th), r*sin(th), z)$
  ```

Gnuplot options:

There are several plot options specific to gnuplot. Some of these options are raw gnuplot commands, specified as strings. Refer to the gnuplot documentation for more details.

- Option: `gnuplot_term`

  Sets the output terminal type for gnuplot.

  - Default value: `default`

    Gnuplot output is displayed in a separate graphical window.

  - Value: `dumb`

    Gnuplot output is displayed in the Maxima console by an "ASCII art" approximation to graphics.

  - Value: `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.

  - Value: 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.

- Option: `gnuplot_out_file`

  Write gnuplot output to a file.

  - Default value: `false`

    No output file specified.

  - Value: *filename*

    Example: `[gnuplot_out_file, "myplot.ps"]` This example sends PostScript output to the file `myplot.ps` when used in conjunction with the PostScript gnuplot terminal.

- Option: `gnuplot_pm3d`

  Controls the usage PM3D mode, which has advanced 3D features. PM3D is only available in gnuplot versions after 3.7. The default value for `gnuplot_pm3d` is `false`.

  Example:

  ```
  [gnuplot_pm3d, true]
  ```

- Option: `gnuplot_preamble`

  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 `gnuplot_preamble` is the empty string `""`.

  Example:

  ```
  [gnuplot_preamble, "set log y"]
  ```

- Option: `gnuplot_curve_titles`

  Controls the titles given in the plot key. The default value is `[default]`, which automatically sets the title of each curve to the function plotted. If not `[default]`, `gnuplot_curve_titles` should contain a list of strings, each of which is `"title '`*title_string*`'"`. (To disable the plot key, add `"set nokey"` to `gnuplot_preamble`.)

  Example:

```
[gnuplot_curve_titles,
["title 'My first function'", "title 'My second function'"]]
```

- Option: `gnuplot_curve_styles`

  A list of strings controlling the appearance of curves, i.e., color, width, dashing, etc., to be sent to the gnuplot plot command. The default value is `["with lines 3", "with lines 1", "with lines 2", "with lines 5", "with lines 4", "with lines 6", "with lines 7"]`, which cycles through different colors. See the gnuplot documentation for `plot` for more information.

  Example:

  ```
  [gnuplot_curve_styles, ["with lines 7", "with lines 2"]]
  ```

- Option: `gnuplot_default_term_command`

  The gnuplot command to set the terminal type for the default terminal. The default value is `set term windows "Verdana" 15` in Windows systems, and `set term x11 font "Helvetica,16"` in X11 windows systems.

  Example:

  ```
  [gnuplot_default_term_command, "set term x11"]
  ```

- Option: `gnuplot_dumb_term_command`

  The gnuplot command to set the terminal type for the dumb terminal. The default value is `"set term dumb 79 22"`, which makes the text output 79 characters by 22 characters.

  Example:

  ```
  [gnuplot_dumb_term_command, "set term dumb 132 50"]
  ```

- Option: `gnuplot_ps_term_command`

  The gnuplot command to set the terminal type for the PostScript terminal. The default value is `"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 `set term postscript` for more information.

  Example:

  All the figures in the examples for the *plot2d* function in this manual were obtained from Postscript files that were generated after setting `gnuplot_ps_term_command` as:

  ```
  [gnuplot_ps_term_command, "set size 1.3, 1.3; \
          set term postscript eps color solid lw 2.5 30"]
  ```

Examples:

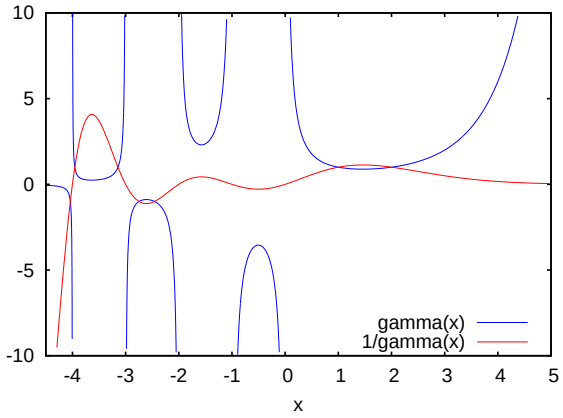
- Saves a plot of `sin(x)` to the file `sin.eps`.

  ```
  (%i1) plot2d (sin(x), [x, 0, 2*%pi], [gnuplot_term, ps],
                          [gnuplot_out_file, "sin.eps"])$
  ```

- Uses the y option to chop off singularities and the gnuplot_preamble option to put the key at the bottom of the plot instead of the top.

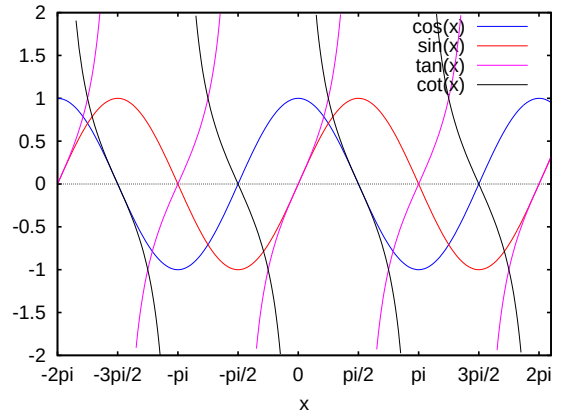
  ```
  (%i2) plot2d ([gamma(x), 1/gamma(x)], [x, -4.5, 5], [y, -10, 10],
  ```

```
                                    [gnuplot_preamble, "set key bottom"])$
```



- Uses a very complicated `gnuplot_preamble` to produce fancy x-axis labels. (Note that the `gnuplot_preamble` string must be entered without any line breaks.)

```
(%i3) my_preamble: "set xzeroaxis; set xtics ('-2pi' -6.283, \
'-3pi/2' -4.712, '-pi' -3.1415, '-pi/2' -1.5708, '0' 0, \
'pi/2' 1.5708, 'pi' 3.1415,'3pi/2' 4.712, '2pi' 6.283)"$

(%i4) plot2d([cos(x), sin(x), tan(x), cot(x)],
       [x, -2*%pi, 2.1*%pi], [y, -2, 2],
       [gnuplot_preamble, my_preamble]);
```



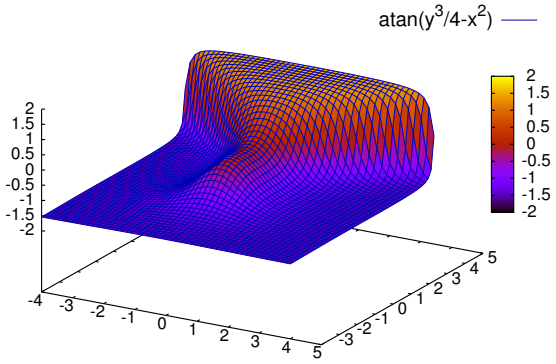
- Uses a very complicated `gnuplot_preamble` to produce fancy x-axis labels, and produces PostScript output that takes advantage of the advanced text formatting available in gnuplot. (Note that the `gnuplot_preamble` string must be entered without any line breaks.)

```
(%i5) my_preamble: "set xzeroaxis; set xtics ('-2{/Symbol p}' \
-6.283, '-3{/Symbol p}/2' -4.712, '-{/Symbol p}' -3.1415, \
'-{/Symbol p}/2' -1.5708, '0' 0,'{/Symbol p}/2' 1.5708, \
'{/Symbol p}' 3.1415,'3{/Symbol p}/2' 4.712, '2{/Symbol p}' \
```

```
6.283)"$

(%i6) plot2d ([cos(x), sin(x), tan(x)], [x, -2*%pi, 2*%pi],
    [y, -2, 2], [gnuplot_preamble, my_preamble],
    [gnuplot_term, ps], [gnuplot_out_file, "trig.eps"]);
```

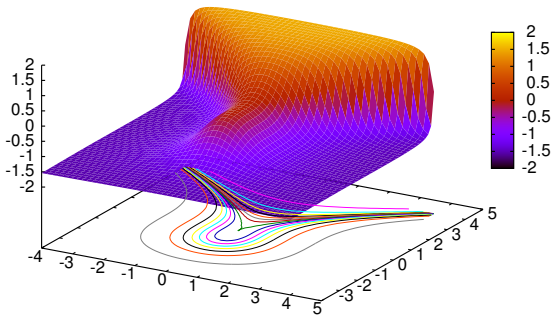
- A three-dimensional plot using the gnuplot pm3d terminal.

```
(%i7) plot3d (atan (-x^2 + y^3/4), [x, -4, 4], [y, -4, 4],
        [grid, 50, 50], [gnuplot_pm3d, true])$
```



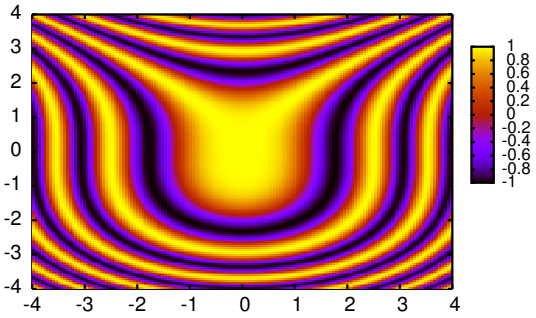
- A three-dimensional plot without a mesh and with contours projected on the bottom plane.

```
(%i8) my_preamble: "set pm3d at s;unset surface;set contour;\
set cntrparam levels 20;unset key"$
(%i9) plot3d(atan(-x^2 + y^3/4), [x, -4, 4], [y, -4, 4],
    [grid, 50, 50], [gnuplot_pm3d, true],
    [gnuplot_preamble, my_preamble])$
```



- A plot where the z-axis is represented by color only. (Note that the `gnuplot_preamble` string must be entered without any line breaks.)

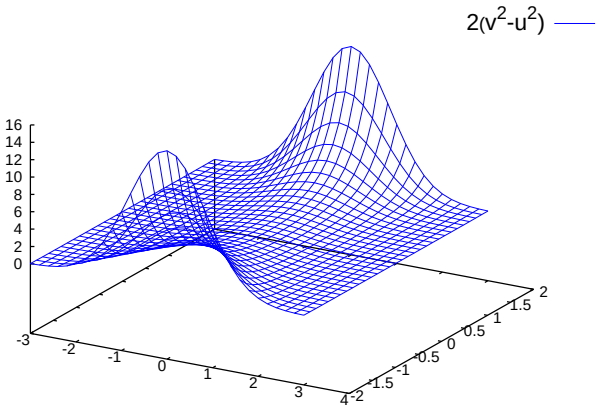
```
(%i10) plot3d (cos (-x^2 + y^3/4), [x, -4, 4], [y, -4, 4],
    [gnuplot_preamble, "set view map; unset surface"],
    [gnuplot_pm3d, true], [grid, 150, 150])$
```



plot3d ([*expr_1*, *expr_2*, *expr_3*], *x_range*, *y_range*, ..., *options*, ...) Function
plot3d (*expr*, *x_range*, *y_range*, ..., *options*, ...) Function
plot3d (*name*, *x_range*, *y_range*, ..., *options*, ...) Function
plot3d ([*expr_1*, *expr_2*, *expr_3*], *x_rge*, *y_rge*) Function
plot3d ([*name_1*, *name_2*, *name_3*], *x_range*, *y_range*, ..., *options*, ...) Function

Displays a plot of one or three expressions as functions of two variables.

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

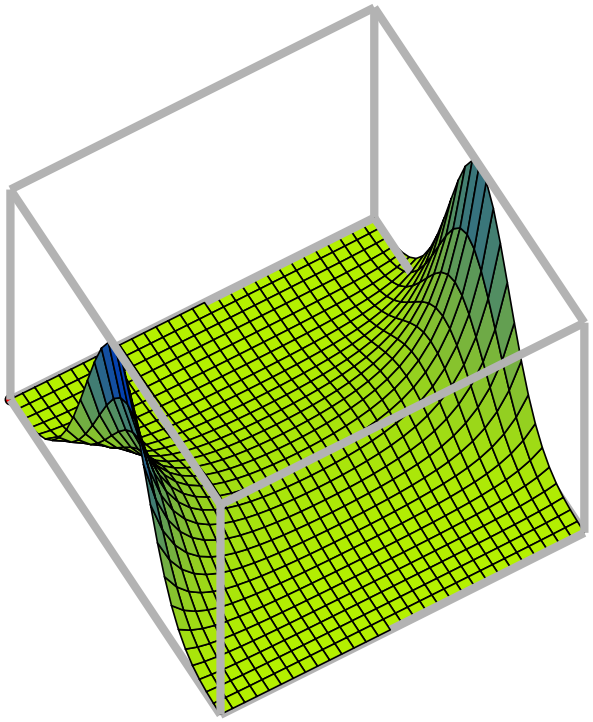


plots `z = 2^(-u^2+v^2)` with `u` and `v` varying in [-3,3] and [-2,2] respectively, and with *u* on the x axis, and `v` on the y axis.

The same graph can be plotted using openmath (if Xmaxima is installed):

```
(%i2)  plot3d (2^(-u^2 + v^2), [u, -3, 3], [v, -2, 2],
```

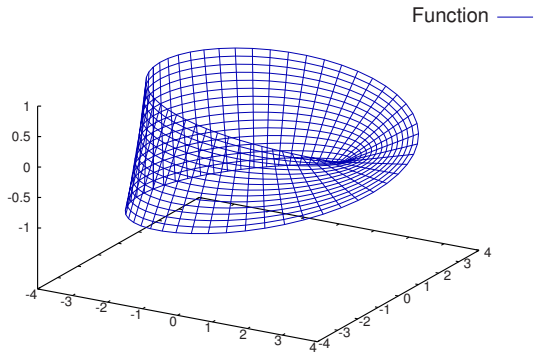
```
[plot_format, openmath]);
```



in this case the mouse can be used to rotate the plot to look at the surface from different sides.

An example of the third pattern of arguments is

```
(%i3) plot3d ([cos(x)*(3 + y*cos(x/2)), sin(x)*(3 + y*cos(x/2)),
   y*sin(x/2)], [x, -%pi, %pi], [y, -1, 1], ['grid, 50, 15]);
```

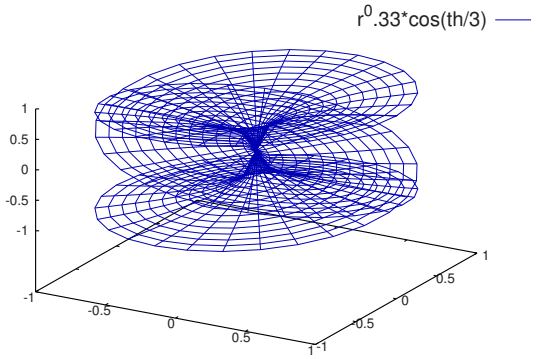


which plots a Moebius band, parametrized by the three expressions given as the first argument to `plot3d`. An additional optional argument `['grid, 50, 15]` gives the grid number of rectangles in the x direction and y direction.

The function to be plotted may be specified as the name of a Maxima or Lisp function or operator, a Maxima lambda expression, or a general Maxima expression. In the form `plot3d (`*f*`, ...)` where *f* is the name of a function or a lambda expression, the function must be a function of two arguments. In the form `plot3d ([`*f_1*`, `*f_2*`, `*f_3*`], ...)` where *f_1*, *f_2*, and *f_3* are names of functions or lambda expressions, each function must be a function of three arguments.

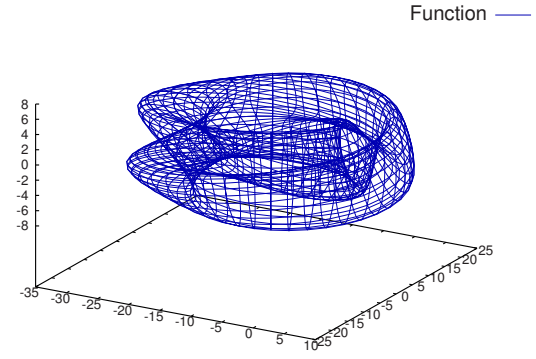
This example shows a plot of the real part of `z^1/3`.

```
(%i4) plot3d (r^.33*cos(th/3), [r, 0, 1], [th, 0, 6*%pi],
      ['grid, 12, 80], ['transform_xy, polar_to_xy]);
```



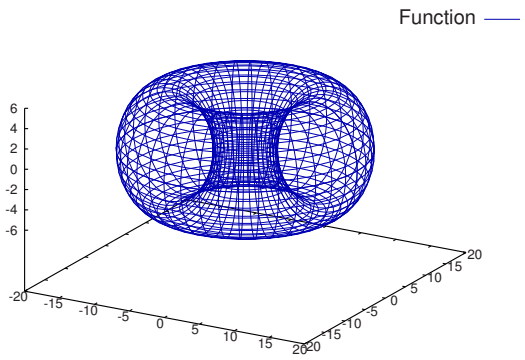
Other examples are the Klein bottle:

```
(%i5) expr_1: 5*cos(x)*(cos(x/2)*cos(y) + sin(x/2)*sin(2*y)
      + 3.0) - 10.0$
(%i6) expr_2: -5*sin(x)*(cos(x/2)*cos(y) + sin(x/2)*sin(2*y)
      + 3.0)$
(%i7) expr_3: 5*(-sin(x/2)*cos(y) + cos(x/2)*sin(2*y))$

(%i8) plot3d ([expr_1, expr_2, expr_3], [x, -%pi, %pi],
      [y, -%pi, %pi], ['grid, 40, 40]);
```



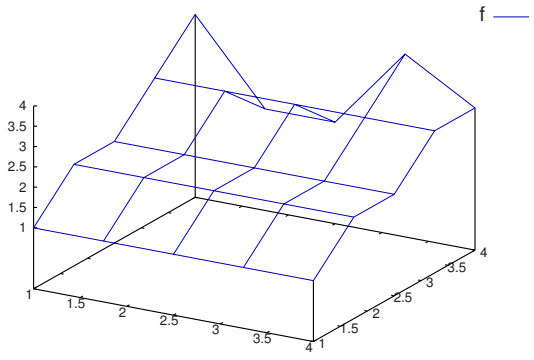
and a torus:

```
(%i9) expr_1: cos(y)*(10.0+6*cos(x))$
(%i10) expr_2: sin(y)*(10.0+6*cos(x))$
(%i11) expr_3: -6*sin(x)$
(%i12) plot3d ([expr_1, expr_2, expr_3], [x, 0, 2*%pi],
       [y, 0, 2*%pi], ['grid, 40, 40]);
```



Sometimes it is necessary to define a function to plot the expression. All the arguments to plot3d are evaluated before being passed to plot3d, and so trying to make an expression which does just what is needed may be difficult, and it is just easier to make a function.

```
(%i13) M: matrix([1, 2, 3, 4], [1, 2, 3, 2], [1, 2, 3, 4],
       [1, 2, 3, 3])$
(%i14) f(x, y) := float (M [?round(x), ?round(y)])$
(%i15) plot3d (f, [x, 1, 4], [y, 1, 4], ['grid, 4, 4])$
```



See `plot_options` for more examples.

make_transform (*vars*, *fx*, *fy*, *fz*) Function

Returns a function suitable for the transform function in plot3d. Use with the plot option `transform_xy`.

```
make_transform ([r, th, z], r*cos(th), r*sin(th), z)$
```

is a transformation to polar coordinates.

set_plot_option (*option*) Function

Assigns one of the global variables for plotting. *option* is specified as a list of two or more elements, in which the first element is one of the keywords on the `plot_options` list.

`set_plot_option` evaluates its argument and returns the complete list `plot_options` (after modifying one of its elements).

See also `plot_options`, `plot2d`, and `plot3d`.

Examples:

Modify the `grid` and `x` values. When a `plot_options` keyword has an assigned value, quote it to prevent evaluation.

```
(%i1) set_plot_option ([grid, 30, 40]);
(%o1) [[x, - 1.755559702014E+305, 1.755559702014E+305],
[y, - 1.755559702014E+305, 1.755559702014E+305], [t, - 3, 3],
[grid, 30, 40], [transform_xy, false], [run_viewer, true],
[plot_format, gnuplot], [gnuplot_term, default],
[gnuplot_out_file, false], [nticks, 10], [adapt_depth, 10],
[gnuplot_pm3d, false], [gnuplot_preamble, ],
[gnuplot_curve_titles, [default]],
[gnuplot_curve_styles, [with lines 3, with lines 1,
with lines 2, with lines 5, with lines 4, with lines 6,
with lines 7]], [gnuplot_default_term_command, ],
[gnuplot_dumb_term_command, set term dumb 79 22],
[gnuplot_ps_term_command, set size 1.5, 1.5;set term postscript #
eps enhanced color solid 24]]
(%i2) x: 42;
(%o2)                          42
(%i3) set_plot_option ([’x, -100, 100]);
(%o3) [[x, - 100.0, 100.0], [y, - 1.755559702014E+305,
1.755559702014E+305], [t, - 3, 3], [grid, 30, 40],
[transform_xy, false], [run_viewer, true],
[plot_format, gnuplot], [gnuplot_term, default],
[gnuplot_out_file, false], [nticks, 10], [adapt_depth, 10],
[gnuplot_pm3d, false], [gnuplot_preamble, ],
[gnuplot_curve_titles, [default]],
[gnuplot_curve_styles, [with lines 3, with lines 1,
with lines 2, with lines 5, with lines 4, with lines 6,
with lines 7]], [gnuplot_default_term_command, ],
[gnuplot_dumb_term_command, set term dumb 79 22],
[gnuplot_ps_term_command, set size 1.5, 1.5;set term postscript #
eps enhanced color solid 24]]
```

8.1.1 Functions for working with the gnuplot_pipes format

gnuplot_start () Function

Opens the pipe to gnuplot used for plotting with the `gnuplot_pipes` format. Is not necessary to manually open the pipe before plotting.

gnuplot_close () Function

Closes the pipe to gnuplot which is used with the `gnuplot_pipes` format.

gnuplot_restart () Function

Closes the pipe to gnuplot which is used with the `gnuplot_pipes` format and opens a new pipe.

gnuplot_replot () Function
gnuplot_replot (*s*) Function

Updates the gnuplot window. If `gnuplot_replot` is called with a gnuplot command in a string *s*, then `s` is sent to gnuplot before reploting the window.

gnuplot_reset () Function

Resets the state of gnuplot used with the `gnuplot_pipes` format. To update the gnuplot window call `gnuplot_replot` after `gnuplot_reset`.

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`.

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:

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

_ System variable

_ is the most recent input expression (e.g., %i1, %i2, %i3, ...).

_ is assigned the input expression before the input is simplified or evaluated. However, the value of _ is simplified (but not evaluated) when it is displayed.

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

See also __ and %.

Examples:

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

```
(%i8) _;
(%o8)                          b + a
(%i9) a + b;
(%o9)                           42
(%i10) ev (_);
(%o10)                          42
```

% System variable

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

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

See also _, %%, and %th.

%% System variable

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

%edispflag Option variable

Default value: false

When %edispflag is true, Maxima displays %e to a negative exponent as a quotient. For example, %e^-x is displayed as 1/%e^x.

%th (*i*) Function

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

`%th` is 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 7 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.

absboxchar Option variable

Default value: `!`

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

file_output_append Option variable

Default value: `false`

`file_output_append` governs whether file output functions append or truncate their output file. When `file_output_append` is `true`, such functions append to their output file. Otherwise, the output file is truncated.

`save`, `stringout`, and `with_stdout` respect `file_output_append`. Other functions which write output files do not respect `file_output_append`. In particular, plotting and translation functions always truncate their output file, and `tex` and `appendfile` always append.

appendfile (*filename*) Function

Appends a console transcript to *filename*. `appendfile` is the same as `writefile`, except that the transcript file, if it exists, is always appended.

`closefile` closes the transcript file opened by `appendfile` or `writefile`.

batch (*filename*) Function

Reads Maxima expressions from *filename* and evaluates them. `batch` searches for *filename* in the list `file_search_maxima`. See `file_search`.

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` has no return value.

See also `load`, `batchload`, and `demo`.

batchload (*filename*) Function

Reads Maxima expressions from *filename* and evaluates them, without displaying the input or output expressions and without assigning labels to output expressions. Printed output (such as produced by `print` or `describe`) is displayed, however.

The special variable `%` and the function `%th` refer to previous results from the interactive interpreter, not results within the file. The file cannot include `:lisp` constructs.

`batchload` returns the path of *filename*, as a string. `batchload` evaluates its argument.

See also `batch` and `load`.

closefile () Function

Closes the transcript file opened by `writefile` or `appendfile`.

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.

```
(%i1) y: 7$
(%i2) z: 88$
(%i3) concat (y, z/2);
(%o3)                          744
(%i4) concat ('y, z/2);
(%o4)                          y44
```

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

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

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

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

dispcon (*tensor_1, tensor_2, ...*) Function

dispcon (*all*) 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`.

```
(%i1) display(B[1,2]);
                                        2
                              B     = X - X
                               1, 2
(%o1)                                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.

display_format_internal Option variable

Default value: `false`

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

Examples:

```
User     part       inpart
a-b;      a - b     a + (- 1) b

           a            - 1
a/b;       -         a b
           b
                       1/2
sqrt(x);   sqrt(x)    x

          4 X        4
X*4/3;    ---        - X
           3         3
```

dispterms (*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 `P1`, `P2`, ... are very large expressions then the display program may run out of storage space in trying to display `P1 + P2 + ...` all at once. However, `dispterms (P1 + P2 + ...)` displays `P1`, then below it `P2`, etc. When not using `dispterms`, 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`).

error_size Option variable

Default value: 10

`error_size` modifies error messages according to the size of expressions which appear in them. If the size of an expression (as determined by the Lisp function `ERROR-SIZE`)

is greater than `error_size`, the expression is replaced in the message by a symbol, and the symbol is assigned the expression. The symbols are taken from the list `error_syms`.

Otherwise, the expression is smaller than `error_size`, and the expression is displayed in the message.

See also `error` and `error_syms`.

Example:

The size of `U`, as determined by `ERROR-SIZE`, is 24.

```
(%i1) U: (C^D^E + B + A)/(cos(X-1) + 1)$

(%i2) error_size: 20$

(%i3) error ("Example expression is", U);

Example expression is errexp1
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i4) errexp1;
                             E
                            D
                           C   + B + A
(%o4)                     --------------
                          cos(X - 1) + 1
(%i5) error_size: 30$

(%i6) error ("Example expression is", U);

                           E
                          D
                         C   + B + A
Example expression is --------------
                      cos(X - 1) + 1
 -- an error.  Quitting.  To debug this try debugmode(true);
```

error_syms Option variable

Default value: `[errexp1, errexp2, errexp3]`

In error messages, expressions larger than `error_size` are replaced by symbols, and the symbols are set to the expressions. The symbols are taken from the list `error_syms`. The first too-large expression is replaced by `error_syms[1]`, the second by `error_syms[2]`, and so on.

If there are more too-large expressions than there are elements of `error_syms`, symbols are constructed automatically, with the n-th symbol equivalent to `concat ('errexp, n)`.

See also `error` and `error_size`.

expt (a, b) Function

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

`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 (*filename, pathlist*) 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,

```
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 Option variable
file_search_lisp Option variable
file_search_demo Option variable

These variables specify lists of directories to be searched by `load`, `demo`, and some other Maxima functions. The default values of these variables name various directories in the Maxima installation.

The user can modify these variables, either to replace the default values or to append additional directories. For example,

```
file_search_maxima: ["/usr/local/foo/###.mac",
    "/usr/local/bar/###.mac"]$
```

replaces the default value of `file_search_maxima`, while

```
file_search_maxima: append (file_search_maxima,
    ["/usr/local/foo/###.mac", "/usr/local/bar/###.mac"])$
```

appends two additional directories. It may be convenient to put such an expression in the file `maxima-init.mac` so that the file search path is assigned automatically when Maxima starts.

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*) Function

Returns a guess about the content of *filename*, based on the filename extension. *filename* need not refer to an actual file; no attempt is made to open the file and inspect the content.

The return value is a symbol, either `object`, `lisp`, or `maxima`. If the extension starts with `m` or `d`, `file_type` returns `maxima`. If the extension starts with `l`, `file_type` returns `lisp`. If none of the above, `file_type` returns `object`.

grind (*expr*) Function

grind Option variable

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

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

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

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]);
                              [ aa  17 ]
(%o5)                         [        ]
                              [ 29  bb ]
(%i6) grind (%);
matrix([aa,17],[29,bb])$
(%o6)                            done
(%i7) set (aa, 17, 29, bb);
(%o7)                     {17, 29, aa, bb}
(%i8) grind (%);
{17,29,aa,bb}$
(%o8)                            done
(%i9) exp (aa / (bb + 17)^29);
                                  aa
                               -----------
                                        29
                              (bb + 17)
(%o9)                        %e
(%i10) grind (%);
%e^(aa/(bb+17)^29)$
(%o10)                           done
(%i11) expr: expand ((aa + bb)^10);
         10           9        2   8         3   7         4   6
(%o11) bb   + 10 aa bb  + 45 aa  bb  + 120 aa  bb  + 210 aa  bb
         5   5         6   4         7   3        8   2
 + 252 aa  bb  + 210 aa  bb  + 120 aa  bb  + 45 aa  bb
        9        10
 + 10 aa  bb + aa
(%i12) grind (expr);
bb^10+10*aa*bb^9+45*aa^2*bb^8+120*aa^3*bb^7+210*aa^4*bb^6
     +252*aa^5*bb^5+210*aa^6*bb^4+120*aa^7*bb^3+45*aa^8*bb^2
     +10*aa^9*bb+aa^10$
(%o12)                           done
(%i13) string (expr);
(%o13) bb^10+10*aa*bb^9+45*aa^2*bb^8+120*aa^3*bb^7+210*aa^4*bb^6\
+252*aa^5*bb^5+210*aa^6*bb^4+120*aa^7*bb^3+45*aa^8*bb^2+10*aa^9*\
bb+aa^10
(%i14) cholesky (A):= block ([n : length (A), L : copymatrix (A),
  p : makelist (0, i, 1, length (A))], for i thru n do
```

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

ibase Option variable

Default value: 10

Integers entered into Maxima are interpreted with respect to the base `ibase`.

`ibase` may be assigned any integer between 2 and 35 (decimal), inclusive. When `ibase` is greater than 10, the numerals comprise the decimal numerals 0 through 9 plus capital letters of the alphabet A, B, C, ..., as needed. The numerals for base 35, the largest acceptable base, comprise 0 through 9 and A through Y.

See also `obase`.

inchar Option variable

Default value: `%i`

`inchar` is the prefix of the labels of expressions entered by the user. Maxima automatically constructs a label for each input expression by concatenating `inchar` and `linenum`. `inchar` may be assigned any string or symbol, not necessarily a single character.

```
(%i1) inchar: "input";
(%o1)                                input
(input1) expand ((a+b)^3);
                             3        2      2      3
(%o1)                       b  + 3 a b  + 3 a  b + a
(input2)
```

See also `labels`.

ldisp (*expr_1, ..., expr_n*) Function

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

See also `disp`.

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

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

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

ldisplay (*expr_1, ..., expr_n*) Function

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

See also `display`.

```
(%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
```

linechar Option variable

Default value: `%t`

`linechar` is the prefix of the labels of intermediate expressions generated by Maxima. Maxima constructs a label for each intermediate expression (if displayed) by concatenating `linechar` and `linenum`. `linechar` may be assigned any string or symbol, not necessarily a single character.

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

linel Option variable

Default value: 79

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

lispdisp Option variable

Default value: `false`

When `lispdisp` is `true`, Lisp symbols are displayed with a leading question mark `?`. Otherwise, Lisp symbols are displayed with no leading mark.

Examples:

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

load (*filename*) Function

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

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

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

See also `loadfile`, `batch`, `batchload`, and `demo`. `loadfile` processes Lisp files; `batch`, `batchload`, and `demo` process Maxima files.

See `file_search` for more detail about the file search mechanism.

`load` evaluates its argument.

loadfile (*filename*) Function

Evaluates Lisp expressions in *filename*. `loadfile` does not invoke `file_search`, so `filename` must include the file extension and as much of the path as needed to find the file.

`loadfile` can process files created by `save`, `translate_file`, and `compile_file`. The user may find it more convenient to use `load` instead of `loadfile`.

loadprint Option variable

Default value: `true`

`loadprint` tells whether to print a message when a file is loaded.

- When `loadprint` is `true`, always print a message.
- When `loadprint` is `'loadfile`, print a message only if a file is loaded by the function `loadfile`.
- When `loadprint` is `'autoload`, print a message only if a file is automatically loaded. See `setup_autoload`.
- When `loadprint` is `false`, never print a message.

obase Option variable

Default value: 10

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

`obase` may be assigned any integer between 2 and 35 (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. The numerals for base 35, the largest acceptable base, comprise 0 through 9, and A through Y.

See also `ibase`.

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.

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

```
                                  3        2      2      3
(output2)                        b  + 3 a b  + 3 a  b + a
(%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.

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;
(%o5)                          65536/343
(%i6) (a+b)/8;
(%o6)                         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*) Function
tcl_output (*list*, *i0*) Function
tcl_output ([*list_1*, ..., *list_n*], *i*) Function

Prints elements of a list enclosed by curly braces `{ }`, suitable as part of a program in the Tcl/Tk language.

`tcl_output (`*list*`, `*i0*`, `*skip*`)` prints *list*, beginning with element *i0* and printing elements *i0* + *skip*, *i0* + 2 *skip*, etc.

`tcl_output (`*list*`, `*i0*`)` is equivalent to `tcl_output (`*list*`, `*i0*`, 2)`.

`tcl_output ([`*list_1*`, ..., `*list_n*`], `*i*`)` prints the *i*'th elements of *list_1*, ..., *list_n*.

Examples:

```
(%i1) tcl_output ([1, 2, 3, 4, 5, 6], 1, 3)$

 {1.000000000     4.000000000
 }
(%i2) tcl_output ([1, 2, 3, 4, 5, 6], 2, 3)$

 {2.000000000     5.000000000
 }
(%i3) tcl_output ([3/7, 5/9, 11/13, 13/17], 1)$

 {((RAT SIMP) 3 7) ((RAT SIMP) 11 13)
```

```
 }
(%i4) tcl_output ([x1, y1, x2, y2, x3, y3], 2)$

 {$Y1 $Y2 $Y3
 }
(%i5) tcl_output ([[1, 2, 3], [11, 22, 33]], 1)$

 {SIMP 1.000000000     11.00000000
 }
```

read (*expr_1*, ..., *expr_n*) Function

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

See also `readonly`.

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

readonly (*expr_1*, ..., *expr_n*) Function

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

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

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 `Sum(`*n*`)` where *n* is the number of operands of the sum.
- Products are replaced by `Product(`*n*`)` where *n* is the number of operands of the product.
- Exponentials are replaced by `Expt`.

- Quotients are replaced by `Quotient`.
- Unary negation is replaced by `Negterm`.

When *depth* is greater than or equal to the maximum depth of *expr*, `reveal (`*expr*`,` *depth*`)` returns *expr* unmodified.

`reveal` evaluates its arguments. `reveal` returns the summarized expression.

Example:

```
(%i1) e: expand ((a - b)^2)/expand ((exp(a) + exp(b))^2);
                           2            2
                          b  - 2 a b + a
(%o1)                -------------------------
                         b + a     2 b     2 a
                      2 %e      + %e    + %e
(%i2) reveal (e, 1);
(%o2)                          Quotient
(%i3) reveal (e, 2);
                                Sum(3)
(%o3)                           ------
                                Sum(3)
(%i4) reveal (e, 3);
                     Expt + Negterm + Expt
(%o4)                ----------------------
                     Product(2) + Expt + Expt
(%i5) reveal (e, 4);
                        2                  2
                       b  - Product(3) + a
(%o5)          ----------------------------------
                          Product(2)     Product(2)
                2 Expt + %e           + %e
(%i6) reveal (e, 5);
                           2            2
                          b  - 2 a b + a
(%o6)                -------------------------
                         Sum(2)     2 b     2 a
                      2 %e       + %e    + %e
(%i7) reveal (e, 6);
                           2            2
                          b  - 2 a b + a
(%o7)                -------------------------
                         b + a     2 b     2 a
                      2 %e      + %e    + %e
```

rmxchar Option variable

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 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*, ...) Function
stringout (*filename*, [*m*, *n*]) Function
stringout (*filename*, *input*) Function
stringout (*filename*, *functions*) Function
stringout (*filename*, *values*) Function

`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.

tex (*expr*) Function
tex (*expr, destination*) Function
tex (*expr, false*) Function
tex (*label*) Function
tex (*label, destination*) Function
tex (*label, false*) Function

Prints a representation of an expression suitable for the TeX document preparation system. The result is a fragment of a document, which can be copied into a larger document but not processed by itself.

`tex (`*expr*`)` prints a TeX representation of *expr* on the console.

`tex (`*label*`)` prints a TeX representation of the expression named by *label* and assigns it an equation label (to be displayed to the left of the expression). The TeX equation label is the same as the Maxima label.

destination may be an output stream or file name. When *destination* is a file name, `tex` appends its output to the file. The functions `openw` and `opena` create output streams.

`tex (`*expr*`, false)` and `tex (`*label*`, false)` return their TeX output as a string.

`tex` evaluates its first argument after testing it to see if it is a label. Quote-quote `''` forces evaluation of the argument, thereby defeating the test and preventing the label.

See also `texput`.

Examples:

```
(%i1) integrate (1/(1+x^3), x);
                                        2 x - 1
                     2             atan(-------)
                  log(x  - x + 1)        sqrt(3)    log(x + 1)
(%o1)          - --------------- + ------------- + ----------
                        6             sqrt(3)          3
(%i2) tex (%o1);
$$-{{\log \left(x^2-x+1\right)}\over{6}}+{{\arctan \left({{2\,x-1
 }\over{\sqrt{3}}}\right)}\over{\sqrt{3}}}+{{\log \left(x+1\right)
 }\over{3}}\leqno{\tt (\%o1)}$$
(%o2)                          (\%o1)
(%i3) tex (integrate (sin(x), x));
$$-\cos x$$
(%o3)                           false
(%i4) tex (%o1, "foo.tex");
(%o4)                          (\%o1)
```

`tex` (*expr*, `false`) returns its TeX output as a string.

```
(%i1) S : tex (x * y * z, false);
(%o1) $$x\,y\,z$$
(%i2) S;
(%o2) $$x\,y\,z$$
```

texput (*a*, *s*) Function
texput (*a*, *s*, *operator_type*) Function
texput (*a*, [*s_1*, *s_2*], *matchfix*) Function
texput (*a*, [*s_1*, *s_2*, *s_3*], *matchfix*) Function

Assign the TeX output for the atom *a*, which can be a symbol or the name of an operator.

`texput` (*a*, *s*) causes the `tex` function to interpolate the string *s* into the TeX output in place of *a*.

`texput` (*a*, *s*, *operator_type*), where *operator_type* is `prefix`, `infix`, `postfix`, `nary`, or `nofix`, causes the `tex` function to interpolate *s* into the TeX output in place of *a*, and to place the interpolated text in the appropriate position.

`texput` (*a*, [*s_1*, *s_2*], `matchfix`) causes the `tex` function to interpolate *s_1* and *s_2* into the TeX output on either side of the arguments of *a*. The arguments (if more than one) are separated by commas.

`texput` (*a*, [*s_1*, *s_2*, *s_3*], `matchfix`) causes the `tex` function to interpolate *s_1* and *s_2* into the TeX output on either side of the arguments of *a*, with *s_3* separating the arguments.

Examples:

Assign TeX output for a variable.

```
(%i1) texput (me,"\\mu_e");
(%o1)                          \mu_e
(%i2) tex (me);
$$\mu_e$$
(%o2)                          false
```

Assign TeX output for an ordinary function (not an operator).

```
(%i1) texput (lcm, "\\mathrm{lcm}");
(%o1)                      \mathrm{lcm}
(%i2) tex (lcm (a, b));
$$\mathrm{lcm}\left(a , b\right)$$
(%o2)                           false
```

Assign TeX output for a prefix operator.

```
(%i1) prefix ("grad");
(%o1)                           grad
(%i2) texput ("grad", " \\nabla ", prefix);
(%o2)                          \nabla
(%i3) tex (grad f);
$$ \nabla f$$
(%o3)                           false
```

Assign TeX output for an infix operator.

```
(%i1) infix ("~");
(%o1)                             ~
(%i2) texput ("~", " \\times ", infix);
(%o2)                          \times
(%i3) tex (a ~ b);
$$a \times b$$
(%o3)                           false
```

Assign TeX output for a postfix operator.

```
(%i1) postfix ("##");
(%o1)                            ##
(%i2) texput ("##", "!!", postfix);
(%o2)                            !!
(%i3) tex (x ##);
$$x!!$$
(%o3)                           false
```

Assign TeX output for a nary operator.

```
(%i1) nary ("@@");
(%o1)                            @@
(%i2) texput ("@@", " \\circ ", nary);
(%o2)                           \circ
(%i3) tex (a @@ b @@ c @@ d);
$$a \circ b \circ c \circ d$$
(%o3)                           false
```

Assign TeX output for a nofix operator.

```
(%i1) nofix ("foo");
(%o1)                            foo
(%i2) texput ("foo", "\\mathsc{foo}", nofix);
(%o2)                      \mathsc{foo}
(%i3) tex (foo);
$$\mathsc{foo}$$
(%o3)                           false
```

Assign TeX output for a matchfix operator.

```
(%i1) matchfix ("<<", ">>");
(%o1)                           <<
(%i2) texput ("<<", [" \\langle ", " \\rangle "], matchfix);
(%o2)                 [ \langle ,  \rangle ]
(%i3) tex (<<a>>);
$$ \langle a \rangle $$
(%o3)                           false
(%i4) tex (<<a, b>>);
$$ \langle a , b \rangle $$
(%o4)                           false
(%i5) texput ("<<", [" \\langle ", " \\rangle ", " \\, | \\,"],
      matchfix);
(%o5)           [ \langle ,  \rangle ,  \, | \,]
(%i6) tex (<<a>>);
$$ \langle a \rangle $$
(%o6)                           false
(%i7) tex (<<a, b>>);
$$ \langle a \, | \,b \rangle $$
(%o7)                           false
```

get_tex_environment (*op*) Function
set_tex_environment (*op, before, after*) Function

Customize the TeX environment output by `tex`. As maintained by these functions, the TeX environment comprises two strings: one is printed before any other TeX output, and the other is printed after.

Only the TeX environment of the top-level operator in an expression is output; TeX environments associated with other operators are ignored.

`get_tex_environment` returns the TeX enviroment which is applied to the operator *op*; returns the default if no other environment has been assigned.

`set_tex_environment` assigns the TeX environment for the operator *op*.

Examples:

```
(%i1) get_tex_environment (":=");
(%o1) [
\begin{verbatim}
, ;
\end{verbatim}
]
(%i2) tex (f (x) := 1 - x);

\begin{verbatim}
f(x):=1-x;
\end{verbatim}

(%o2)                           false
(%i3) set_tex_environment (":=", "$$", "$$");
(%o3)                         [$$, $$]
(%i4) tex (f (x) := 1 - x);
```

```
$$f(x):=1-x$$
(%o4)                          false
```

get_tex_environment_default () Function

set_tex_environment_default (*before, after*) Function

Customize the TeX environment output by `tex`. As maintained by these functions, the TeX environment comprises two strings: one is printed before any other TeX output, and the other is printed after.

`get_tex_environment_default` returns the TeX environment which is applied to expressions for which the top-level operator has no specific TeX environment (as assigned by `set_tex_environment`).

`set_tex_environment_default` assigns the default TeX environment.

Examples:

```
(%i1) get_tex_environment_default ();
(%o1)                         [$$, $$]
(%i2) tex (f(x) + g(x));
$$g\left(x\right)+f\left(x\right)$$
(%o2)                          false
(%i3) set_tex_environment_default ("\\begin{equation}
", "
\\end{equation}");
(%o3) [\begin{equation}
,
\end{equation}]
(%i4) tex (f(x) + g(x));
\begin{equation}
g\left(x\right)+f\left(x\right)
\end{equation}
(%o4)                          false
```

system (*command*) Function

Executes *command* as a separate process. The command is passed to the default shell for execution. `system` is not supported by all operating systems, but generally exists in Unix and Unix-like environments.

Supposing `_hist.out` is a list of frequencies which you wish to plot as a bar graph using `xgraph`.

```
(%i1) (with_stdout("_hist.out",
           for i:1 thru length(hist) do (
             print(i,hist[i]))),
       system("xgraph -bar -brw .7 -nl < _hist.out"));
```

In order to make the plot be done in the background (returning control to Maxima) and remove the temporary file after it is done do:

```
system("(xgraph -bar -brw .7 -nl < _hist.out;  rm -f _hist.out)&")
```

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, ...*) Function
with_stdout (*s, expr_1, expr_2, expr_3, ...*) Function

Evaluates *expr_1*, *expr_2*, *expr_3*, ... and writes any output thus generated to a file *f* or output stream *s*. The evaluated expressions are not written to the output. Output may be generated by `print`, `display`, `grind`, among other functions.

The global flag `file_output_append` governs whether `with_stdout` appends or truncates the output file *f*. When `file_output_append` is `true`, `with_stdout` appends to the output file. Otherwise, `with_stdout` truncates the output file. In either case, `with_stdout` creates the file if it does not yet exist.

`with_stdout` returns the value of its final argument.

See also `writefile`.

```
(%i1) with_stdout ("tmp.out", for i:5 thru 10 do
      print (i, "! yields", i!))$
(%i2) printfile ("tmp.out")$
5 ! yields 120
6 ! yields 720
7 ! yields 5040
8 ! yields 40320
9 ! yields 362880
10 ! yields 3628800
```

writefile (*filename*) Function

Begins writing a transcript of the Maxima session to *filename*. All interaction between the user and Maxima is then recorded in this file, just as it appears on the console.

As the transcript is printed in the console output format, it cannot be reloaded into Maxima. To make a file containing expressions which can be reloaded, see `save` and `stringout`. `save` stores expressions in Lisp form, while `stringout` stores expressions in Maxima form.

The effect of executing `writefile` when *filename* already exists depends on the underlying Lisp implementation; the transcript file may be clobbered, or the file may be appended. `appendfile` always appends to the transcript file.

It may be convenient to execute `playback` after `writefile` to save the display of previous interactions. As `playback` displays only the input and output variables (`%i1`, `%o1`, etc.), any output generated by a print statement in a function (as opposed to a return value) is not displayed by `playback`.

`closefile` closes the transcript file opened by `writefile` or `appendfile`.

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 `1.000000000000000B0`. Otherwise, this is displayed as `1.0B0`.

cbffac (*z*, *fpprec*) Function

Complex bigfloat factorial.

`load ("bffac")` loads this function.

float (*expr*) Function

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

float2bf Option variable

Default value: `false`

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

floatnump (*expr*) Function

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

fpprec Option variable

Default value: 16

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

See also `bfloat` and `fpprintprec`.

fpprintprec Option variable

Default value: 0

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

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

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

`fpprintprec` cannot be 1.

11 Contexts

11.1 Functions and Variables for Contexts

activate (*context_1, ..., context_n*) Function

Activates the contexts *context_1*, ..., *context_n*. The facts in these contexts are then available to make deductions and retrieve information. The facts in these contexts are not listed by `facts ()`.

The variable `activecontexts` is the list of contexts which are active by way of the `activate` function.

activecontexts System variable

Default value: `[]`

`activecontexts` is a list of the contexts which are active by way of the `activate` function, as opposed to being active because they are subcontexts of the current context.

assume (*pred_1, ..., pred_n*) Function

Adds predicates *pred_1*, ..., *pred_n* to the current context. If a predicate is inconsistent or redundant with the predicates in the current context, it is not added to the context. The context accumulates predicates from each call to `assume`.

`assume` returns a list whose elements are the predicates added to the context or the atoms `redundant` or `inconsistent` where applicable.

The predicates *pred_1*, ..., *pred_n* can only be expressions with the relational operators `< <= equal notequal >=` and `>`. Predicates cannot be literal equality `=` or literal inequality `#` expressions, nor can they be predicate functions such as `integerp`.

Compound predicates of the form *pred_1* `and ... and` *pred_n* are recognized, but not *pred_1* `or ... or` *pred_n*. `not` *pred_k* is recognized if *pred_k* is a relational predicate. Expressions of the form `not (`*pred_1* `and` *pred_2*`)` and `not (`*pred_1* `or` *pred_2*`)` are not recognized.

Maxima's deduction mechanism is not very strong; there are many obvious consequences which cannot be determined by `is`. This is a known weakness.

`assume` evaluates its arguments.

See also `is`, `facts`, `forget`, `context`, and `declare`.

Examples:

```
(%i1) assume (xx > 0, yy < -1, zz >= 0);
(%o1)              [xx > 0, yy < - 1, zz >= 0]
(%i2) assume (aa < bb and bb < cc);
(%o2)                   [bb > aa, cc > bb]
(%i3) facts ();
(%o3)     [xx > 0, - 1 > yy, zz >= 0, bb > aa, cc > bb]
(%i4) is (xx > yy);
(%o4)                          true
```

```
(%i5) is (yy < -yy);
(%o5)                          true
(%i6) is (sinh (bb - aa) > 0);
(%o6)                          true
(%i7) forget (bb > aa);
(%o7)                        [bb > aa]
(%i8) prederror : false;
(%o8)                          false
(%i9) is (sinh (bb - aa) > 0);
(%o9)                         unknown
(%i10) is (bb^2 < cc^2);
(%o10)                        unknown
```

assumescalar Option variable

Default value: `true`

`assumescalar` helps govern whether expressions `expr` for which `nonscalarp (expr)` is `false` are assumed to behave like scalars for certain transformations.

Let `expr` represent any expression other than a list or a matrix, and let `[1, 2, 3]` represent any list or matrix. Then `expr . [1, 2, 3]` yields `[expr, 2 expr, 3 expr]` if `assumescalar` is `true`, or `scalarp (expr)` is `true`, or `constantp (expr)` is `true`.

If `assumescalar` is `true`, such expressions will behave like scalars only for commutative operators, but not for noncommutative multiplication `..`

When `assumescalar` is `false`, such expressions will behave like non-scalars.

When `assumescalar` is `all`, such expressions will behave like scalars for all the operators listed above.

assume_pos Option variable

Default value: `false`

When `assume_pos` is `true` and the sign of a parameter x cannot be determined from the current context or other considerations, `sign` and `asksign (`x`)` return `true`. This may forestall some automatically-generated `asksign` queries, such as may arise from `integrate` or other computations.

By default, a parameter is x such that `symbolp (`x`)` or `subvarp (`x`)`. The class of expressions considered parameters can be modified to some extent via the variable `assume_pos_pred`.

`sign` and `asksign` attempt to deduce the sign of expressions from the sign of operands within the expression. For example, if `a` and `b` are both positive, then `a + b` is also positive.

However, there is no way to bypass all `asksign` queries. In particular, when the `asksign` argument is a difference x `-` y or a logarithm `log(`x`)`, `asksign` always requests an input from the user, even when `assume_pos` is `true` and `assume_pos_pred` is a function which returns `true` for all arguments.

assume_pos_pred Option variable

Default value: `false`

When `assume_pos_pred` is assigned the name of a function or a lambda expression of one argument *x*, that function is called to determine whether *x* is considered a parameter for the purpose of `assume_pos`. `assume_pos_pred` is ignored when `assume_pos` is `false`.

The `assume_pos_pred` function is called by `sign` and `asksign` with an argument *x* which is either an atom, a subscripted variable, or a function call expression. If the `assume_pos_pred` function returns `true`, *x* is considered a parameter for the purpose of `assume_pos`.

By default, a parameter is *x* such that `symbolp (`*x*`)` or `subvarp (`*x*`)`.

See also `assume` and `assume_pos`.

Examples:

```
(%i1) assume_pos: true$
(%i2) assume_pos_pred: symbolp$
(%i3) sign (a);
(%o3)                          pos
(%i4) sign (a[1]);
(%o4)                          pnz
(%i5) assume_pos_pred: lambda ([x], display (x), true)$
(%i6) asksign (a);
                               x = a

(%o6)                          pos
(%i7) asksign (a[1]);
                              x = a
                                  1

(%o7)                          pos
(%i8) asksign (foo (a));
                            x = foo(a)

(%o8)                          pos
(%i9) 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)                               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
facts () 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 Declaration

Maxima recognizes certain mathematical properties of functions and variables. These are called "features".

`declare (x, foo)` gives the property *foo* to the function or variable *x*.

`declare (foo, feature)` declares a new feature *foo*. For example, `declare ([red, green, blue], feature)` declares three new features, `red`, `green`, and `blue`.

The predicate `featurep (x, foo)` returns `true` if *x* has the *foo* property, and `false` otherwise.

The infolist `features` is a list of known features. These are `integer`, `noninteger`, `even`, `odd`, `rational`, `irrational`, `real`, `imaginary`, `complex`, `analytic`, `increasing`, `decreasing`, `oddfun`, `evenfun`, `posfun`, `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 unkilled 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`.

newcontext (*name*) Function

Creates a new, empty context, called *name*, which has `global` as its only subcontext. The newly-created context becomes the currently active context.

`newcontext` evaluates its argument. `newcontext` returns *name*.

supcontext (*name, context*) Function
supcontext (*name*) Function

Creates a new context, called *name*, which has *context* as a subcontext. *context* must exist.

If *context* is not specified, the current context is assumed.

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*) Function

Returns the coefficient of *x*`^`*n* in *expr*. *n* may be omitted if it is 1. *x* may be an atom, or complete subexpression of *expr* e.g., `sin(x)`, `a[i+1]`, `x + y`, etc. (In the last case the expression `(x + y)` should occur in *expr*). Sometimes it may be necessary to expand or factor *expr* in order to make *x*`^`*n* explicit. This is not done automatically by `coeff`.

Examples:

```
(%i1) coeff (2*a*tan(x) + tan(x) + b = 5*tan(x) + 3, tan(x));
(%o1)                        2 a + 1 = 5
(%i2) coeff (y + x*%e^x + 1, x, 0);
(%o2)                           y + 1
```

combine (*expr*) Function

Simplifies the sum *expr* by combining terms with the same denominator into a single term.

content (*p_1, x_1, ..., x_n*) Function

Returns a list whose first element is the greatest common divisor of the coefficients of the terms of the polynomial *p_1* in the variable *x_n* (this is the content) and whose second element is the polynomial *p_1* divided by the content.

Examples:

```
(%i1) content (2*x*y + 4*x^2*y^2, y);
                                      2
(%o1)                    [2 x, 2 x y  + y]
```

denom (*expr*) Function

Returns the denominator of the rational expression *expr*.

divide (*p_1*, *p_2*, *x_1*, ..., *x_n*) Function

computes the quotient and remainder of the polynomial *p_1* divided by the polynomial *p_2*, in a main polynomial variable, *x_n*. The other variables are as in the `ratvars` function. The result is a list whose first element is the quotient and whose second element is the remainder.

Examples:

```
(%i1) divide (x + y, x - y, x);
(%o1)                         [1, 2 y]
(%i2) divide (x + y, x - y);
(%o2)                        [- 1, 2 x]
```

Note that `y` is the main variable in the second example.

eliminate ([*eqn_1*, ..., *eqn_n*], [*x_1*, ..., *x_k*]) Function

Eliminates variables from equations (or expressions assumed equal to zero) by taking successive resultants. This returns a list of *n* - *k* expressions with the *k* variables *x_1*, ..., *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;
                                      2
(%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;
                           2    2
(%o3)                      z  - y  + x + 5
(%i4) eliminate ([expr3, expr2, expr1], [y, z]);
             8         7         6          5          4
(%o4) [7425 x  - 1170 x  + 1299 x  + 12076 x  + 22887 x

                                      3         2
                                 - 5154 x  - 1291 x  + 7688 x + 15376]
```

ezgcd (*p_1*, *p_2*, *p_3*, ...) Function

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)*n!` into `(n + 1)!`.

`sumsplitfact` if set to `false` will cause `minfactorial` to be applied after a `factcomb`.

factor (*expr*) Function

factor (*expr, p*) 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 integers 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:

```
(%i1) factor (2^63 - 1);
                     2
(%o1)              7  73 127 337 92737 649657
(%i2) factor (-8*y - 4*x + z^2*(2*y + x));
(%o2)                (2 y + x) (z - 2) (z + 2)
(%i3) -1 - 2*x - x^2 + y^2 + 2*x*y^2 + x^2*y^2;
                  2  2        2    2    2
(%o3)           x  y  + 2 x y  + y  - x  - 2 x - 1
(%i4) block ([dontfactor: [x]], factor (%/36/(1 + 2*y + y^2)));
                       2
                      (x  + 2 x + 1) (y - 1)
(%o4)                 ----------------------
                             36 (y + 1)
(%i5) factor (1 + %e^(3*x));
                      x          2 x     x
(%o5)               (%e  + 1) (%e    - %e  + 1)
(%i6) factor (1 + x^4, a^2 - 2);
                     2              2
```

```
(%o6)             (x  - a x + 1) (x  + a x + 1)
(%i7) factor (-y^2*z^2 - x*z^2 + x^2*y^2 + x^3);
                       2
(%o7)               - (y  + x) (z - x) (z + x)
(%i8) (2 + x)/(3 + x)/(b + x)/(c + x)^2;
                             x + 2
(%o8)                -----------------------
                                          2
                     (x + 3) (x + b) (x + c)
(%i9) ratsimp (%);
                   4                    3
(%o9) (x + 2)/(x  + (2 c + b + 3) x

    2                        2             2                    2
 + (c  + (2 b + 6) c + 3 b) x  + ((b + 3) c  + 6 b c) x + 3 b c )
(%i10) partfrac (%, x);
           2                   4                3
(%o10) - (c  - 4 c - b + 6)/((c  + (- 2 b - 6) c

    2              2         2                2
 + (b  + 12 b + 9) c  + (- 6 b  - 18 b) c + 9 b ) (x + c))

                c - 2
 - --------------------------------
    2                            2
   (c  + (- b - 3) c + 3 b) (x + c)

                         b - 2
 + ------------------------------------------------
             2             2       3      2
   ((b - 3) c  + (6 b - 2 b ) c + b  - 3 b ) (x + b)

                         1
 - ---------------------------------------------
             2
   ((b - 3) c  + (18 - 6 b) c + 9 b - 27) (x + 3)
(%i11) map ('factor, %);
              2
             c  - 4 c - b + 6                 c - 2
(%o11) - ------------------------ - ------------------------
                2        2                                2
         (c - 3)  (c - b)  (x + c)   (c - 3) (c - b) (x + c)

                         b - 2                          1
            + ------------------------ - ------------------------
                                  2                              2
              (b - 3) (c - b)  (x + b)   (b - 3) (c - 3)  (x + 3)
(%i12) ratsimp ((x^5 - 1)/(x - 1));
                         4    3    2
```

```
(%o12)                x  + x  + x  + x + 1
(%i13) subst (a, x, %);
                       4    3    2
(%o13)                a  + a  + a  + a + 1
(%i14) factor (%th(2), %);
                       2        3        3    2
(%o14)   (x - a) (x - a ) (x - a ) (x + a  + a  + a + 1)
(%i15) factor (1 + x^12);
                       4        8    4
(%o15)               (x  + 1) (x  - x  + 1)
(%i16) factor (1 + x^99);
                   2           6    3
(%o16) (x + 1) (x  - x + 1) (x  - x  + 1)

  10    9    8    7    6    5    4    3    2
 (x   - x  + x  - x  + x  - x  + x  - x  + x  - x + 1)

  20    19    17    16    14    13    11    10    9    7    6
 (x   + x   - x   - x   + x   + x   - x   - x   - x  + x  + x

   4    3           60    57    51    48    42    39    33
 - x  - x  + x + 1) (x   + x   - x   - x   + x   + x   - x

   30    27    21    18    12    9    3
 - x   - x   + x   + x   - x   - x  + x  + 1)
```

factorflag Option variable

Default value: `false`

When `factorflag` is `false`, suppresses the factoring of integer factors of rational expressions.

factorout (*expr, x_1, x_2, ...*) Function

Rearranges the sum *expr* into a sum of terms of the form `f (x_1, x_2, ...)*g` where `g` is a product of expressions not containing any *x_i* and `f` is factored.

factorsum (*expr*) Function

Tries to group terms in factors of *expr* which are sums into groups of terms such that their sum is factorable. `factorsum` can recover the result of `expand ((x + y)^2 + (z + w)^2)` but it can't recover `expand ((x + 1)^2 + (x + y)^2)` because the terms have variables in common.

Example:

```
(%i1) expand ((x + 1)*((u + v)^2 + a*(w + z)^2));
           2      2                              2      2
(%o1) a x z  + a z  + 2 a w x z + 2 a w z + a w  x + v  x

                                  2       2    2           2
                       + 2 u v x + u  x + a w  + v  + 2 u v + u
```

```
(%i2) factorsum (%);
                                        2          2
(%o2)            (x + 1) (a (z + w)  + (v + u) )
```

fasttimes (*p_1, p_2*) Function

Returns the product of the polynomials *p_1* and *p_2* by using a special algorithm for multiplication of polynomials. `p_1` and `p_2` should be multivariate, dense, and nearly the same size. Classical multiplication is of order `n_1 n_2` where `n_1` is the degree of `p_1` and `n_2` is the degree of `p_2`. `fasttimes` is of order `max (n_1, n_2)^1.585`.

fullratsimp (*expr*) Function

`fullratsimp` repeatedly applies `ratsimp` followed by non-rational simplification to an expression until no further change occurs, and returns the result.

When non-rational expressions are involved, one call to `ratsimp` followed as is usual by non-rational ("general") simplification may not be sufficient to return a simplified result. Sometimes, more than one such call may be necessary. `fullratsimp` makes this process convenient.

`fullratsimp` (*expr, x_1, ..., x_n*) takes one or more arguments similar to `ratsimp` and `rat`.

Example:

```
(%i1) expr: (x^(a/2) + 1)^2*(x^(a/2) - 1)^2/(x^a - 1);
                        a/2     2   a/2     2
                       (x    - 1)  (x    + 1)
(%o1)                  ----------------------
                                a
                               x  - 1
(%i2) ratsimp (expr);
                            2 a      a
                           x    - 2 x  + 1
(%o2)                      ---------------
                                a
                               x  - 1
(%i3) fullratsimp (expr);
                                a
(%o3)                          x  - 1
(%i4) rat (expr);
                         a/2 4       a/2 2
                        (x   )  - 2 (x   )  + 1
(%o4)/R/                ----------------------
                                a
                               x  - 1
```

fullratsubst (*a, b, c*) Function

is the same as `ratsubst` except that it calls itself recursively on its result until that result stops changing. This function is useful when the replacement expression and the replaced expression have one or more variables in common.

`fullratsubst` will also accept its arguments in the format of `lratsubst`. That is, the first argument may be a single substitution equation or a list of such equations, while the second argument is the expression being processed.

`load ("lrats")` loads `fullratsubst` and `lratsubst`.

Examples:

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

- `subst` can carry out multiple substitutions. `lratsubst` is analogous to `subst`.

```
(%i2) subst ([a = b, c = d], a + c);
(%o2)                          d + b
(%i3) lratsubst ([a^2 = b, c^2 = d], (a + e)*c*(a + c));
(%o3)                (d + a c) e + a d + b c
```

- If only one substitution is desired, then a single equation may be given as first argument.

```
(%i4) lratsubst (a^2 = b, a^3);
(%o4)                           a b
```

- `fullratsubst` is equivalent to `ratsubst` except that it recurses until its result stops changing.

```
(%i5) ratsubst (b*a, a^2, a^3);
                                 2
(%o5)                           a  b
(%i6) fullratsubst (b*a, a^2, a^3);
                                   2
(%o6)                           a b
```

- `fullratsubst` also accepts a list of equations or a single equation as first argument.

```
(%i7) fullratsubst ([a^2 = b, b^2 = c, c^2 = a], a^3*b*c);
(%o7)                             b
(%i8) fullratsubst (a^2 = b*a, a^3);
                                   2
(%o8)                           a b
```

- `fullratsubst` may cause an indefinite recursion.

```
(%i9) errcatch (fullratsubst (b*a^2, a^2, a^3));

*** - Lisp stack overflow. RESET
```

gcd (*p_1, p_2, x_1, ...*) Function

Returns the greatest common divisor of *p_1* and *p_2*. The flag `gcd` determines which algorithm is employed. Setting `gcd` to `ez`, `subres`, `red`, or `spmod` selects the `ezgcd`, subresultant `prs`, reduced, or modular algorithm, respectively. If `gcd false` then `gcd` (*p_1*, *p_2*, *x*) always returns 1 for all *x*. Many functions (e.g. `ratsimp`, `factor`, etc.) cause gcd's to be taken implicitly. For homogeneous polynomials it is recommended that `gcd` equal to `subres` be used. To take the gcd when an algebraic is present, e.g., `gcd (x^2 - 2*sqrt(2)*x + 2, x - sqrt(2))`, `algebraic` must be `true` and `gcd` must not be `ez`.

The `gcd` flag, default: `spmod`, if `false` will also prevent the greatest common divisor from being taken when expressions are converted to canonical rational expression (CRE) form. This will sometimes speed the calculation if gcds are not required.

gcdex (*f*, *g*) Function
gcdex (*f*, *g*, *x*) Function

Returns a list `[`*a*`, `*b*`, `*u*`]` where *u* is the greatest common divisor (gcd) of *f* and *g*, and *u* is equal to *a f* `+` *b g*. The arguments *f* and *g* should be univariate polynomials, or else polynomials in *x* a supplied **main** variable since we need to be in a principal ideal domain for this to work. The gcd means the gcd regarding *f* and *g* as univariate polynomials with coefficients being rational functions in the other variables.

`gcdex` implements the Euclidean algorithm, where we have a sequence of `L[i]: [a[i], b[i], r[i]]` which are all perpendicular to `[f, g, -1]` and the next one is built as if `q = quotient(r[i]/r[i+1])` then `L[i+2]: L[i] - q L[i+1]`, and it terminates at `L[i+1]` when the remainder `r[i+2]` is zero.

```
(%i1) gcdex (x^2 + 1, x^3 + 4);
                       2
                      x  + 4 x - 1  x + 4
(%o1)/R/           [- ------------, -----, 1]
                            17        17
(%i2) % . [x^2 + 1, x^3 + 4, -1];
(%o2)/R/                          0
```

Note that the gcd in the following is `1` since we work in `k(y)[x]`, not the `y+1` we would expect in `k[y, x]`.

```
(%i1) gcdex (x*(y + 1), y^2 - 1, x);
                                 1
(%o1)/R/                 [0, ------, 1]
                              2
                             y  - 1
```

gcfactor (*n*) Function

Factors the Gaussian integer *n* over the Gaussian integers, i.e., numbers of the form *a* `+` *b* `%i` where *a* and *b* are rational integers (i.e., ordinary integers). Factors are normalized by making *a* and *b* non-negative.

gfactor (*expr*) Function

Factors the polynomial *expr* over the Gaussian integers (that is, the integers with the imaginary unit `%i` adjoined). This is like `factor (`*expr*`, `*a*`^2+1)` where *a* is `%i`.

Example:

```
(%i1) gfactor (x^4 - 1);
(%o1)           (x - 1) (x + 1) (x - %i) (x + %i)
```

gfactorsum (*expr*) Function

is similar to `factorsum` but applies `gfactor` instead of `factor`.

hipow (*expr*, *x*) Function

Returns the highest explicit exponent of *x* in *expr*. *x* may be a variable or a general expression. If *x* does not appear in *expr*, `hipow` returns `0`.

`hipow` does not consider expressions equivalent to `expr`. In particular, `hipow` does not expand `expr`, so `hipow (`*expr*`, `*x*`)` and `hipow (expand (`*expr*`, `*x*`))` may yield different results.

Examples:

```
(%i1) hipow (y^3 * x^2 + x * y^4, x);
(%o1)                          2
(%i2) hipow ((x + y)^5, x);
(%o2)                          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
```

intfaclim Option variable

Default value: true

If `true`, maxima will give up factorization of integers if no factor is found after trial divisions and Pollard's rho method and factorization will not be complete.

When `intfaclim` is `false` (this is the case when the user calls `factor` explicitly), complete factorization will be attempted. `intfaclim` is set to `false` when factors are computed in `divisors`, `divsum` and `totient`.

Internal calls to `factor` respect the user-specified value of `intfaclim`. Setting `intfaclim` to `true` may reduce the time spent factoring large integers.

keepfloat Option variable

Default value: `false`

When `keepfloat` is `true`, prevents floating point numbers from being rationalized when expressions which contain them are converted to canonical rational expression (CRE) form.

lratsubst (*L*, *expr*) Function

is analogous to `subst (`*L*`, `*expr*`)` except that it uses `ratsubst` instead of `subst`.

The first argument of `lratsubst` is an equation or a list of equations identical in format to that accepted by `subst`. The substitutions are made in the order given by the list of equations, that is, from left to right.

`load ("lrats")` loads `fullratsubst` and `lratsubst`.

Examples:

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

- `subst` can carry out multiple substitutions. `lratsubst` is analogous to `subst`.

```
(%i2) subst ([a = b, c = d], a + c);
(%o2)                          d + b
(%i3) lratsubst ([a^2 = b, c^2 = d], (a + e)*c*(a + c));
(%o3)                (d + a c) e + a d + b c
```

- If only one substitution is desired, then a single equation may be given as first argument.

```
(%i4) lratsubst (a^2 = b, a^3);
(%o4)                           a b
```

modulus Option variable

Default value: `false`

When `modulus` is a positive number *p*, operations on rational numbers (as returned by `rat` and related functions) are carried out modulo *p*, using the so-called "balanced" modulus system in which *n* `modulo` *p* is defined as an integer *k* in `[-(`*p*`-1)/2, ..., 0, ..., (`*p*`-1)/2]` when *p* is odd, or `[-(`*p*`/2 - 1), ..., 0, ...., `*p*`/2]` when *p* is even, such that *a p* + *k* equals *n* for some integer *a*.

If *expr* is already in canonical rational expression (CRE) form when `modulus` is reset, then you may need to re-rat *expr*, e.g., `expr: rat (ratdisrep (expr))`, in order to get correct results.

Typically `modulus` is set to a prime number. If `modulus` is set to a positive non-prime integer, this setting is accepted, but a warning message is displayed. Maxima will allow zero or a negative integer to be assigned to `modulus`, although it is not clear if that has any useful consequences.

num (*expr*) Function

Returns the numerator of *expr* if it is a ratio. If *expr* is not a ratio, *expr* is returned.

`num` evaluates its argument.

polydecomp (*p*, *x*) Function

Decomposes the polynomial *p* in the variable *x* into the functional composition of polynomials in *x*. `polydecomp` returns a list `[`*p_1*`, ..., `*p_n*`]` such that

```
lambda ([x], p_1) (lambda ([x], p_2) (... (lambda ([x], p_n) (x))
  ...))
```

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:

```
(%i1) polydecomp (x^210, x);
                           7   5   3   2
(%o1)                    [x , x , x , x ]
(%i2) p : expand (subst (x^3 - x - 1, x, x^2 - a));
                6      4      3    2
(%o2)          x  - 2 x  - 2 x  + x  + 2 x - a + 1
(%i3) polydecomp (p, x);
                         2       3
(%o3)                 [x  - a, x  - x - 1]
```

The following function composes `L = [e_1, ..., e_n]` as functions in `x`; it is the inverse of polydecomp:

```
compose (L, x) :=
  block ([r : x], for e in L do r : subst (e, x, r), r) $
```

Re-express above example using `compose`:

```
(%i3) polydecomp (compose ([x^2 - a, x^3 - x - 1], x), x);
                          2      3
(%o3)                   [x  - a, x  - x - 1]
```

Note that though `compose (polydecomp (`*p*`, x), x)` always returns *p* (unexpanded), `polydecomp (compose ([`*p_1*`, ..., `*p_n*`], x), x)` does *not* necessarily return `[`*p_1*`, ..., `*p_n*`]`:

```
(%i4) polydecomp (compose ([x^2 + 2*x + 3, x^2], x), x);
                             2       2
(%o4)                      [x  + 2, x  + 1]
(%i5) polydecomp (compose ([x^2 + x + 1, x^2 + x + 1], x), x);
                           2       2
                          x  + 3  x  + 5
(%o5)                    [------, ------, 2 x + 1]
                            4       2
```

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:

```
(%i1) ((x - 2*y)^4/(x^2 - 4*y^2)^2 + 1)*(y + a)*(2*y + x) /
      (4*y^2 + x^2);
                                        4
                                  (x - 2 y)
                (y + a) (2 y + x) (------------ + 1)
                                    2      2 2
                                   (x  - 4 y )
(%o1)           -----------------------------------
                                 2    2
                              4 y  + x
(%i2) rat (%, y, a, x);
                               2 a + 2 y
(%o2)/R/                       ---------
                                x + 2 y
```

ratalgdenom Option variable

Default value: `true`

When `ratalgdenom` is `true`, allows rationalization of denominators with respect to radicals to take effect. `ratalgdenom` has an effect only when canonical rational expressions (CRE) are used in algebraic mode.

ratcoef (*expr, x, n*) Function
ratcoef (*expr, x*) Function

Returns the coefficient of the expression *x*`^`*n* in the expression *expr*. If omitted, *n* is assumed to be 1.

The return value is free (except possibly in a non-rational sense) of the variables in *x*. If no coefficient of this type exists, 0 is returned.

`ratcoef` expands and rationally simplifies its first argument and thus it may produce answers different from those of `coeff` which is purely syntactic. Thus `ratcoef ((x + 1)/y + x, x)` returns `(y + 1)/y` whereas `coeff` returns 1.

`ratcoef (`*expr*`, x, 0)`, viewing *expr* as a sum, returns a sum of those terms which do not contain *x*. Therefore if *x* occurs to any negative powers, `ratcoef` should not be used.

Since *expr* is rationally simplified before it is examined, coefficients may not appear quite the way they were envisioned.

Example:

```
(%i1) s: a*x + b*x + 5$
(%i2) ratcoef (s, a + b);
(%o2)                           x
```

ratdenom (*expr*) Function

Returns the denominator of *expr*, after coercing *expr* to a canonical rational expression (CRE). The return value is a CRE.

expr is coerced to a CRE by `rat` if it is not already a CRE. This conversion may change the form of *expr* by putting all terms over a common denominator.

`denom` is similar, but returns an ordinary expression instead of a CRE. Also, `denom` does not attempt to place all terms over a common denominator, and thus some expressions which are considered ratios by `ratdenom` are not considered ratios by `denom`.

ratdenomdivide Option variable

Default value: `true`

When `ratdenomdivide` is `true`, `ratexpand` expands a ratio in which the numerator is a sum into a sum of ratios, all having a common denominator. Otherwise, `ratexpand` collapses a sum of ratios into a single ratio, the numerator of which is the sum of the numerators of each ratio.

Examples:

```
(%i1) expr: (x^2 + x + 1)/(y^2 + 7);
                              2
                             x  + x + 1
(%o1)                        ----------
                                2
                               y  + 7
(%i2) ratdenomdivide: true$
(%i3) ratexpand (expr);
                         2
                        x        x        1
(%o3)                 ------ + ------ + ------
                      2        2        2
                     y  + 7   y  + 7   y  + 7
(%i4) ratdenomdivide: false$
(%i5) ratexpand (expr);
                              2
                             x  + x + 1
(%o5)                        ----------
                                2
                               y  + 7
(%i6) expr2: a^2/(b^2 + 3) + b/(b^2 + 3);
                                       2
                             b        a
(%o6)                      ------ + ------
                           2        2
                          b  + 3   b  + 3
(%i7) ratexpand (expr2);
                                 2
                               b + a
(%o7)                          ------
```

```
 2
b  + 3
```

ratdiff (*expr*, *x*) 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:

```
(%i1) expr: (4*x^3 + 10*x - 11)/(x^5 + 5);
                          3
                       4 x  + 10 x - 11
(%o1)                  ----------------
                             5
                            x  + 5
(%i2) ratdiff (expr, x);
                    7       5       4       2
                  8 x  + 40 x  - 55 x  - 60 x  - 50
(%o2)          - --------------------------------
                           10       5
                          x   + 10 x  + 25
(%i3) expr: f(x)^3 - f(x)^2 + 7;
                          3       2
(%o3)                  f (x) - f (x) + 7
(%i4) ratdiff (expr, f(x));
                           2
(%o4)                  3 f (x) - 2 f(x)
(%i5) expr: (a + b)^3 + (a + b)^2;
                                3          2
(%o5)                  (b + a)  + (b + a)
(%i6) ratdiff (expr, a + b);
                    2                   2
(%o6)           3 b  + (6 a + 2) b + 3 a  + 2 a
```

ratdisrep (*expr*) Function

Returns its argument as a general expression. If *expr* is a general expression, it is returned unchanged.

Typically `ratdisrep` is called to convert a canonical rational expression (CRE) into a general expression. This is sometimes convenient if one wishes to stop the "contagion", or use rational functions in non-rational contexts.

See also `totaldisrep`.

ratepsilon Option variable

Default value: 2.0e-8

`ratepsilon` is the tolerance used in the conversion of floating point numbers to rational numbers.

ratexpand (*expr*) Function

ratexpand Option variable

Expands *expr* by multiplying out products of sums and exponentiated sums, combining fractions over a common denominator, cancelling the greatest common divisor of the numerator and denominator, then splitting the numerator (if a sum) into its respective terms divided by the denominator.

The return value of `ratexpand` is a general expression, even if *expr* is a canonical rational expression (CRE).

The switch `ratexpand` if `true` will cause CRE expressions to be fully expanded when they are converted back to general form or displayed, while if it is `false` then they will be put into a recursive form. See also `ratsimp`.

When `ratdenomdivide` is `true`, `ratexpand` expands a ratio in which the numerator is a sum into a sum of ratios, all having a common denominator. Otherwise, `ratexpand` collapses a sum of ratios into a single ratio, the numerator of which is the sum of the numerators of each ratio.

When `keepfloat` is `true`, prevents floating point numbers from being rationalized when expressions which contain them are converted to canonical rational expression (CRE) form.

Examples:

```
(%i1) ratexpand ((2*x - 3*y)^3);
                      3         2       2        3
(%o1)          - 27 y  + 54 x y  - 36 x  y + 8 x
(%i2) expr: (x - 1)/(x + 1)^2 + 1/(x - 1);
                           x - 1       1
(%o2)                    -------- + -----
                                 2   x - 1
                          (x + 1)
(%i3) expand (expr);
                        x              1           1
(%o3)           ------------ - ------------ + -----
                 2             2             x - 1
                x  + 2 x + 1   x  + 2 x + 1
(%i4) ratexpand (expr);
                          2
                       2 x                 2
(%o4)           --------------- + ---------------
                 3    2            3    2
                x  + x  - x - 1   x  + x  - x - 1
```

ratfac Option variable

Default value: `false`

When `ratfac` is `true`, canonical rational expressions (CRE) are manipulated in a partially factored form.

During rational operations the expression is maintained as fully factored as possible without calling `factor`. This should always save space and may save time in some computations. The numerator and denominator are made relatively prime, for example `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 `ratfac` and `ratweight` schemes are incompatible and may not both be used at the same time.

ratnumer (*expr*) Function

Returns the numerator of *expr*, after coercing *expr* to a canonical rational expression (CRE). The return value is a CRE.

expr is coerced to a CRE by `rat` if it is not already a CRE. This conversion may change the form of *expr* by putting all terms over a common denominator.

`num` is similar, but returns an ordinary expression instead of a CRE. Also, `num` does not attempt to place all terms over a common denominator, and thus some expressions which are considered ratios by `ratnumer` are not considered ratios by `num`.

ratnump (*expr*) Function

Returns `true` if *expr* is a literal integer or ratio of literal integers, otherwise `false`.

ratp (*expr*) Function

Returns `true` if *expr* is a canonical rational expression (CRE) or extended CRE, otherwise `false`.

CRE are created by `rat` and related functions. Extended CRE are created by `taylor` and related functions.

ratprint Option variable

Default value: `true`

When `ratprint` is `true`, a message informing the user of the conversion of floating point numbers to rational numbers is displayed.

ratsimp (*expr*) Function

ratsimp (*expr, x_1, ..., x_n*) Function

Simplifies the expression *expr* and all of its subexpressions, including the arguments to non-rational functions. The result is returned as the quotient of two polynomials in a recursive form, that is, the coefficients of the main variable are polynomials in the other variables. Variables may include non-rational functions (e.g., `sin (x^2 + 1)`) and the arguments to any such functions are also rationally simplified.

`ratsimp (`*expr*`, `*x_1*`, ..., `*x_n*`)` enables rational simplification with the specification of variable ordering as in `ratvars`.

When `ratsimpexpons` is `true`, `ratsimp` is applied to the exponents of expressions during simplification.

See also `ratexpand`. Note that `ratsimp` is affected by some of the flags which affect `ratexpand`.

Examples:

```
(%i1) sin (x/(x^2 + x)) = exp ((log(x) + 1)^2 - log(x)^2);
                                                2      2
                     x          (log(x) + 1)  - log (x)
(%o1)        sin(------) = %e
                  2
                 x  + x
(%i2) ratsimp (%);
                              1          2
(%o2)                  sin(-----) = %e x
                           x + 1
(%i3) ((x - 1)^(3/2) - (x + 1)*sqrt(x - 1))/sqrt((x - 1)*(x + 1));
                          3/2
                (x - 1)    - sqrt(x - 1) (x + 1)
(%o3)           -------------------------------
                     sqrt((x - 1) (x + 1))
(%i4) ratsimp (%);
                             2 sqrt(x - 1)
(%o4)                      - -------------
                                    2
                             sqrt(x  - 1)
(%i5) x^(a + 1/a), ratsimpexpons: true;
                                2
                               a  + 1
                               ------
                                 a
(%o5)                         x
```

ratsimpexpons 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 `z + a`.

When `radsubstflag` is `true`, `ratsubst` makes substitutions for radicals in expressions which don't explicitly contain them.

Examples:

```
(%i1) ratsubst (a, x*y^2, x^4*y^3 + x^4*y^8);
                                3      4
(%o1)                        a x  y + a
(%i2) cos(x)^4 + cos(x)^3 + cos(x)^2 + cos(x) + 1;
                4         3         2
(%o2)       cos (x) + cos (x) + cos (x) + cos(x) + 1
(%i3) ratsubst (1 - sin(x)^2, cos(x)^2, %);
              4           2                      2
(%o3)    sin (x) - 3 sin (x) + cos(x) (2 - sin (x)) + 3
(%i4) ratsubst (1 - cos(x)^2, sin(x)^2, sin(x)^4);
                         4           2
(%o4)                 cos (x) - 2 cos (x) + 1
(%i5) radsubstflag: false$
(%i6) ratsubst (u, sqrt(x), x);
(%o6)                             x
(%i7) radsubstflag: true$
(%i8) ratsubst (u, sqrt(x), x);
                                  2
(%o8)                            u
```

ratvars (*x_1, ..., x_n*) Function

ratvars () Function

ratvars System variable

Declares main variables *x_1*, ..., *x_n* for rational expressions. *x_n*, if present in a rational expression, is considered the main variable. Otherwise, *x_[n-1]* is considered the main variable if present, and so on through the preceding variables to *x_1*, which is considered the main variable only if none of the succeeding variables are present.

If a variable in a rational expression is not present in the `ratvars` list, it is given a lower priority than *x_1*.

The arguments to `ratvars` can be either variables or non-rational functions such as `sin(x)`.

The variable `ratvars` is a list of the arguments of the function `ratvars` when it was called most recently. Each call to the function `ratvars` resets the list. `ratvars ()` clears the list.

ratweight (*x_1, w_1, ..., x_n, w_n*) Function

ratweight () Function

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.

`kill (ratweights)` and `save (ratweights)` both work as expected.

ratwtlvl Option variable

Default value: `false`

`ratwtlvl` is used in combination with the `ratweight` function to control the truncation of canonical rational expressions (CRE). For the default value of `false`, no truncation occurs.

remainder (*p_1*, *p_2*) Function
remainder (*p_1*, *p_2*, *x_1*, ..., *x_n*) Function

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*) Function
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
(%o1)                    (2 x + 1)  (x  - 1)
```

tellrat (*p_1*, ..., *p_n*) Function

tellrat () Function

Adds to the ring of algebraic integers known to Maxima the elements which are the solutions of the polynomials *p_1*, ..., *p_n*. Each argument *p_i* is a polynomial with integer coefficients.

`tellrat (x)` effectively means substitute 0 for *x* in rational functions.

`tellrat ()` returns a list of the current substitutions.

`algebraic` must be set to `true` in order for the simplification of algebraic integers to take effect.

Maxima initially knows about the imaginary unit `%i` and all roots of integers.

There is a command `untellrat` which takes kernels and removes `tellrat` properties.

When `tellrat`'ing a multivariate polynomial, e.g., `tellrat (x^2 - y^2)`, there would be an ambiguity as to whether to substitute `y^2` for `x^2` or vice versa. Maxima picks a particular ordering, but if the user wants to specify which, e.g. `tellrat (y^2 = x^2)` provides a syntax which says replace `y^2` by `x^2`.

Examples:

```
(%i1) 10*(%i + 1)/(%i + 3^(1/3));
                             10 (%i + 1)
(%o1)                        -----------
                                    1/3
                              %i + 3
(%i2) ev (ratdisrep (rat(%)), algebraic);
             2/3      1/3              2/3      1/3
(%o2)    (4 3    - 2 3    - 4) %i + 2 3    + 4 3    - 2
(%i3) tellrat (1 + a + a^2);
                              2
(%o3)                        [a  + a + 1]
(%i4) 1/(a*sqrt(2) - 1) + a/(sqrt(3) + sqrt(2));
                         1                a
```

```
(%o4)           ------------- + -----------------
                sqrt(2) a - 1   sqrt(3) + sqrt(2)
(%i5) ev (ratdisrep (rat(%)), algebraic);
         (7 sqrt(3) - 10 sqrt(2) + 2) a - 2 sqrt(2) - 1
(%o5)    ---------------------------------------------
                                7
(%i6) tellrat (y^2 = x^2);
                         2    2   2
(%o6)                 [y  - x , a  + a + 1]
```

totaldisrep (*expr*) Function

Converts every subexpression of *expr* from canonical rational expressions (CRE) to general form and returns the result. If *expr* is itself in CRE form then `totaldisrep` is identical to `ratdisrep`.

`totaldisrep` may be useful for ratdisrepping expressions such as equations, lists, matrices, etc., which have some subexpressions in CRE form.

untellrat (*x_1, ..., x_n*) Function

Removes `tellrat` properties from *x_1*, ..., *x_n*.

13 Constants

13.1 Functions and Variables for Constants

%e Constant

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

%i Constant

`%i` represents the imaginary unit, $sqrt(-1)$.

false Constant

`false` represents the Boolean constant of the same name. Maxima implements `false` by the value `NIL` in Lisp.

ind Constant

`ind` represents a bounded, indefinite result.

See also `limit`.

Example:

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

inf Constant

`inf` represents real positive infinity.

infinity Constant

`infinity` represents complex infinity.

minf Constant

`minf` represents real minus (i.e., negative) infinity.

%phi Constant

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

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

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

Examples:

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

```
(%i1) fibtophi (fib (n));
                            n             n
                        %phi  - (1 - %phi)
(%o1)                   ------------------
                            2 %phi - 1
(%i2) fib (n-1) + fib (n) - fib (n+1);
(%o2)          - fib(n + 1) + fib(n) + fib(n - 1)
(%i3) fibtophi (%);
             n + 1             n + 1       n             n
         %phi      - (1 - %phi)         %phi  - (1 - %phi)
(%o3) - -------------------------- + ------------------
                2 %phi - 1                2 %phi - 1
                                          n - 1             n - 1
                                       %phi      - (1 - %phi)
                                     + --------------------------
                                                2 %phi - 1
(%i4) ratsimp (%);
(%o4)                           0
```

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

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

%pi Constant

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

true Constant

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

und Constant

`und` represents an undefined result.

See also `limit`.

Example:

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

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 `radcan` command also does this transformation, and more complicated transformations of this ilk as well. The `logcontract` command "contracts" expressions containing `log`.

li [*s*] (*z*) Function

Represents the polylogarithm function of order *s* and argument *z*, defined by the infinite series

$$\mathrm{Li}_s(z) = \sum_{k=1}^{\inf} \frac{z^k}{k^s}$$

`li [1]` is `- log (1 - z)`. `li [2]` and `li [3]` are the dilogarithm and trilogarithm functions, respectively.

When the order is 1, the polylogarithm simplifies to `- log (1 - z)`, which in turn simplifies to a numerical value if *z* is a real or complex floating point number or the `numer` evaluation flag is present.

When the order is 2 or 3, the polylogarithm simplifies to a numerical value if *z* is a real floating point number or the `numer` evaluation flag is present.

Examples:

```
(%i1) assume (x > 0);
(%o1)                          [x > 0]
(%i2) integrate ((log (1 - t)) / t, t, 0, x);
(%o2)                        - li (x)
                                 2
(%i3) li [2] (7);
(%o3)                         li (7)
                                2
(%i4) li [2] (7), numer;
(%o4)        1.24827317833392 - 6.113257021832577 %i
(%i5) li [3] (7);
(%o5)                         li (7)
                                3
(%i6) li [2] (7), numer;
(%o6)        1.24827317833392 - 6.113257021832577 %i
(%i7) L : makelist (i / 4.0, i, 0, 8);
```

```
(%o7)   [0.0, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0]
(%i8) map (lambda ([x], li [2] (x)), L);
(%o8) [0, .2676526384986274, .5822405249432515,
.9784693966661848, 1.64493407, 2.190177004178597
 - .7010261407036192 %i, 2.374395264042415
 - 1.273806203464065 %i, 2.448686757245154
 - 1.758084846201883 %i, 2.467401098097648
 - 2.177586087815347 %i]
(%i9) map (lambda ([x], li [3] (x)), L);
(%o9) [0, .2584613953442624, 0.537213192678042,
.8444258046482203, 1.2020569, 1.642866878950322
 - .07821473130035025 %i, 2.060877505514697
 - .2582419849982037 %i, 2.433418896388322
 - .4919260182322965 %i, 2.762071904015935
 - .7546938285978846 %i]
```

log (x) Function

Represents the natural (base e) logarithm of x.

Maxima does not have a built-in function for the base 10 logarithm or other bases. `log10(x) := log(x) / log(10)` is a useful definition.

Simplification and evaluation of logarithms is governed by several global flags:

`logexpand` - causes `log(a^b)` to become `b*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#1`. (`log(1/b)`, for `b` integer, always simplifies.) If it is set to `false`, all of these simplifications will be turned off.

`logsimp` - if `false` then no simplification of `%e` to a power containing `log`'s is done.

`lognumer` - 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) -> log(n)+%i*%pi` for `n` a positive integer.

`%e_to_numlog` - 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 `radcan` command also does this transformation, and more complicated transformations of this ilk as well. The `logcontract` command "contracts" expressions containing `log`.

logabs Option variable

Default value: `false`

When doing indefinite integration where logs are generated, e.g. `integrate(1/x,x)`, the answer is given in terms of `log(abs(...))` if `logabs` is `true`, but in terms of `log(...)` if `logabs` is `false`. For definite integration, the `logabs:true` setting is used, because here "evaluation" of the indefinite integral at the endpoints is often needed.

logarc Option variable
logarc (*expr*) Function

When the global variable `logarc` is `true`, inverse circular and hyperbolic functions are replaced by equivalent logarithmic functions. The default value of `logarc` is `false`.

The function `logarc(expr)` carries out that replacement for an expression *expr* without setting the global variable `logarc`.

logconcoeffp Option variable

Default value: `false`

Controls which coefficients are contracted when using `logcontract`. It may be set to the name of a predicate function of one argument. E.g. if you like to generate SQRTs, you can do `logconcoeffp:'logconfun$ logconfun(m):=featurep(m,integer) or ratnump(m)$` . Then `logcontract(1/2*log(x));` will give `log(sqrt(x))`.

logcontract (*expr*) Function

Recursively scans the expression *expr*, transforming subexpressions of the form `a1*log(b1) + a2*log(b2) + c` into `log(ratsimp(b1^a1 * b2^a2)) + c`

```
(%i1) 2*(a*log(x) + 2*a*log(y))$
(%i2) logcontract(%);
                                    2  4
(%o2)                       a log(x  y )
```

If you do `declare(n,integer);` then `logcontract(2*a*n*log(x));` gives `a*log(x^(2*n))`. The coefficients that "contract" in this manner are those such as the 2 and the `n` here which satisfy `featurep(coeff,integer)`. The user can control which coefficients are contracted by setting the option `logconcoeffp` to the name of a predicate function of one argument. E.g. if you like to generate SQRTs, you can do `logconcoeffp:'logconfun$ logconfun(m):=featurep(m,integer) or ratnump(m)$` . Then `logcontract(1/2*log(x));` will give `log(sqrt(x))`.

logexpand Option variable

Default value: `true`

Causes `log(a^b)` to become `b*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#1`. (`log(1/b)`, for integer `b`, always simplifies.) If it is set to `false`, all of these simplifications will be turned off.

lognegint Option variable

Default value: `false`

If `true` implements the rule `log(-n) -> log(n)+%i*%pi` for `n` a positive integer.

lognumer 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` .

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: `acos`, `acosh`, `acot`, `acoth`, `acsc`, `acsch`, `asec`, `asech`, `asin`, `asinh`, `atan`, `atanh`, `cos`, `cosh`, `cot`, `coth`, `csc`, `csch`, `sec`, `sech`, `sin`, `sinh`, `tan`, and `tanh`. There are a number of commands especially for handling trigonometric functions, see `trigexpand`, `trigreduce`, and the switch `trigsign`. Two share packages extend the simplification rules built into Maxima, `ntrig` and `atrig1`. Do `describe(`*command*`)` for details.

15.2 Functions and Variables for Trigonometric

%piargs — Option variable

Default value: `true`

When `%piargs` is `true`, trigonometric functions are simplified to algebraic constants when the argument is an integer multiple of π, $\pi/2$, $\pi/3$, $\pi/4$, or $\pi/6$.

Maxima knows some identities which can be applied when π, etc., are multiplied by an integer variable (that is, a symbol declared to be integer).

Examples:

```
(%i1) %piargs : false;
(%o1)                          false
(%i2) [sin (%pi), sin (%pi/2), sin (%pi/3)];
                                %pi       %pi
(%o2)            [sin(%pi), sin(---), sin(---)]
                                 2         3
(%i3) [sin (%pi/4), sin (%pi/5), sin (%pi/6)];
                      %pi       %pi       %pi
(%o3)            [sin(---), sin(---), sin(---)]
                       4         5         6
(%i4) %piargs : true;
(%o4)                          true
(%i5) [sin (%pi), sin (%pi/2), sin (%pi/3)];
                                sqrt(3)
(%o5)                    [0, 1, -------]
                                  2
(%i6) [sin (%pi/4), sin (%pi/5), sin (%pi/6)];
                         1         %pi   1
(%o6)                [-------, sin(---), -]
                      sqrt(2)       5    2
(%i7) [cos (%pi/3), cos (10*%pi/3), tan (10*%pi/3), cos (sqrt(2)*%pi/3)];
                1    1               sqrt(2) %pi
(%o7)          [-, - -, sqrt(3), cos(-----------)]
```

```
                    2    2                    3
```

Some identities are applied when π and $\pi/2$ are multiplied by an integer variable.

```
(%i1) declare (n, integer, m, even);
(%o1)                         done
(%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;
(%o1)                         false
(%i2) [sin (%i * x), cos (%i * x), tan (%i * x)];
(%o2)          [sin(%i x), cos(%i x), tan(%i x)]
(%i3) %iargs : true;
(%o3)                         true
(%i4) [sin (%i * x), cos (%i * x), tan (%i * x)];
(%o4)          [%i sinh(x), cosh(x), %i tanh(x)]
```

Even when the argument is demonstrably real, the simplification is applied.

```
(%i1) declare (x, imaginary);
(%o1)                         done
(%i2) [featurep (x, imaginary), featurep (x, real)];
(%o2)                     [true, false]
(%i3) sin (%i * x);
(%o3)                      %i sinh(x)
```

acos (x) Function

- Arc Cosine.

acosh (x) Function

- Hyperbolic Arc Cosine.

acot (x) Function

- Arc Cotangent.

acoth (x) Function

- Hyperbolic Arc Cotangent.

acsc (x) Function

- Arc Cosecant.

acsch (x) Function
- Hyperbolic Arc Cosecant.

asec (x) Function
- Arc Secant.

asech (x) Function
- Hyperbolic Arc Secant.

asin (x) Function
- Arc Sine.

asinh (x) Function
- Hyperbolic Arc Sine.

atan (x) Function
- Arc Tangent.

atan2 (y, x) Function
- yields the value of `atan(`y/x`)` in the interval `-%pi` to `%pi`.

atanh (x) Function
- Hyperbolic Arc Tangent.

atrig1 Package
The `atrig1` package contains several additional simplification rules for inverse trigonometric functions. Together with rules already known to Maxima, the following angles are fully implemented: `0`, `%pi/6`, `%pi/4`, `%pi/3`, and `%pi/2`. Corresponding angles in the other three quadrants are also available. Do `load(atrig1);` to use them.

cos (x) Function
- Cosine.

cosh (x) Function
- Hyperbolic Cosine.

cot (x) Function
- Cotangent.

coth (x) Function
- Hyperbolic Cotangent.

csc (x) Function
- Cosecant.

csch (*x*) Function

- Hyperbolic Cosecant.

halfangles Option variable

Default value: `false`

When `halfangles` is `true`, trigonometric functions of arguments *expr*`/2` are simplified to functions of *expr*.

Examples:

```
(%i1) halfangles : false;
(%o1)                          false
(%i2) sin (x / 2);
                                  x
(%o2)                         sin(-)
                                  2
(%i3) halfangles : true;
(%o3)                          true
(%i4) sin (x / 2);
                          sqrt(1 - cos(x))
(%o4)                     ----------------
                              sqrt(2)
```

ntrig Package

The `ntrig` package contains a set of simplification rules that are used to simplify trigonometric function whose arguments are of the form *f*`(`*n* `%pi/10)` where *f* is any of the functions `sin`, `cos`, `tan`, `csc`, `sec` and `cot`.

sec (*x*) Function

- Secant.

sech (*x*) Function

- Hyperbolic Secant.

sin (*x*) Function

- Sine.

sinh (*x*) Function

- Hyperbolic Sine.

tan (*x*) Function

- Tangent.

tanh (*x*) Function

- Hyperbolic Tangent.

trigexpand (*expr*) Function

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

`trigexpand` is governed by the following global flags:

`trigexpand`
: If `true` causes expansion of all expressions containing sin's and cos's occurring subsequently.

`halfangles`
: If `true` causes half-angles to be simplified away.

`trigexpandplus`
: Controls the "sum" rule for `trigexpand`, expansion of sums (e.g. `sin(x + y)`) will take place only if `trigexpandplus` is `true`.

`trigexpandtimes`
: Controls the "product" rule for `trigexpand`, expansion of products (e.g. `sin(2 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));
(%o2)          cos(10 x) sin(y) + sin(10 x) cos(y)
```

trigexpandplus Option variable

Default value: `true`

`trigexpandplus` controls the "sum" rule for `trigexpand`. Thus, when the `trigexpand` command is used or the `trigexpand` switch set to `true`, expansion of sums (e.g. `sin(x+y)`) will take place only if `trigexpandplus` is `true`.

trigexpandtimes Option variable

Default value: `true`

`trigexpandtimes` controls the "product" rule for `trigexpand`. Thus, when the `trigexpand` command is used or the `trigexpand` switch set to `true`, expansion of products (e.g. `sin(2*x)`) will take place only if `trigexpandtimes` is `true`.

triginverses Option variable

Default value: `all`

`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
trigreduce (*expr*) 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);
                cos(2 x)      cos(2 x)   1        1
(%o1)           -------- + 3 (-------- + -) + x - -
                   2             2       2        2
```

The trigonometric simplification routines will use declared information in some simple cases. Declarations about variables are used as follows, e.g.

```
(%i1) declare(j, integer, e, even, o, odd)$
(%i2) sin(x + (e + 1/2)*%pi);
(%o2)                         cos(x)
(%i3) sin(x + (o + 1/2)*%pi);
(%o3)                        - cos(x)
```

trigsign Option variable

Default value: `true`

When `trigsign` is `true`, it permits simplification of negative arguments to trigonometric functions. E.g., `sin(-x)` will become `-sin(x)` only if `trigsign` is `true`.

trigsimp (*expr*) Function

Employs the identities $sin(x)^2 + cos(x)^2 = 1$ and $cosh(x)^2 - sinh(x)^2 = 1$ to simplify expressions containing `tan`, `sec`, etc., to `sin`, `cos`, `sinh`, `cosh`.

`trigreduce`, `ratsimp`, and `radcan` may be able to further simplify the result.

`demo ("trgsmp.dem")` displays some examples of `trigsimp`.

trigrat (*expr*) Function

Gives a canonical simplifyed quasilinear form of a trigonometrical expression; *expr* is a rational fraction of several `sin`, `cos` or `tan`, the arguments of them are linear forms in some variables (or kernels) and `%pi`/*n* (*n* integer) with integer coefficients. The result is a simplified fraction with numerator and denominator linear in `sin` and `cos`. Thus `trigrat` linearize always when it is possible.

```
(%i1) trigrat(sin(3*a)/sin(a+%pi/3));
(%o1)            sqrt(3) sin(2 a) + cos(2 a) - 1
```

The following example is taken from Davenport, Siret, and Tournier, *Calcul Formel*, Masson (or in English, Addison-Wesley), section 1.5.5, Morley theorem.

```
(%i1) c: %pi/3 - a - b;
                                        %pi
(%o1)                       - b - a + ---
```

```
                                      3
(%i2) bc: sin(a)*sin(3*c)/sin(a+b);
                      sin(a) sin(3 b + 3 a)
(%o2)                 ---------------------
                           sin(b + a)
(%i3) ba: bc, c=a, a=c$
(%i4) ac2: ba^2 + bc^2 - 2*bc*ba*cos(b);
         2       2
      sin (a) sin (3 b + 3 a)
(%o4) -----------------------
                2
             sin (b + a)

                                          %pi
  2 sin(a) sin(3 a) cos(b) sin(b + a - ---) sin(3 b + 3 a)
                                           3
 - ------------------------------------------------------
                           %pi
                  sin(a - ---) sin(b + a)
                           3

     2         2         %pi
  sin (3 a) sin (b + a - ---)
                           3
 + --------------------------
             2     %pi
          sin (a - ---)
                    3
(%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`
`%he[n] (z)`	`Hermite polynomial (Nota bene: he, not h. See A&S 22.5.18)`
`assoc_legendre_p[v,u] (z)`	`Legendre function of degree v and order u`
`assoc_legendre_q[v,u] (z)`	`Legendre function, 2nd kind`
`hstruve[n] (z)`	`Struve H function`
`lstruve[n] (z)`	`Struve L function`
`%f[p,q] ([], [], expr)`	`Generalized Hypergeometric function`
`gamma()`	`Gamma function`
`gammagreek(a,z)`	`Incomplete gamma function`
`gammaincomplete(a,z)`	`Tail of incomplete gamma function`
`slommel`	
`%m[u,k] (z)`	`Whittaker function, 1st kind`
`%w[u,k] (z)`	`Whittaker function, 2nd kind`
`erfc (z)`	`Complement of the erf function`
`ei (z)`	`Exponential integral (?)`
`kelliptic (z)`	`Complete elliptic integral of the first kind (K)`
`%d [n] (z)`	`Parabolic cylinder function`

16.2 Functions and Variables for Special Functions

airy_ai (x) Function

The Airy function Ai, as defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Section 10.4.

The Airy equation `diff (y(x), x, 2) - x y(x) = 0` has two linearly independent solutions, `y = Ai(x)` and `y = Bi(x)`. The derivative `diff (airy_ai(x), x)` is `airy_dai(x)`.

If the argument `x` is a real or complex floating point number, the numerical value of `airy_ai` is returned when possible.

See also `airy_bi`, `airy_dai`, `airy_dbi`.

airy_dai (x) Function

The derivative of the Airy function Ai `airy_ai(x)`.

See `airy_ai`.

airy_bi (x) Function

The Airy function Bi, as defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Section 10.4, is the second solution of the Airy equation `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 `airy_bi` is returned when possible. In other cases the unevaluated expression is returned.

The derivative `diff (airy_bi(x), x)` is `airy_dbi(x)`.

See `airy_ai`, `airy_dbi`.

airy_dbi (x) Function

The derivative of the Airy Bi function `airy_bi(x)`.

See `airy_ai` and `airy_bi`.

asympa Function

`asympa` is a package for asymptotic analysis. The package contains simplification functions for asymptotic analysis, including the "big O" and "little o" functions that are widely used in complexity analysis and numerical analysis.

`load ("asympa")` loads this package.

bessel (z, a) Function

The Bessel function of the first kind.

This function is deprecated. Write `bessel_j (z, a)` instead.

bessel_j (v, z) Function

The Bessel function of the first kind of order v and argument z.

`bessel_j` computes the array `besselarray` such that `besselarray [i] = bessel_j [i + v - int(v)] (z)` for `i` from zero to `int(v)`.

`bessel_j` is defined as

$$\sum_{k=0}^{\infty} \frac{(-1)^k \ \left(\frac{z}{2}\right)^{v+2\,k}}{k!\,\Gamma\left(v+k+1\right)}$$

although the infinite series is not used for computations.

bessel_y (v, z) Function

The Bessel function of the second kind of order v and argument z.

`bessel_y` computes the array `besselarray` such that `besselarray [i] = bessel_y [i + v - int(v)] (z)` for `i` from zero to `int(v)`.

`bessel_y` is defined as

$$\frac{\cos\left(\pi\,v\right)\ J_v(z) - J_{-v}(z)}{\sin\left(\pi\,v\right)}$$

when v is not an integer. When v is an integer n, the limit as v approaches n is taken.

bessel_i (*v*, *z*) Function

The modified Bessel function of the first kind of order v and argument z.

`bessel_i` computes the array `besselarray` such that `besselarray [i] = bessel_i [i + v - int(v)] (z)` for `i` from zero to `int(v)`.

`bessel_i` is defined as

$$\sum_{k=0}^{\infty} \frac{1}{k!\,\Gamma\left(v+k+1\right)} \left(\frac{z}{2}\right)^{v+2\,k}$$

although the infinite series is not used for computations.

bessel_k (*v*, *z*) Function

The modified Bessel function of the second kind of order v and argument z.

`bessel_k` computes the array `besselarray` such that `besselarray [i] = bessel_k [i + v - int(v)] (z)` for `i` from zero to `int(v)`.

`bessel_k` is defined as

$$\frac{\pi\,\csc\left(\pi\,v\right)\,\left(I_{-v}(z)-I_{v}(z)\right)}{2}$$

when v is not an integer. If v is an integer n, then the limit as v approaches n is taken.

besselexpand Option variable

Default value: `false`

Controls expansion of the Bessel functions when the order is half of an odd integer. In this case, the Bessel functions can be expanded in terms of other elementary functions. When `besselexpand` is `true`, the Bessel function is expanded.

```
(%i1) besselexpand: false$
(%i2) bessel_j (3/2, z);
                                    3
(%o2)                      bessel_j(-, z)
                                    2
(%i3) besselexpand: true$
(%i4) bessel_j (3/2, z);
                           2 z   sin(z)   cos(z)
(%o4)                 sqrt(---) (------ - ------)
                           %pi      2       z
                                   z
```

scaled_bessel_i (*v*, *z*) Function

The scaled modified Bessel function of the first kind of order v and argument z. That is, $scaled_besself_i(v,z) = exp(-abs(z)) * bessel_i(v,z)$. This function is particularly useful for calculating $bessel_i$ for large z, which is large. However, maxima does not otherwise know much about this function. For symbolic work, it is probably preferable to work with the expression `exp(-abs(z))*bessel_i(v, z)`.

scaled_bessel_i0 (*z*) Function

Identical to `scaled_bessel_i(0,z)`.

scaled_bessel_i1 (*z*) Function

Identical to `scaled_bessel_i(1,z)`.

beta (*x*, *y*) Function

The beta function, defined as `gamma(x) gamma(y)/gamma(x + y)`.

gamma (*x*) Function

The gamma function.

See also `makegamma`.

The variable `gammalim` controls simplification of the gamma function.

The Euler-Mascheroni constant is `%gamma`.

gammalim Option variable

Default value: 1000000

`gammalim` controls simplification of the gamma function for integral and rational number arguments. If the absolute value of the argument is not greater than `gammalim`, then simplification will occur. Note that the `factlim` switch controls simplification of the result of `gamma` of an integer argument as well.

intopois (*a*) Function

Converts *a* into a Poisson encoding.

makefact (*expr*) Function

Transforms instances of binomial, gamma, and beta functions in *expr* into factorials.

See also `makegamma`.

makegamma (*expr*) Function

Transforms instances of binomial, factorial, and beta functions in *expr* into gamma functions.

See also `makefact`.

numfactor (*expr*) Function

Returns the numerical factor multiplying the expression *expr*, which should be a single term.

`content` returns the greatest common divisor (gcd) of all terms in a sum.

```
(%i1) gamma (7/2);
                              15 sqrt(%pi)
(%o1)                         ------------
                                   8
(%i2) numfactor (%);
                                   15
(%o2)                              --
                                   8
```

outofpois (*a*) Function

Converts *a* from Poisson encoding to general representation. If *a* is not in Poisson form, `outofpois` carries out the conversion, i.e., the return value is `outofpois (intopois (a))`. This function is thus a canonical simplifier for sums of powers of sine and cosine terms of a particular type.

poisdiff (*a*, *b*) Function

Differentiates *a* with respect to *b*. *b* must occur only in the trig arguments or only in the coefficients.

poisexpt (*a*, *b*) Function

Functionally identical to `intopois (a^b)`. *b* must be a positive integer.

poisint (*a*, *b*) Function

Integrates in a similarly restricted sense (to `poisdiff`). Non-periodic terms in *b* are dropped if *b* is in the trig arguments.

poislim Option variable

Default value: 5

`poislim` determines the domain of the coefficients in the arguments of the trig functions. The initial value of 5 corresponds to the interval [-2^(5-1)+1,2^(5-1)], or [-15,16], but it can be set to [-2^(n-1)+1, 2^(n-1)].

poismap (*series*, *sinfn*, *cosfn*) Function

will map the functions *sinfn* on the sine terms and *cosfn* on the cosine terms of the Poisson series given. *sinfn* and *cosfn* are functions of two arguments which are a coefficient and a trigonometric part of a term in series respectively.

poisplus (*a*, *b*) Function

Is functionally identical to `intopois (a + b)`.

poissimp (*a*) Function

Converts *a* into a Poisson series for *a* in general representation.

poisson Special symbol

The symbol `/P/` follows the line label of Poisson series expressions.

poissubst (*a*, *b*, *c*) Function

Substitutes *a* for *b* in *c*. *c* is a Poisson series.

(1) Where *B* is a variable *u*, *v*, *w*, *x*, *y*, or *z*, then *a* must be an expression linear in those variables (e.g., `6*u + 4*v`).

(2) Where *b* is other than those variables, then *a* must also be free of those variables, and furthermore, free of sines or cosines.

`poissubst (a, b, c, d, n)` is a special type of substitution which operates on *a* and *b* as in type (1) above, but where *d* is a Poisson series, expands `cos(d)` and `sin(d)` to order *n* so as to provide the result of substituting *a* + *d* for *b* in *c*. The idea is that *d* is an expansion in terms of a small parameter. For example, `poissubst (u, v, cos(v), %e, 3)` yields `cos(u)*(1 - %e^2/2) - sin(u)*(%e - %e^3/6)`.

poistimes (*a*, *b*) Function

Is functionally identical to `intopois (a*b)`.

poistrim () Function

is a reserved function name which (if the user has defined it) gets applied during Poisson multiplication. It is a predicate function of 6 arguments which are the coefficients of the *u*, *v*, ..., *z* in a term. Terms for which `poistrim` is `true` (for the coefficients of that term) are eliminated during multiplication.

printpois (*a*) Function

Prints a Poisson series in a readable format. In common with `outofpois`, it will convert *a* into a Poisson encoding first, if necessary.

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.

specint (*exp*(- *s***t*) * *expr*, *t*) Function

Compute the Laplace transform of *expr* with respect to the variable *t*. The integrand *expr* may contain special functions.

If `specint` cannot compute the integral, the return value may contain various Lisp symbols, including `other-defint-to-follow-negtest`, `other-lt-exponential-to-follow`, `product-of-y-with-nofract-indices`, etc.; this is a bug.

`demo(hypgeo)` displays several examples of Laplace transforms computed by `specint`.

Examples:

```
(%i1) assume (p > 0, a > 0);
(%o1)                      [p > 0, a > 0]
(%i2) specint (t^(1/2) * exp(-a*t/4) * exp(-p*t), t);
                             sqrt(%pi)
(%o2)                        ------------
                                    a 3/2
                             2 (p + -)
                                    4
(%i3) specint (t^(1/2) * bessel_j(1, 2 * a^(1/2) * t^(1/2))
              * exp(-p*t), t);
                                      - a/p
                           sqrt(a) %e
(%o3)                      ---------------
                                 2
                                p
```

hgfred (*a*, *b*, *t*) Function

Simplify the generalized hypergeometric function in terms of other, simpler, forms. *a* is a list of numerator parameters and *b* is a list of the denominator parameters.

If `hgfred` cannot simplify the hypergeometric function, it returns an expression of the form `%f[p,q]([a], [b], x)` where *p* is the number of elements in *a*, and *q* is the number of elements in *b*. This is the usual `pFq` generalized hypergeometric function.

```
(%i1) assume(not(equal(z,0)));
(%o1)                          [notequal(z, 0)]
(%i2) hgfred([v+1/2],[2*v+1],2*%i*z);

                     v/2                                  %i z
                    4    bessel_j(v, z) gamma(v + 1) %e
(%o2)               ----------------------------------------
                                      v
                                     z
(%i3) hgfred([1,1],[2],z);

                                   log(1 - z)
(%o3)                            - ----------
                                       z
(%i4) hgfred([a,a+1/2],[3/2],z^2);
```

```
                                      1 - 2 a          1 - 2 a
                                (z + 1)        - (1 - z)
(%o4)                           -------------------------------
                                         2 (1 - 2 a) z
```

It can be beneficial to load orthopoly too as the following example shows. Note that L is the generalized Laguerre polynomial.

```
(%i5) load(orthopoly)$
(%i6) hgfred([-2],[a],z);

                                        (a - 1)
                                     2 L       (z)
                                        2
(%o6)                                ------------
                                       a (a + 1)
(%i7) ev(%);

                                      2
                                     z        2 z
(%o7)                             --------- - --- + 1
                                  a (a + 1)    a
```

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 by 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 α. This is one area where we differ from Abramowitz and Stegun who use the modular angle for the elliptic functions. The following relationships are true:

$$m = k^2$$

and

$$k = \sin\alpha$$

The elliptic functions and integrals are primarily intended to support symbolic computation. Therefore, most of derivatives of the functions and integrals are known. However, if floating-point values are given, a floating-point result is returned.

Support for most of the other properties of elliptic functions and integrals other than derivatives has not yet been written.

Some examples of elliptic functions:

```
(%i1) jacobi_sn (u, m);
(%o1)                      jacobi_sn(u, m)
(%i2) jacobi_sn (u, 1);
(%o2)                          tanh(u)
(%i3) jacobi_sn (u, 0);
(%o3)                          sin(u)
(%i4) diff (jacobi_sn (u, m), u);
(%o4)             jacobi_cn(u, m) jacobi_dn(u, m)
(%i5) diff (jacobi_sn (u, m), m);
(%o5) jacobi_cn(u, m) jacobi_dn(u, m)

      elliptic_e(asin(jacobi_sn(u, m)), m)
 (u - ------------------------------------)/(2 m)
                     1 - m

            2
  jacobi_cn (u, m) jacobi_sn(u, m)
+ --------------------------------
              2 (1 - m)
```

Some examples of elliptic integrals:

```
(%i1) elliptic_f (phi, m);
(%o1)                   elliptic_f(phi, m)
```

```
(%i2) elliptic_f (phi, 0);
(%o2)                           phi
(%i3) elliptic_f (phi, 1);
                                phi   %pi
(%o3)                  log(tan(--- + ---))
                                 2     4
(%i4) elliptic_e (phi, 1);
(%o4)                       sin(phi)
(%i5) elliptic_e (phi, 0);
(%o5)                           phi
(%i6) elliptic_kc (1/2);
                                     1
(%o6)                    elliptic_kc(-)
                                     2
(%i7) makegamma (%);
                                 2 1
                              gamma (-)
                                   4
(%o7)                        -----------
                             4 sqrt(%pi)
(%i8) diff (elliptic_f (phi, m), phi);
                                  1
(%o8)                  ---------------------
                                    2
                       sqrt(1 - m sin (phi))
(%i9) diff (elliptic_f (phi, m), m);
       elliptic_e(phi, m) - (1 - m) elliptic_f(phi, m)
(%o9) (-----------------------------------------------
                              m

                                  cos(phi) sin(phi)
                               - ---------------------)/(2 (1 - m))
                                                2
                                 sqrt(1 - m sin (phi))
```

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 $dn(u, m)$.

jacobi_ns (u, m) Function
The Jacobian elliptic function $ns(u, m) = 1/sn(u, m)$.

jacobi_sc (u, m) Function
The Jacobian elliptic function $sc(u, m) = sn(u, m)/cn(u, m)$.

jacobi_sd (u, m) Function
The Jacobian elliptic function $sd(u, m) = sn(u, m)/dn(u, m)$.

jacobi_nc (u, m) Function
The Jacobian elliptic function $nc(u, m) = 1/cn(u, m)$.

jacobi_cs (u, m) Function
The Jacobian elliptic function $cs(u, m) = cn(u, m)/sn(u, m)$.

jacobi_cd (u, m) Function
The Jacobian elliptic function $cd(u, m) = cn(u, m)/dn(u, m)$.

jacobi_nd (u, m) Function
The Jacobian elliptic function $nc(u, m) = 1/cn(u, m)$.

jacobi_ds (u, m) Function
The Jacobian elliptic function $ds(u, m) = dn(u, m)/sn(u, m)$.

jacobi_dc (u, m) Function
The Jacobian elliptic function $dc(u, m) = dn(u, m)/cn(u, m)$.

inverse_jacobi_sn (u, m) Function
The inverse of the Jacobian elliptic function $sn(u, m)$.

inverse_jacobi_cn (u, m) Function
The inverse of the Jacobian elliptic function $cn(u, m)$.

inverse_jacobi_dn (u, m) Function
The inverse of the Jacobian elliptic function $dn(u, m)$.

inverse_jacobi_ns (u, m) Function
The inverse of the Jacobian elliptic function $ns(u, m)$.

inverse_jacobi_sc (u, m) Function
The inverse of the Jacobian elliptic function $sc(u, m)$.

inverse_jacobi_sd (u, m) Function
The inverse of the Jacobian elliptic function $sd(u, m)$.

inverse_jacobi_nc (*u*, *m*) Function
The inverse of the Jacobian elliptic function $nc(u, m)$.

inverse_jacobi_cs (*u*, *m*) Function
The inverse of the Jacobian elliptic function $cs(u, m)$.

inverse_jacobi_cd (*u*, *m*) Function
The inverse of the Jacobian elliptic function $cd(u, m)$.

inverse_jacobi_nd (*u*, *m*) Function
The inverse of the Jacobian elliptic function $nc(u, m)$.

inverse_jacobi_ds (*u*, *m*) Function
The inverse of the Jacobian elliptic function $ds(u, m)$.

inverse_jacobi_dc (*u*, *m*) Function
The inverse of the Jacobian elliptic function $dc(u, m)$.

17.3 Functions and Variables for Elliptic Integrals

elliptic_f (*phi*, *m*) Function
The incomplete elliptic integral of the first kind, defined as

$$\int_0^{\phi} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}$$

See also [elliptic_e], page 208 and [elliptic_kc], page 209.

elliptic_e (*phi*, *m*) Function
The incomplete elliptic integral of the second kind, defined as

$$\int_0^{\phi} \sqrt{1 - m\sin^2\theta} d\theta$$

See also [elliptic_e], page 208 and [elliptic_ec], page 209.

elliptic_eu (*u*, *m*) Function
The incomplete elliptic integral of the second kind, defined as

$$\int_0^{u} \mathrm{dn}(v, m) dv = \int_0^{\tau} \sqrt{\frac{1 - mt^2}{1 - t^2}} dt$$

where $\tau = \mathrm{sn}(u, m)$
This is related to *elliptic*$_e$ by

$$E(u, m) = E(\phi, m)$$

where $\phi = \sin^{-1}\mathrm{sn}(u, m)$ See also [elliptic_e], page 208.

elliptic_pi (*n*, *phi*, *m*) Function

The incomplete elliptic integral of the third kind, defined as

$$\int_0^{\phi} \frac{d\theta}{(1 - n\sin^2\theta)\sqrt{1 - m\sin^2\theta}}$$

Only the derivative with respect to *phi* is known by Maxima.

elliptic_kc (*m*) Function

The complete elliptic integral of the first kind, defined as

$$\int_0^{\frac{\pi}{2}} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}$$

For certain values of m, the value of the integral is known in terms of *Gamma* functions. Use `makegamma` to evaluate them.

elliptic_ec (*m*) Function

The complete elliptic integral of the second kind, defined as

$$\int_0^{\frac{\pi}{2}} \sqrt{1 - m\sin^2\theta}\, d\theta$$

For certain values of m, the value of the integral is known in terms of *Gamma* functions. Use `makegamma` to evaluate them.

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).

`lhospitallim` is the maximum number of times L'Hospital's rule is used in `limit`. This prevents infinite looping in cases like `limit (cot(x)/csc(x), x, 0)`.

`tlimswitch` when true will allow the `limit` command to use Taylor series expansion when necessary.

`limsubst` prevents `limit` from attempting substitutions on unknown forms. This is to avoid bugs like `limit (f(n)/f(n+1), n, inf)` giving 1. Setting `limsubst` to `true` will allow such substitutions.

`limit` with one argument is often called upon to simplify constant expressions, for example, `limit (inf-1)`.

`example (limit)` displays some examples.

For the method see Wang, P., "Evaluation of Definite Integrals by Symbolic Manipulation", Ph.D. thesis, MAC TR-92, October 1971.

limsubst Option variable

default value: `false` - prevents `limit` from attempting substitutions on unknown forms. This is to avoid bugs like `limit (f(n)/f(n+1), n, inf)` giving 1. Setting `limsubst` to `true` will allow such substitutions.

tlimit (*expr, x, val, dir*) Function
tlimit (*expr, x, val*) Function
tlimit (*expr*) Function

Take the limit of the Taylor series expansion of `expr` in `x` at `val` from direction `dir`.

tlimswitch Option variable

Default value: `true`

When `tlimswitch` is `true`, the `limit` command will use a Taylor series expansion if the limit of the input expression cannot be computed directly. This allows evaluation of limits such as `limit(x/(x-1)-1/log(x),x,1,plus)`. When `tlimswitch` is `false` and the limit of input expression cannot be computed directly, `limit` will return an unevaluated limit expression.

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
(%o2)                  y(x) %e     (-- (z(x)))
                                     dx
(%i3) a1: antid (expr, x, z(x));
                           z(x)      z(x)  d
(%o3)           [y(x) %e    , - %e     (-- (y(x)))]
                                       dx
(%i4) a2: antidiff (expr, x, z(x));
                                /
                        z(x)    [   z(x)  d
(%o4)          y(x) %e     - I %e     (-- (y(x))) dx
                                ]         dx
                                /
(%i5) a2 - (first (a1) + 'integrate (second (a1), x));
(%o5)                           0
(%i6) antid (expr, x, y(x));
                                 z(x)  d
(%o6)              [0, y(x) %e     (-- (z(x)))]
                                    dx
(%i7) antidiff (expr, x, y(x));
                     /
                     [        z(x)  d
(%o7)                I y(x) %e     (-- (z(x))) dx
                     ]              dx
                     /
```

antidiff (*expr, x, u*(*x*)) 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);
                                z(x)  d
(%o2)                  y(x) %e     (-- (z(x)))
                                     dx
(%i3) a1: antid (expr, x, z(x));
                         z(x)      z(x)  d
(%o3)           [y(x) %e    , - %e     (-- (y(x)))]
                                       dx
(%i4) a2: antidiff (expr, x, z(x));
                              /
                        z(x)  [   z(x)  d
(%o4)          y(x) %e     - I %e     (-- (y(x))) dx
                              ]         dx
                              /
(%i5) a2 - (first (a1) + 'integrate (second (a1), x));
(%o5)                                0
(%i6) antid (expr, x, y(x));
                               z(x)  d
(%o6)              [0, y(x) %e     (-- (z(x)))]
                                  dx
(%i7) antidiff (expr, x, y(x));
                    /
                    [        z(x)  d
(%o7)               I y(x) %e     (-- (z(x))) dx
                    ]              dx
                    /
```

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_n*, *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);
(%o2)                       @2 + 1
(%i3) printprops (all, atvalue);
                                !
                 d              !
                --- (f(@1, @2))!       = @2 + 1
                d@1             !
                                !@1 = 0

                                     2
                          f(0, 1) = a

(%o3)                        done
(%i4) diff (4*f(x,y)^2 - u(x,y)^2, x);
                  d                                d
(%o4)  8 f(x, y) (-- (f(x, y))) - 2 u(x, y) (-- (u(x, y)))
                  dx                               dx
(%i5) at (%, [x = 0, y = 1]);
                                              !
             2              d                 !
(%o5)     16 a  - 2 u(0, 1) (-- (u(x, y)))!             )
                             dx               !
                                              !x = 0, y = 1
```

cartan - 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 product) and `|` (contraction of a form with a vector.) Type `demo (tensor)` to see a brief description of these commands along with examples.

`cartan` was implemented by F.B. Estabrook and H.D. Wahlquist.

del (*x*) Function

`del (x)` represents the differential of the variable *x*.

`diff` returns an expression containing `del` if an independent variable is not specified. In this case, the return value is the so-called "total differential".

Examples:

```
(%i1) diff (log (x));
                                del(x)
(%o1)                           ------
                                  x
(%i2) diff (exp (x*y));
                         x y                x y
(%o2)              x %e    del(y) + y %e    del(x)
(%i3) diff (x*y*z);
(%o3)          x y del(z) + x z del(y) + y z del(x)
```

delta (*t*) Function

The Dirac Delta function.

Currently only `laplace` knows about the `delta` function.

Example:

```
(%i1) laplace (delta (t - a) * sin(b*t), t, s);
Is  a  positive, negative, or zero?

p;
                                        - a s
(%o1)                     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, `diff (f, x)` yields zero. If

`depends (f, x)` is declared, `diff (f, x)` yields a symbolic derivative (that is, a `diff` noun).

Each argument *f_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*.

`diff` recognizes indirect dependencies established by `depends` and applies the chain rule in these cases.

`remove (`*f*`, dependency)` removes all dependencies declared for *f*.

`depends` returns a list of the dependencies established. The dependencies are appended to the global variable `dependencies`. `depends` evaluates its arguments.

`diff` is the only Maxima command which recognizes dependencies established by `depends`. Other functions (`integrate`, `laplace`, etc.) only recognize dependencies explicitly represented by their arguments. For example, `integrate` does not recognize the dependence of `f` on `x` unless explicitly represented as `integrate (f(x), x)`.

```
(%i1) depends ([f, g], x);
(%o1)                        [f(x), g(x)]
(%i2) depends ([r, s], [u, v, w]);
(%o2)                [r(u, v, w), s(u, v, w)]
(%i3) depends (u, t);
(%o3)                           [u(t)]
(%i4) dependencies;
(%o4)      [f(x), g(x), r(u, v, w), s(u, v, w), u(t)]
(%i5) diff (r.s, u);
                           dr           ds
(%o5)                      -- . s + r . --
                           du           du
(%i6) diff (r.s, t);
                       dr du           ds du
(%o6)                  -- -- . s + r . -- --
                       du dt           du dt
(%i7) remove (r, dependency);
(%o7)                             done
(%i8) diff (r.s, t);
                                  ds du
(%o8)                          r . -- --
                                  du dt
```

derivabbrev Option variable

Default value: `false`

When `derivabbrev` is `true`, symbolic derivatives (that is, `diff` nouns) are displayed as subscripts. Otherwise, derivatives are displayed in the Leibniz notation `dy/dx`.

derivdegree (*expr*, *y*, *x*) Function

Returns the highest degree of the derivative of the dependent variable *y* with respect to the independent variable *x* occuring in *expr*.

Example:

```
(%i1) 'diff (y, x, 2) + 'diff (y, z, 3) + 'diff (y, x) * x^2;
                          3     2
                         d y   d y    2 dy
(%o1)                    --- + --- + x  --
                           3     2      dx
                         dz    dx
(%i2) derivdegree (%, y, x);
(%o2)                             2
```

derivlist (*var_1, ..., var_k*) Function

Causes only differentiations with respect to the indicated variables, within the `ev` command.

derivsubst Option variable

Default value: `false`

When `derivsubst` is `true`, a non-syntactic substitution such as `subst (x, 'diff (y, t), 'diff (y, t, 2))` yields `'diff (x, t)`.

diff (*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 its variables times the differential `del` of each variable. No further simplification of `del` is offered.

The noun form of `diff` is required in some contexts, such as stating a differential equation. In these cases, `diff` may be quoted (as `'diff`) to yield the noun form instead of carrying out the differentiation.

When `derivabbrev` is `true`, derivatives are displayed as subscripts. Otherwise, derivatives are displayed in the Leibniz notation, `dy/dx`.

Examples:

```
(%i1) diff (exp (f(x)), x, 2);
                        2
                  f(x)  d                f(x)  d          2
(%o1)           %e     (--- (f(x))) + %e     (-- (f(x)))
                         2                    dx
                        dx
```

```
(%i2) derivabbrev: true$
(%i3) 'integrate (f(x, y), y, g(x), h(x));
                          h(x)
                         /
                         [
(%o3)                    I     f(x, y) dy
                         ]
                         /
                          g(x)
(%i4) diff (%, x);
       h(x)
      /
      [
(%o4) I     f(x, y)  dy + f(x, h(x)) h(x)  - f(x, g(x)) g(x)
      ]             x                      x                  x
      /
       g(x)
```

For the tensor package, the following modifications have been incorporated:

(1) The derivatives of any indexed objects in *expr* will have the variables *x_i* appended as additional arguments. Then all the derivative indices will be sorted.

(2) The *x_i* may be integers from 1 up to the value of the variable `dimension` [default value: 4]. This will cause the differentiation to be carried out with respect to the *x_i*'th member of the list `coordinates` which should be set to a list of the names of the coordinates, e.g., `[x, y, z, t]`. If `coordinates` is bound to an atomic variable, then that variable subscripted by *x_i* will be used for the variable of differentiation. This permits an array of coordinate names or subscripted names like `X[1]`, `X[2]`, ... to be used. If `coordinates` has not been assigned a value, then the variables will be treated as in (1) above.

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*.

`load ("ctensor")` loads this function.

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 `~`.

Symbolic derivatives (that is, `diff` nouns) in the return value of express may be evaluated by including `diff` in the `ev` function call or command line. In this context, `diff` acts as an `evfun`.

`load ("vect")` loads this function.

Examples:

```
(%i1) load ("vect")$
(%i2) grad (x^2 + y^2 + z^2);
                              2    2    2
(%o2)                    grad (z  + y  + x )
(%i3) express (%);
      d    2    2    2   d    2    2    2   d    2    2    2
(%o3) [-- (z  + y  + x ), -- (z  + y  + x ), -- (z  + y  + x )]
      dx                 dy                 dz
(%i4) ev (%, diff);
(%o4)                      [2 x, 2 y, 2 z]
(%i5) div ([x^2, y^2, z^2]);
                               2   2   2
(%o5)                     div [x , y , z ]
(%i6) express (%);
                   d    2    d    2    d    2
(%o6)              -- (z ) + -- (y ) + -- (x )
                   dz        dy        dx
(%i7) ev (%, diff);
(%o7)                      2 z + 2 y + 2 x
(%i8) curl ([x^2, y^2, z^2]);
                                 2   2   2
(%o8)                     curl [x , y , z ]
(%i9) express (%);
      d    2    d    2   d    2    d    2   d    2   d    2
(%o9) [-- (z ) - -- (y ), -- (x ) - -- (z ), -- (y ) - -- (x )]
      dy        dz       dz        dx       dx        dy
(%i10) ev (%, diff);
(%o10)                        [0, 0, 0]
(%i11) laplacian (x^2 * y^2 * z^2);
                                  2  2  2
(%o11)                 laplacian (x  y  z )
(%i12) express (%);
        2                2                2
       d     2  2  2    d     2  2  2    d     2  2  2
(%o12)  --- (x  y  z ) + --- (x  y  z ) + --- (x  y  z )
          2                2                2
       dz               dy               dx
(%i13) ev (%, diff);
                     2  2      2  2      2  2
(%o13)              2 y  z  + 2 x  z  + 2 x  y
(%i14) [a, b, c] ~ [x, y, z];
(%o14)                  [a, b, c] ~ [x, y, z]
(%i15) express (%);
(%o15)           [b z - c y, c x - a z, a y - b x]
```

gradef (*f*(*x_1*, ..., *x_n*), *g_1*, ..., *g_m*) Function
gradef (*a*, *x*, *expr*) Function

Defines the partial derivatives (i.e., the components of the gradient) of the function *f* or variable *a*.

`gradef (`*f*`(`*x_1*`, ..., `*x_n*`), `*g_1*`, ..., `*g_m*`)` defines `df/dx_`*i* as *g_i*, where *g_i* is an expression; *g_i* may be a function call, but not the name of a function. The number of partial derivatives *m* may be less than the number of arguments *n*, in which case derivatives are defined with respect to *x_1* through *x_m* only.

`gradef (`*a*`, `*x*`, `*expr*`)` defines the derivative of variable *a* with respect to *x* as *expr*. This also establishes the dependence of *a* on *x* (via `depends (`*a*`, `*x*`)`).

The first argument *f*`(`*x_1*`, ..., `*x_n*`)` or *a* is quoted, but the remaining arguments *g_1*, ..., *g_m* are evaluated. `gradef` returns the function or variable for which the partial derivatives are defined.

`gradef` can redefine the derivatives of Maxima's built-in functions. For example, `gradef (sin(x), sqrt (1 - sin(x)^2))` redefines the derivative of `sin`.

`gradef` cannot define partial derivatives for a subscripted function.

`printprops ([`*f_1*`, ..., `*f_n*`], gradef)` displays the partial derivatives of the functions *f_1*, ..., *f_n*, as defined by `gradef`.

`printprops ([`*a_n*`, ..., `*a_n*`], atomgrad)` displays the partial derivatives of the variables *a_n*, ..., *a_n*, as defined by `gradef`.

`gradefs` is the list of the functions for which partial derivatives have been defined by `gradef`. `gradefs` does not include any variables for which partial derivatives have been defined by `gradef`.

Gradients are needed when, for example, a function is not known explicitly but its first derivatives are and it is desired to obtain higher order derivatives.

gradefs System variable

Default value: `[]`

`gradefs` is the list of the functions for which partial derivatives have been defined by `gradef`. `gradefs` does not include any variables for which partial derivatives have been defined by `gradef`.

laplace (*expr*, *t*, *s*) Function

Attempts to compute the Laplace transform of *expr* with respect to the variable *t* and transform parameter *s*. If `laplace` cannot find a solution, a noun `'laplace` is returned.

`laplace` recognizes in *expr* the functions `delta`, `exp`, `log`, `sin`, `cos`, `sinh`, `cosh`, and `erf`, as well as `derivative`, `integrate`, `sum`, and `ilt`. If some other functions are present, `laplace` may not be able to compute the transform.

expr may also be a linear, constant coefficient differential equation in which case `atvalue` of the dependent variable is used. The required atvalue may be supplied either before or after the transform is computed. Since the initial conditions must be specified at zero, if one has boundary conditions imposed elsewhere he can impose these on the general solution and eliminate the constants by solving the general solution for them and substituting their values back.

`laplace` recognizes convolution integrals of the form `integrate (f(x) * g(t - x), x, 0, t)`; other kinds of convolutions are not recognized.

Functional relations must be explicitly represented in *expr*; implicit relations, established by `depends`, are not recognized. That is, if f depends on x and y, `f (x, y)` must appear in *expr*.

See also `ilt`, the inverse Laplace transform.

Examples:

```
(%i1) laplace (exp (2*t + a) * sin(t) * t, t, s);
                              a
                             %e  (2 s - 4)
(%o1)                       ---------------
                              2           2
                             (s  - 4 s + 5)
(%i2) laplace ('diff (f (x), x), x, s);
(%o2)             s laplace(f(x), x, s) - f(0)
(%i3) diff (diff (delta (t), t), t);
                             2
                            d
(%o3)                       --- (delta(t))
                              2
                            dt
(%i4) laplace (%, t, s);
                                !
                d               !         2
(%o4)         - -- (delta(t))!      + s  - delta(0) s
                dt              !
                                !t = 0
```

20 Integration

20.1 Introduction to Integration

Maxima has several routines for handling integration. The `integrate` function makes use of most of them. There is also the `antid` package, which handles an unspecified function (and its derivatives, of course). For numerical uses, there is a set of adaptive integrators from QUADPACK, named `quad_qag`, `quad_qags`, etc., which are described under the heading `QUADPACK`. Hypergeometric functions are being worked on, see `specint` for details. Generally speaking, Maxima only handles integrals which are integrable in terms of the "elementary functions" (rational functions, trigonometrics, logs, exponentials, radicals, etc.) and a few extensions (error function, dilogarithm). It does not handle integrals in terms of unknown functions such as `g(x)` and `h(x)`.

20.2 Functions and Variables for Integration

changevar (*expr, f*(*x*,*y*), *y*, *x*) Function

Makes the change of variable given by *f(x,y)* = 0 in all integrals occurring in *expr* with integration with respect to *x*. The new variable is *y*.

```
(%i1) assume(a > 0)$
(%i2) 'integrate (%e**sqrt(a*y), y, 0, 4);
                      4
                     /
                     [    sqrt(a) sqrt(y)
(%o2)                I  %e                dy
                     ]
                     /
                      0
(%i3) changevar (%, y-z^2/a, z, y);
                      0
                     /
                     [                abs(z)
                   2 I            z %e       dz
                     ]
                     /
                      - 2 sqrt(a)
(%o3)            - ----------------------------
                                 a
```

An expression containing a noun form, such as the instances of `'integrate` above, may be evaluated by `ev` with the `nouns` flag. For example, the expression returned by `changevar` above may be evaluated by `ev (%o3, nouns)`.

`changevar` may also be used to changes in the indices of a sum or product. However, it must be realized that when a change is made in a sum or product, this change must be a shift, i.e., `i = j+ ...`, not a higher degree function. E.g.,

```
(%i4) sum (a[i]*x^(i-2), i, 0, inf);
                          inf
                          ====
                          \          i - 2
(%o4)                      >    a  x
                          /      i
                          ====
                          i = 0
(%i5) changevar (%, i-2-n, n, i);
                         inf
                         ====
                         \                n
(%o5)                     >      a      x
                         /        n + 2
                         ====
                         n = - 2
```

dblint (*f*, *r*, *s*, *a*, *b*) 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

```
/b /s(x)
|  |
|  |    f(x,y) dy dx
|  |
/a /r(x)
```

The function *f* must be a translated or compiled function of two variables, and *r* and *s* must each be a translated or compiled function of one variable, while *a* and *b* must be floating point numbers. The routine has two global variables which determine the number of divisions of the x and y intervals: `dblint_x` and `dblint_y`, both of which are initially 10, and can be changed independently to other integer values (there are `2*dblint_x+1` points computed in the x direction, and `2*dblint_y+1` in the y direction). The routine subdivides the X axis and then for each value of X it first computes *r*`(x)` and *s*`(x)`; then the Y axis between *r*`(x)` and *s*`(x)` is subdivided and the integral along the Y axis is performed using Simpson's rule; then the integral along the X axis is done using Simpson's rule with the function values being the Y-integrals. This procedure may be numerically unstable for a great variety of reasons, but is reasonably fast: avoid using it on highly oscillatory functions and functions with singularities (poles or branch points in the region). The Y integrals depend on how far apart *r*`(x)` and *s*`(x)` are, so if the distance *s*`(x) - r(x)` varies rapidly with X, there may be substantial errors arising from truncation with different step-sizes in the various Y integrals. One can increase `dblint_x` and `dblint_y` in an effort to improve the coverage of the region, at the expense of computation time. The function values are not saved, so if the function is very time-consuming, you will have to wait for re-computation if you change anything (sorry). It is required that the functions *f*, *r*, and *s* be either translated or compiled prior to calling `dblint`. This will result in orders of magnitude speed improvement over interpreted code in many cases!

`demo (dblint)` executes a demonstration of `dblint` applied to an example problem.

defint (*expr*, *x*, *a*, *b*) Function

Attempts to compute a definite integral. `defint` is called by `integrate` when limits of integration are specified, i.e., when `integrate` is called as `integrate (`*expr*`, `*x*`, `*a*`, `*b*`)`. Thus from the user's point of view, it is sufficient to call `integrate`.

`defint` returns a symbolic expression, either the computed integral or the noun form of the integral. See `quad_qag` and related functions for numerical approximation of definite integrals.

erf (*x*) Function

Represents the error function, whose derivative is: `2*exp(-x^2)/sqrt(%pi)`.

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*, *t*, *s*) Function

Computes the inverse Laplace transform of *expr* with respect to *t* and parameter *s*. *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 (%, t, s);
                                 a laplace(f(t), t, s)   2
(%o2)  b laplace(f(t), t, s) + ---------------------- = --
                                       2    2           3
                                      s  - a           s
(%i3) linsolve ([%], ['laplace(f(t), t, s)]);
                                        2      2
                                     2 s  - 2 a
(%o3)     [laplace(f(t), t, s) = --------------------]
                                    5         2     3
                                 b s  + (a - a  b) s
(%i4) ilt (rhs (first (%)), s, t);
Is  a b (a b - 1)  positive, negative, or zero?

pos;
```

```
                      sqrt(a b (a b - 1)) t
                  2 cosh(--------------------)       2
                                 b              a t
(%o4) - ---------------------------- + -------
                3  2      2              a b - 1
               a  b  - 2 a  b + a

                                                    2
                                               + ----------------
                                                 3  2      2
                                                a  b  - 2 a  b + a
```

integrate (*expr*, *x*) Function
integrate (*expr*, *x*, *a*, *b*) Function

Attempts to symbolically compute the integral of *expr* with respect to *x*. `integrate (expr, x)` is an indefinite integral, while `integrate (expr, x, a, b)` is a definite integral, with limits of integration *a* and *b*. The limits should not contain *x*, although `integrate` does not enforce this restriction. *a* need not be less than *b*. If *b* is equal to *a*, `integrate` returns zero.

See `quad_qag` and related functions for numerical approximation of definite integrals. See `residue` for computation of residues (complex integration). See `antid` for an alternative means of computing indefinite integrals.

The integral (an expression free of `integrate`) is returned if `integrate` succeeds. Otherwise the return value is the noun form of the integral (the quoted operator `'integrate`) or an expression containing one or more noun forms. The noun form of `integrate` is displayed with an integral sign.

In some circumstances it is useful to construct a noun form by hand, by quoting `integrate` with a single quote, e.g., `'integrate (expr, x)`. For example, the integral may depend on some parameters which are not yet computed. The noun may be applied to its arguments by `ev (i, nouns)` where *i* is the noun form of interest.

`integrate` handles definite integrals separately from indefinite, and employs a range of heuristics to handle each case. Special cases of definite integrals include limits of integration equal to zero or infinity (`inf` or `minf`), trigonometric functions with limits of integration equal to zero and `%pi` or `2 %pi`, rational functions, integrals related to the definitions of the `beta` and `psi` functions, and some logarithmic and trigonometric integrals. Processing rational functions may include computation of residues. If an applicable special case is not found, an attempt will be made to compute the indefinite integral and evaluate it at the limits of integration. This may include taking a limit as a limit of integration goes to infinity or negative infinity; see also `ldefint`.

Special cases of indefinite integrals include trigonometric functions, exponential and logarithmic functions, and rational functions. `integrate` may also make use of a short table of elementary integrals.

`integrate` may carry out a change of variable if the integrand has the form `f(g(x)) * diff(g(x), x)`. `integrate` attempts to find a subexpression `g(x)` such that the derivative of `g(x)` divides the integrand. This search may make use of derivatives defined by the `gradef` function. See also `changevar` and `antid`.

If none of the preceding heuristics find the indefinite integral, the Risch algorithm is executed. The flag `risch` may be set as an `evflag`, in a call to `ev` or on the command line, e.g., `ev (integrate (`*expr*`, `*x*`), risch)` or `integrate (`*expr*`, `*x*`), risch`. If `risch` is present, `integrate` calls the `risch` function without attempting heuristics first. See also `risch`.

`integrate` works only with functional relations represented explicitly with the `f(x)` notation. `integrate` does not respect implicit dependencies established by the `depends` function.

`integrate` may need to know some property of a parameter in the integrand. `integrate` will first consult the `assume` database, and, if the variable of interest is not there, `integrate` will ask the user. Depending on the question, suitable responses are `yes;` or `no;`, or `pos;`, `zero;`, or `neg;`.

`integrate` is not, by default, declared to be linear. See `declare` and `linear`.

`integrate` attempts integration by parts only in a few special cases.

Examples:

- Elementary indefinite and definite integrals.

```
(%i1) integrate (sin(x)^3, x);
                              3
                         cos (x)
(%o1)                    ------- - cos(x)
                            3
(%i2) integrate (x/ sqrt (b^2 - x^2), x);
                                    2    2
(%o2)                     - sqrt(b  - x )
(%i3) integrate (cos(x)^2 * exp(x), x, 0, %pi);
                                %pi
                              3 %e      3
(%o3)                         ------- - -
                                 5      5
(%i4) integrate (x^2 * exp(-x^2), x, minf, inf);
                               sqrt(%pi)
(%o4)                          ---------
                                   2
```

- Use of `assume` and interactive query.

```
(%i1) assume (a > 1)$
(%i2) integrate (x**a/(x+1)**(5/2), x, 0, inf);
    2 a + 2
Is  -------  an integer?
       5

no;
Is  2 a - 3  positive, negative, or zero?

neg;
                                      3
(%o2)                  beta(a + 1, - - a)
                                      2
```

- Change of variable. There are two changes of variable in this example: one using a derivative established by `gradef`, and one using the derivation `diff(r(x))` of an unspecified function `r(x)`.

```
(%i3) gradef (q(x), sin(x**2));
(%o3)                          q(x)
(%i4) diff (log (q (r (x))), x);
                        d               2
                       (-- (r(x))) sin(r (x))
                        dx
(%o4)                  ----------------------
                              q(r(x))
(%i5) integrate (%, x);
(%o5)                       log(q(r(x)))
```

- Return value contains the `'integrate` noun form. In this example, Maxima can extract one factor of the denominator of a rational function, but cannot factor the remainder or otherwise find its integral. `grind` shows the noun form `'integrate` in the result. See also `integrate_use_rootsof` for more on integrals of rational functions.

```
(%i1) expand ((x-4) * (x^3+2*x+1));
                     4      3      2
(%o1)               x  - 4 x  + 2 x  - 7 x - 4
(%i2) integrate (1/%, x);
                                /  2
                                [ x  + 4 x + 18
                                I ------------- dx
                                ]  3
                  log(x - 4)    / x  + 2 x + 1
(%o2)             ---------- - ------------------
                      73               73
(%i3) grind (%);
log(x-4)/73-('integrate((x^2+4*x+18)/(x^3+2*x+1),x))/73$
```

- Defining a function in terms of an integral. The body of a function is not evaluated when the function is defined. Thus the body of `f_1` in this example contains the noun form of `integrate`. The quote-quote operator `''` causes the integral to be evaluated, and the result becomes the body of `f_2`.

```
(%i1) f_1 (a) := integrate (x^3, x, 1, a);
                                       3
(%o1)           f_1(a) := integrate(x , x, 1, a)
(%i2) ev (f_1 (7), nouns);
(%o2)                          600
(%i3) /* Note parentheses around integrate(...) here */
      f_2 (a) := ''(integrate (x^3, x, 1, a));
                                     4
                                    a    1
(%o3)                   f_2(a) := -- - -
                                    4    4
(%i4) f_2 (7);
(%o4)                          600
```

integration_constant System variable

Default value: `%c`

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

`integration_constant` may be assigned any symbol.

Examples:

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

integration_constant_counter System variable

Default value: 0

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

`integration_constant_counter` is incremented before constructing the next integration constant.

Examples:

```
(%i1) integrate (x^2 = 1, x);
                              3
                             x
(%o1)                        -- = x + %c1
                             3
(%i2) integrate (x^2 = 1, x);
                              3
                             x
(%o2)                        -- = x + %c2
                             3
(%i3) integrate (x^2 = 1, x);
                              3
                             x
(%o3)                        -- = x + %c3
                             3
(%i4) reset (integration_constant_counter);
(%o4)             [integration_constant_counter]
(%i5) integrate (x^2 = 1, x);
                              3
```

```
                                x
(%o5)                           -- = x + %c1
                                3
```

integrate_use_rootsof Option variable

Default value: `false`

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

For example, with `integrate_use_rootsof` set to `false`, `integrate` returns an unsolved integral of a rational function in noun form:

```
(%i1) integrate_use_rootsof: false$
(%i2) integrate (1/(1+x+x^5), x);
       /  2
       [ x  - 4 x + 5
       I ------------ dx                              2 x + 1
       ] 3    2                 2             5 atan(-------)
       / x  - x  + 1       log(x  + x + 1)          sqrt(3)
(%o2)  ----------------- - --------------- + ---------------
                7                 14             7 sqrt(3)
```

Now we set the flag to be true and the unsolved part of the integral will be expressed as a summation over the roots of the denominator of the rational function:

```
(%i3) integrate_use_rootsof: true$
(%i4) integrate (1/(1+x+x^5), x);
      ====        2
      \       (%r4  - 4 %r4 + 5) log(x - %r4)
       >      ------------------------------
      /                    2
      ====            3 %r4  - 2 %r4
                      3    2
      %r4 in rootsof(x  - x  + 1)
(%o4) ------------------------------------------------------------
               7

                                                    2 x + 1
                                          2      5 atan(-------)
                                       log(x  + x + 1)   sqrt(3)
                                     - --------------- + ---------
                                            14           7 sqrt(3)
```

Alternatively the user may compute the roots of the denominator separately, and then express the integrand in terms of these roots, e.g., `1/((x - a)*(x - b)*(x - c))` or `1/((x^2 - (a+b)*x + a*b)*(x - c))` if the denominator is a cubic polynomial. Sometimes this will help Maxima obtain a more useful result.

ldefint (*expr*, *x*, *a*, *b*) Function

Attempts to compute the definite integral of *expr* by using `limit` to evaluate the indefinite integral of *expr* with respect to *x* at the upper limit *b* and at the lower

limit *a*. If it fails to compute the definite integral, `ldefint` returns an expression containing limits as noun forms.

`ldefint` is not called from `integrate`, so executing `ldefint (`*expr*`, `*x*`, `*a*`, `*b*`)` may yield a different result than `integrate (`*expr*`, `*x*`, `*a*`, `*b*`)`. `ldefint` always uses the same method to evaluate the definite integral, while `integrate` may employ various heuristics and may recognize some special cases.

potential (*givengradient*) Function

The calculation makes use of the global variable `potentialzeroloc[0]` which must be `nonlist` or of the form

```
[indeterminatej=expressionj, indeterminatek=expressionk, ...]
```

the former being equivalent to the nonlist expression for all right-hand sides in the latter. The indicated right-hand sides are used as the lower limit of integration. The success of the integrations may depend upon their values and order. `potentialzeroloc` is initially set to 0.

residue (*expr*, *z*, *z_0*) Function

Computes the residue in the complex plane of the expression *expr* when the variable *z* assumes the value *z_0*. The residue is the coefficient of `(`*z* `- `*z_0*`)^(-1)` in the Laurent series for *expr*.

```
(%i1) residue (s/(s**2+a**2), s, a*%i);
                                1
(%o1)                           -
                                2
(%i2) residue (sin(a*x)/x**4, x, 0);
                                 3
                                a
                              - --
(%o2)                           6
```

risch (*expr*, *x*) Function

Integrates *expr* with respect to *x* using the transcendental case of the Risch algorithm. (The algebraic case of the Risch algorithm has not been implemented.) This currently handles the cases of nested exponentials and logarithms which the main part of `integrate` can't do. `integrate` will automatically apply `risch` if given these cases.

`erfflag`, if `false`, prevents `risch` from introducing the `erf` function in the answer if there were none in the integrand to begin with.

```
(%i1) risch (x^2*erf(x), x);
                                                          2
             3                          2                - x
        %pi x  erf(x) + (sqrt(%pi) x  + sqrt(%pi)) %e
(%o1)   ---------------------------------------------------
                              3 %pi
(%i2) diff(%, x), ratsimp;
                             2
(%o2)                       x  erf(x)
```

tldefint (*expr, x, a, b*) 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.

`quad_qawo`
Integration of $cos(omegax)f(x)$ or $sin(omegax)f(x)$ over a finite interval, where *omega* is a constant. The rule evaluation component is based on the

[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`
[7] R. Piessens, E. de Doncker-Kapenga, C.W. Uberhuber, and D.K. Kahaner. *QUADPACK: A Subroutine Package for Automatic Integration.* Berlin: Springer-Verlag, 1983, ISBN 0387125531.

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 $(x-a)^{a}lpha(b-x)^{b}etav(x)$ 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*]) Function
quad_qag (*f*, x, a, b, key, [*epsrel, epsabs, limit*]) 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-10.

epsabs Desired absolute error of approximation. Default is 0.

limit Size of internal work array. (*limit* - *limlst*)/2 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)    [.4444444444492108, 3.1700968502883E-9, 961, 0]
(%i2) integrate (x^(1/2)*log(1/x), x, 0, 1);
                                4
(%o2)                           -
                                9
```

quad_qags (*f*(*x*), *x*, *a*, *b*, [*epsrel*, *epsabs*, *limit*]) Function
quad_qags (*f*, *x*, *a*, *b*, [*epsrel*, *epsabs*, *limit*]) Function

Integration of a general function over a finite interval. `quad_qags` implements globally adaptive interval subdivision with extrapolation (de Doncker, 1978) by the Epsilon algorithm (Wynn, 1956).

`quad_qags` computes the integral

$$\int_a^b f(x)dx$$

The function to be integrated is *f(x)*, with dependent variable *x*, and the function is to be integrated between the limits *a* and *b*.

The integrand may be specified as the name of a Maxima or Lisp function or operator, a Maxima lambda expression, or a general Maxima expression.

The keyword arguments are optional and may be specified in any order. They all take the form `key=val`. The keyword arguments are:

epsrel Desired relative error of approximation. Default is 1d-10.

epsabs Desired absolute error of approximation. Default is 0.

limit Size of internal work array. (*limit* - *limlst*)/2 is the maximum number of subintervals to use. Default is 200.

`quad_qags` returns a list of four elements:

an approximation to the integral,

the estimated absolute error of the approximation,

the number integrand evaluations,

an error code.

The error code (fourth element of the return value) can have the values:

`0` no problems were encountered;

`1` too many sub-intervals were done;

`2` excessive roundoff error is detected;

`3` extremely bad integrand behavior occurs;

`4` failed to converge

`5` integral is probably divergent or slowly convergent

`6` if the input is invalid.

Examples:

```
(%i1) quad_qags (x^(1/2)*log(1/x), x, 0, 1, 'epsrel=1d-10);
(%o1)   [.4444444444444448, 1.11022302462516E-15, 315, 0]
```

Note that `quad_qags` is more accurate and efficient than `quad_qag` for this integrand.

quad_qagi (*f*(*x*), *x*, *a*, *inftype*, [*epsrel*, *epsabs*, *limit*]) Function
quad_qagi (*f*, *x*, *a*, *inftype*, [*epsrel*, *epsabs*, *limit*]) Function

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.

The parameter *inftype* determines the integration interval as follows:

`inf` The interval is from *a* to positive infinity.

`minf` The interval is from negative infinity to *a*.

`both` The interval is the entire real line.

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-10.

epsabs Desired absolute error of approximation. Default is 0.

limit Size of internal work array. (*limit* - *limlst*)/2 is the maximum number of subintervals to use. Default is 200.

`quad_qagi` returns a list of four elements:

an approximation to the integral,

the estimated absolute error of the approximation,

the number integrand evaluations,

an error code.

The error code (fourth element of the return value) can have the values:

`0` no problems were encountered;

`1` too many sub-intervals were done;

`2` excessive roundoff error is detected;

`3` extremely bad integrand behavior occurs;

`4` failed to converge

`5` integral is probably divergent or slowly convergent

`6` if the input is invalid.

Examples:

```
(%i1) quad_qagi (x^2*exp(-4*x), x, 0, inf, 'epsrel=1d-8);
(%o1)        [0.03125, 2.95916102995002E-11, 105, 0]
(%i2) integrate (x^2*exp(-4*x), x, 0, inf);
                                1
(%o2)                           --
                                32
```

quad_qawc (*f*(*x*), *x*, *c*, *a*, *b*, [*epsrel*, *epsabs*, *limit*]) Function
quad_qawc (*f*, *x*, *c*, *a*, *b*, [*epsrel*, *epsabs*, *limit*]) Function

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

$$\int_a^b \frac{f\left(x\right)}{x-c}\,dx$$

using the Quadpack QAWC routine. The function to be integrated is *f(x)*/(*x* - *c*), with dependent variable *x*, and the function is to be integrated over the interval *a* to *b*.

The integrand may be specified as the name of a Maxima or Lisp function or operator, a Maxima lambda expression, or a general Maxima expression.

The keyword arguments are optional and may be specified in any order. They all take the form `key=val`. The keyword arguments are:

epsrel Desired relative error of approximation. Default is 1d-10.

epsabs Desired absolute error of approximation. Default is 0.

limit Size of internal work array. (*limit* - *limlst*)/2 is the maximum number of subintervals to use. Default is 200.

`quad_qawc` returns a list of four elements:

an approximation to the integral,

the estimated absolute error of the approximation,

the number integrand evaluations,

an error code.

The error code (fourth element of the return value) can have the values:

`0` no problems were encountered;

`1` too many sub-intervals were done;

`2` excessive roundoff error is detected;

`3` extremely bad integrand behavior occurs;

`6` if the input is invalid.

Examples:

```
(%i1) quad_qawc (2^(-5)*((x-1)^2+4^(-5))^(-1), x, 2, 0, 5, 'epsrel=1d-7);
(%o1)    [- 3.130120337415925, 1.306830140249558E-8, 495, 0]
(%i2) integrate (2^(-alpha)*(((x-1)^2 + 4^(-alpha))*(x-2))^(-1),
      x, 0, 5);
Principal Value
                        alpha
         alpha       9 4                  9
        4      log(------------- + -------------)
                          alpha           alpha
                   64 4       + 4   64 4      + 4
(%o2) (---------------------------------------
                         alpha
                      2 4      + 2

        3 alpha                        3 alpha
        -------                        -------
           2             alpha/2          2          alpha/2
    2 4         atan(4 4        )   2 4         atan(4       )   alpha
```

```
  - ------------------------- - ------------------------)/2
                alpha                        alpha
             2 4      + 2                 2 4      + 2
(%i3) ev (%, alpha=5, numer);
(%o3)                    - 3.130120337415917
```

quad_qawf (*f*(*x*), *x*, *a*, *omega*, *trig*, [*epsabs*, *limit*, *maxp1*, *limlst*]) Function
quad_qawf (*f*, *x*, *a*, *omega*, *trig*, [*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 *trig*:

`cos` $w(x) = cos(omegax)$

`sin` $w(x) = sin(omegax)$

The integrand may be specified as the name of a Maxima or Lisp function or operator, a Maxima lambda expression, or a general Maxima expression.

The keyword arguments are optional and may be specified in any order. They all take the form `key=val`. The keyword arguments are:

epsabs Desired absolute error of approximation. Default is 1d-10.

limit Size of internal work array. (*limit* - *limlst*)/2 is the maximum number of subintervals to use. Default is 200.

maxp1 Maximum number of Chebyshev moments. Must be greater than 0. Default is 100.

limlst Upper bound on the number of cycles. Must be greater than or equal to 3. Default is 10.

`quad_qawf` returns a list of four elements:

an approximation to the integral,

the estimated absolute error of the approximation,

the number integrand evaluations,

an error code.

The error code (fourth element of the return value) can have the values:

`0` no problems were encountered;

`1` too many sub-intervals were done;

`2` excessive roundoff error is detected;

`3` extremely bad integrand behavior occurs;

`6` if the input is invalid.

Examples:

```
(%i1) quad_qawf (exp(-x^2), x, 0, 1, 'cos, 'epsabs=1d-9);
(%o1)   [.6901942235215714, 2.84846300257552E-11, 215, 0]
(%i2) integrate (exp(-x^2)*cos(x), x, 0, inf);
                           - 1/4
                         %e      sqrt(%pi)
(%o2)                    -----------------
                                 2
(%i3) ev (%, numer);
(%o3)                    .6901942235215714
```

quad_qawo (*f*(*x*), *x*, *a*, *b*, *omega*, *trig*, [*epsrel*, *epsabs*, *limit*, *maxp1*, *limlst*]) Function

quad_qawo (*f*, *x*, *a*, *b*, *omega*, *trig*, [*epsrel*, *epsabs*, *limit*, *maxp1*, *limlst*]) Function

Integration of $cos(omegax)f(x)$ or $sin(omegax)f(x)$ 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` $w(x) = cos(omegax)$

`sin` $w(x) = sin(omegax)$

The integrand may be specified as the name of a Maxima or Lisp function or operator, a Maxima lambda expression, or a general Maxima expression.

The keyword arguments are optional and may be specified in any order. They all take the form `key=val`. The keyword arguments are:

epsrel Desired relative error of approximation. Default is 1d-10.

epsabs Desired absolute error of approximation. Default is 0.

limit Size of internal work array. (*limit* - *limlst*)/2 is the maximum number of subintervals to use. Default is 200.

maxp1 Maximum number of Chebyshev moments. Must be greater than 0. Default is 100.

limlst Upper bound on the number of cycles. Must be greater than or equal to 3. Default is 10.

`quad_qawo` returns a list of four elements:

an approximation to the integral,

the estimated absolute error of the approximation,

the number integrand evaluations,

an error code.

The error code (fourth element of the return value) can have the values:

`0` no problems were encountered;

`1` too many sub-intervals were done;

`2` excessive roundoff error is detected;

`3` extremely bad integrand behavior occurs;

`6` if the input is invalid.

Examples:

```
(%i1) quad_qawo (x^(-1/2)*exp(-2^(-2)*x), x, 1d-8, 20*2^2, 1, cos);
(%o1)     [1.376043389877692, 4.72710759424899E-11, 765, 0]
(%i2) rectform (integrate (x^(-1/2)*exp(-2^(-alpha)*x) * cos(x),
      x, 0, inf));
                    alpha/2 - 1/2            2 alpha
        sqrt(%pi) 2              sqrt(sqrt(2        + 1) + 1)
(%o2)   ---------------------------------------------------
                                2 alpha
                           sqrt(2        + 1)
(%i3) ev (%, alpha=2, numer);
(%o3)                     1.376043390090716
```

quad_qaws (*f*(*x*), *x*, *a*, *b*, *alpha*, *beta*, *wfun*, [*epsrel*, *epsabs*, *limit*]) Function
quad_qaws (*f*, *x*, *a*, *b*, *alpha*, *beta*, *wfun*, [*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.

`quad_qaws` computes the integral using the Quadpack QAWS routine:

$$\int_a^b f(x)w(x)dx$$

The weight function w is selected by *wfun*:

`1` $w(x) = (x-a)^a lpha(b-x)^b eta$

`2` $w(x) = (x-a)^a lpha(b-x)^b etalog(x-a)$

`3` $w(x) = (x-a)^a lpha(b-x)^b etalog(b-x)$

`4` $w(x) = (x-a)^a lpha(b-x)^b etalog(x-a)log(b-x)$

The integrand may be specified as the name of a Maxima or Lisp function or operator, a Maxima lambda expression, or a general Maxima expression.

The keyword arguments are optional and may be specified in any order. They all take the form `key=val`. The keyword arguments are:

epsrel Desired relative error of approximation. Default is 1d-10.

epsabs Desired absolute error of approximation. Default is 0.

limit Size of internal work array. *limit*is the maximum number of subintervals to use. Default is 200.

`quad_qaws` returns a list of four elements:

an approximation to the integral,

the estimated absolute error of the approximation,

the number integrand evaluations,

an error code.

The error code (fourth element of the return value) can have the values:

`0` no problems were encountered;

`1` too many sub-intervals were done;

`2` excessive roundoff error is detected;

`3` extremely bad integrand behavior occurs;

`6` if the input is invalid.

Examples:

```
(%i1) quad_qaws (1/(x+1+2^(-4)), x, -1, 1, -0.5, -0.5, 1, 'epsabs=1d-9);
(%o1)     [8.750097361672832, 1.24321522715422E-10, 170, 0]
(%i2) integrate ((1-x*x)^(-1/2)/(x+1+2^(-alpha)), x, -1, 1);
       alpha
Is  4 2      - 1  positive, negative, or zero?

pos;
                            alpha         alpha
                   2 %pi 2      sqrt(2 2      + 1)
(%o2)              ------------------------------
                                 alpha
                             4 2      + 2
(%i3) ev (%, alpha=4, numer);
(%o3)                     8.750097361672829
```

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)`.

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 `solve` is called to find an exact solution.

In some cases, `solve` is not be able to find a solution, or if it does the solution may be a very large expression.

If the equation is univariate and is either linear, quadratic, or biquadratic, then again `solve` is called if no approximations have been introduced. If approximations have been introduced or the equation is not univariate and neither linear, quadratic, or biquadratic, then if the switch `realonly` is `true`, the function `realroots` is called to find the real-valued solutions. If `realonly` is `false`, then `allroots` is called which looks for real and complex-valued solutions.

If `algsys` produces a solution which has fewer significant digits than required, the user can change the value of `algepsilon` to a higher value.

If `algexact` is set to `true`, `solve` will always be called.

(4) Finally, the solutions obtained in step (3) are substituted into previous levels and the solution process returns to (1).

When `algsys` encounters a multivariate equation which contains floating point approximations (usually due to its failing to find exact solutions at an earlier stage), then it does not attempt to apply exact methods to such equations and instead prints the message: "`algsys` cannot solve - system too complicated."

Interactions with `radcan` can produce large or complicated expressions. In that case, it may be possible to isolate parts of the result with `pickapart` or `reveal`.

Occasionally, `radcan` may introduce an imaginary unit `%i` into a solution which is actually real-valued.

Examples:

```
(%i1) e1: 2*x*(1 - a1) - 2*(x - 1)*a2;
(%o1)              2 (1 - a1) x - 2 a2 (x - 1)
(%i2) e2: a2 - a1;
(%o2)                          a2 - a1
(%i3) e3: a1*(-y - x^2 + 1);
                                      2
(%o3)                     a1 (- y - x  + 1)
(%i4) e4: a2*(y - (x - 1)^2);
                                           2
(%o4)                     a2 (y - (x - 1) )
(%i5) algsys ([e1, e2, e3, e4], [x, y, a1, a2]);
(%o5) [[x = 0, y = %r1, a1 = 0, a2 = 0],

                                     [x = 1, y = 0, a1 = 1, a2 = 1]]
(%i6) e1: x^2 - y^2;
                                2    2
(%o6)                          x  - y
(%i7) e2: -1 - y + 2*y^2 - x + x^2;
                             2        2
(%o7)                  2 y  - y + x  - x - 1
(%i8) algsys ([e1, e2], [x, y]);
                  1            1
(%o8) [[x = - -------, y = -------],
              sqrt(3)      sqrt(3)
```

```
        1              1             1        1
[x = -------, y = - -------], [x = - -, y = - -], [x = 1, y = 1]]
     sqrt(3)        sqrt(3)          3        3
```

allroots (*expr*) 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`.

For complex polynomials an algorithm by Jenkins and Traub is used (Algorithm 419, *Comm. ACM*, vol. 15, (1972), p. 97). For real polynomials the algorithm used is due to Jenkins (Algorithm 493, *ACM TOMS*, vol. 1, (1975), p.178).

Examples:

```
(%i1) eqn: (1 + 2*x)^3 = 13.5*(1 + x^5);
                              3          5
(%o1)               (2 x + 1)  = 13.5 (x  + 1)
(%i2) soln: allroots (eqn);
(%o2) [x = .8296749902129361, x = - 1.015755543828121,

x = .9659625152196369 %i - .4069597231924075,

x = - .9659625152196369 %i - .4069597231924075, x = 1.0]
(%i3) for e in soln
        do (e2: subst (e, eqn), disp (expand (lhs(e2) - rhs(e2))));
                      - 3.5527136788005E-15

                    - 5.32907051820075E-15

          4.44089209850063E-15 %i - 4.88498130835069E-15

         - 4.44089209850063E-15 %i - 4.88498130835069E-15

                       3.5527136788005E-15

(%o3)                          done
(%i4) polyfactor: true$
(%i5) allroots (eqn);
```

```
(%o5) - 13.5 (x - 1.0) (x - .8296749902129361)

                                2
 (x + 1.015755543828121) (x  + .8139194463848151 x

 + 1.098699797110288)
```

backsubst Option variable

Default value: `true`

When `backsubst` is `false`, prevents back substitution 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.

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
                                       ---------- - -
                 sqrt(3) %i   1            2       2   1
(%t4)    x = (- ---------- - -) %t3 + -------------- - -
                    2        2           9 %t3       3

                                       sqrt(3) %i   1
                                     - ---------- - -
                sqrt(3) %i   1             2       2   1
(%t5)    x = (---------- - -) %t3 + ---------------- - -
                   2        2            9 %t3       3

                                    1     1
(%t6)                  x = %t3 + ----- - -
                                  9 %t3   3
(%o6)                    [%t4, %t5, %t6]
(%i6) breakup: false$
(%i7) solve (x^3 + x^2 - 1);
```

```
Solution:

                 sqrt(3) %i   1
                 ---------- - -
                     2        2        sqrt(23)    25 1/3
(%t7) x = ---------------------- + (--------- + --)
                 sqrt(23)    25 1/3    6 sqrt(3)   54
              9 (--------- + --)
                 6 sqrt(3)   54

                                                 sqrt(3) %i   1    1
                                               (- ---------- - -) - -
                                                     2        2    3

              sqrt(23)    25 1/3  sqrt(3) %i   1
(%t8) x = (--------- + --)    (---------- - -)
              6 sqrt(3)   54          2        2

                                                  sqrt(3) %i   1
                                                - ---------- - -
                                                     2        2      1
                                             + ---------------------- - -
                                                  sqrt(23)    25 1/3   3
                                               9 (--------- + --)
                                                  6 sqrt(3)   54

              sqrt(23)    25 1/3              1              1
(%t9)  x = (--------- + --)    + ---------------------- - -
              6 sqrt(3)   54          sqrt(23)    25 1/3   3
                                   9 (--------- + --)
                                      6 sqrt(3)   54
(%o9)                     [%t7, %t8, %t9]
```

dimension (*eqn*) Function
dimension (*eqn_1*, ..., *eqn_n*) Function

`dimen` is a package for dimensional analysis. `load ("dimen")` loads this package. `demo ("dimen")` displays a short demonstration.

dispflag Option variable

Default value: `true`

If set to `false` within a `block` will inhibit the display of output generated by the solve functions called from within the `block`. Termination of the `block` with a dollar sign, $, sets `dispflag` to `false`.

funcsolve (*eqn*, *g*(*t*)) Function

Returns `[`*g*`(`*t*`) = ...]` or `[]`, depending on whether or not there exists a rational function *g*`(`*t*`)` satisfying *eqn*, which must be a first order, linear polynomial in (for this case) *g*`(`*t*`)` and *g*`(`*t*`+1)`

```
(%i1) eqn: (n + 1)*f(n) - (n + 3)*f(n + 1)/(n + 1) =
      (n - 1)/(n + 2);
                              (n + 3) f(n + 1)   n - 1
(%o1)         (n + 1) f(n) - ---------------- = -----
                                    n + 1         n + 2
(%i2) funcsolve (eqn, f(n));

Dependent equations eliminated:  (4 3)
                                     n
(%o2)                 f(n) = ---------------
                             (n + 1) (n + 2)
```

Warning: this is a very rudimentary implementation – many safety checks and obvious generalizations are missing.

globalsolve Option variable

Default value: `false`

When `globalsolve` is `true`, solved-for variables are assigned the solution values found by `linsolve`, and by `solve` when solving two or more linear equations.

When `globalsolve` is `false`, solutions found by `linsolve` and by `solve` when solving two or more linear equations are expressed as equations, and the solved-for variables are not assigned.

When solving anything other than two or more linear equations, `solve` ignores `globalsolve`. Other functions which solve equations (e.g., `algsys`) always ignore `globalsolve`.

Examples:

```
(%i1) globalsolve: true$
(%i2) solve ([x + 3*y = 2, 2*x - y = 5], [x, y]);
Solution

                                  17
(%t2)                        x : --
                                  7

                                    1
(%t3)                        y : - -
                                    7
(%o3)                     [[%t2, %t3]]
(%i3) x;
                                  17
(%o3)                             --
                                  7
(%i4) y;
                                    1
(%o4)                             - -
                                    7
(%i5) globalsolve: false$
(%i6) kill (x, y)$
```

```
(%i7) solve ([x + 3*y = 2, 2*x - y = 5], [x, y]);
Solution

                                  17
(%t7)                        x = --
                                  7

                                    1
(%t8)                        y = - -
                                    7
(%o8)                     [[%t7, %t8]]
(%i8) x;
(%o8)                             x
(%i9) y;
(%o9)                             y
```

ieqn (*ie, unk, tech, n, guess*) Function

`inteqn` is a package for solving integral equations. `load ("inteqn")` loads this package.

ie is the integral equation; *unk* is the unknown function; *tech* is the technique to be tried from those given above (*tech* = `first` means: try the first technique which finds a solution; *tech* = `all` means: try all applicable techniques); *n* is the maximum number of terms to take for `taylor`, `neumann`, `firstkindseries`, or `fredseries` (it is also the maximum depth of recursion for the differentiation method); *guess* is the initial guess for `neumann` or `firstkindseries`.

Default values for the 2nd thru 5th parameters are:

unk: `p(x)`, where *p* is the first function encountered in an integrand which is unknown to Maxima and *x* is the variable which occurs as an argument to the first occurrence of *p* found outside of an integral in the case of `secondkind` equations, or is the only other variable besides the variable of integration in `firstkind` equations. If the attempt to search for *x* fails, the user will be asked to supply the independent variable.

tech: `first`

n: 1

guess: `none` which will cause `neumann` and `firstkindseries` to use `f(x)` as an initial guess.

ieqnprint Option variable

Default value: `true`

`ieqnprint` governs the behavior of the result returned by the `ieqn` command. When `ieqnprint` is `false`, the lists returned by the `ieqn` function are of the form

[*solution*, *technique used*, *nterms*, *flag*]

where *flag* is absent if the solution is exact.

Otherwise, it is the word `approximate` or `incomplete` corresponding to an inexact or non-closed form solution, respectively. If a series method was used, *nterms* gives the number of terms taken (which could be less than the n given to `ieqn` if an error prevented generation of further terms).

lhs (*expr*) Function

Returns the left-hand side (that is, the first argument) of the expression *expr*, when the operator of *expr* is one of the relational operators `< <= = # equal notequal >= >`, one of the assignment operators `:= ::= : ::`, or a user-defined binary infix operator, as declared by `infix`.

When *expr* is an atom or its operator is something other than the ones listed above, `lhs` returns *expr*.

See also `rhs`.

Examples:

```
(%i1) e: aa + bb = cc;
(%o1)                       bb + aa = cc
(%i2) lhs (e);
(%o2)                          bb + aa
(%i3) rhs (e);
(%o3)                            cc
(%i4) [lhs (aa < bb), lhs (aa <= bb), lhs (aa >= bb),
       lhs (aa > bb)];
(%o4)                     [aa, aa, aa, aa]
(%i5) [lhs (aa = bb), lhs (aa # bb), lhs (equal (aa, bb)),
       lhs (notequal (aa, bb))];
(%o5)                     [aa, aa, aa, aa]
(%i6) e1: '(foo(x) := 2*x);
(%o6)                       foo(x) := 2 x
(%i7) e2: '(bar(y) ::= 3*y);
(%o7)                      bar(y) ::= 3 y
(%i8) e3: '(x : y);
(%o8)                           x : y
(%i9) e4: '(x :: y);
(%o9)                          x :: y
(%i10) [lhs (e1), lhs (e2), lhs (e3), lhs (e4)];
(%o10)                [foo(x), bar(y), x, x]
(%i11) infix ("][");
(%o11)                           ][
(%i12) lhs (aa ][ bb);
(%o12)                           aa
```

linsolve ([*expr_1*, ..., *expr_m*], [*x_1*, ..., *x_n*]) Function

Solves the list of simultaneous linear equations for the list of variables. The expressions must each be polynomials in the variables and may be equations.

When `globalsolve` is `true`, each solved-for variable is bound to its value in the solution of the equations.

When `backsubst` is `false`, `linsolve` does not carry out back substitution after the equations have been triangularized. This may be necessary in very big problems where back substitution would cause the generation of extremely large expressions.

When `linsolve_params` is `true`, `linsolve` also generates the `%r` symbols used to represent arbitrary parameters described in the manual under `algsys`. Otherwise,

`linsolve` solves an under-determined system of equations with some variables expressed in terms of others.

When `programmode` is `false`, `linsolve` displays the solution with intermediate expression (`%t`) labels, and returns the list of labels.

```
(%i1) e1: x + z = y;
(%o1)                         z + x = y
(%i2) e2: 2*a*x - y = 2*a^2;
                                              2
(%o2)                     2 a x - y = 2 a
(%i3) e3: y - 2*z = 2;
(%o3)                         y - 2 z = 2
(%i4) [globalsolve: false, programmode: true];
(%o4)                        [false, true]
(%i5) linsolve ([e1, e2, e3], [x, y, z]);
(%o5)              [x = a + 1, y = 2 a, z = a - 1]
(%i6) [globalsolve: false, programmode: false];
(%o6)                       [false, false]
(%i7) linsolve ([e1, e2, e3], [x, y, z]);
Solution

(%t7)                         z = a - 1

(%t8)                          y = 2 a

(%t9)                         x = a + 1
(%o9)                       [%t7, %t8, %t9]
(%i9) ''%;
(%o9)              [z = a - 1, y = 2 a, x = a + 1]
(%i10) [globalsolve: true, programmode: false];
(%o10)                       [true, false]
(%i11) linsolve ([e1, e2, e3], [x, y, z]);
Solution

(%t11)                        z : a - 1

(%t12)                         y : 2 a

(%t13)                        x : a + 1
(%o13)                   [%t11, %t12, %t13]
(%i13) ''%;
(%o13)             [z : a - 1, y : 2 a, x : a + 1]
(%i14) [x, y, z];
(%o14)                   [a + 1, 2 a, a - 1]
(%i15) [globalsolve: true, programmode: true];
(%o15)                       [true, true]
(%i16) linsolve ([e1, e2, e3], '[x, y, z]);
(%o16)             [x : a + 1, y : 2 a, z : a - 1]
(%i17) [x, y, z];
(%o17)                   [a + 1, 2 a, a - 1]
```

linsolvewarn Option variable

Default value: `true`

When `linsolvewarn` is `true`, `linsolve` prints a message "Dependent equations eliminated".

linsolve_params Option variable

Default value: `true`

When `linsolve_params` is `true`, `linsolve` also generates the `%r` symbols used to represent arbitrary parameters described in the manual under `algsys`. Otherwise, `linsolve` solves an under-determined system of equations with some variables expressed in terms of others.

multiplicities System variable

Default value: `not_set_yet`

`multiplicities` is set to a list of the multiplicities of the individual solutions returned by `solve` or `realroots`.

nroots (*p, low, high*) Function

Returns the number of real roots of the real univariate polynomial *p* in the half-open interval `(`*low*`,` *high*`]`. The endpoints of the interval may be `minf` or `inf`. 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*) Function

where p is a polynomial with integer coefficients and n is a positive integer returns q, a polynomial over the integers, such that q^n=p or prints an error message indicating that p is not a perfect nth power. This routine is much faster than `factor` or even `sqfr`.

programmode Option variable

Default value: `true`

When `programmode` is `true`, `solve`, `realroots`, `allroots`, and `linsolve` return solutions as elements in a list. (Except when `backsubst` is set to `false`, in which case `programmode: false` is assumed.)

When `programmode` is `false`, `solve`, etc. create intermediate expression labels `%t1`, `t2`, etc., and assign the solutions to them.

realonly Option variable

Default value: `false`

When `realonly` is `true`, `algsys` returns only those solutions which are free of `%i`.

realroots (*expr, bound*) Function
realroots (*eqn, bound*) Function
realroots (*expr*) Function
realroots (*eqn*) Function

Computes rational approximations of the real roots of the polynomial *expr* or polynomial equation *eqn* of one variable, to within a tolerance of *bound*. Coefficients of *expr* or *eqn* must be literal numbers; symbol constants such as `%pi` are rejected.

`realroots` assigns the multiplicities of the roots it finds to the global variable `multiplicities`.

`realroots` constructs a Sturm sequence to bracket each root, and then applies bisection to refine the approximations. All coefficients are converted to rational equivalents before searching for roots, and computations are carried out by exact rational arithmetic. Even if some coefficients are floating-point numbers, the results are rational (unless coerced to floats by the `float` or `numer` flags).

When *bound* is less than 1, all integer roots are found exactly. When *bound* is unspecified, it is assumed equal to the global variable `rootsepsilon`.

When the global variable `programmode` is `true`, `realroots` returns a list of the form `[x = x_1, x = x_2, ...]`. When `programmode` is `false`, `realroots` creates intermediate expression labels `%t1`, `%t2`, ..., assigns the results to them, and returns the list of labels.

Examples:

```
(%i1) realroots (-1 - x + x^5, 5e-6);
                                 612003
(%o1)                      [x = ------]
                                 524288
(%i2) ev (%[1], float);
(%o2)                  x = 1.167303085327148
(%i3) ev (-1 - x + x^5, %);
(%o3)                - 7.396496210176905E-6
(%i1) realroots (expand ((1 - x)^5 * (2 - x)^3 * (3 - x)), 1e-20);
(%o1)                  [x = 1, x = 2, x = 3]
(%i2) multiplicities;
(%o2)                       [5, 3, 1]
```

rhs (*expr*) Function

Returns the right-hand side (that is, the second argument) of the expression *expr*, when the operator of *expr* is one of the relational operators `< <= = # equal notequal >= >`, one of the assignment operators `:= ::= : ::`, or a user-defined binary infix operator, as declared by `infix`.

When *expr* is an atom or its operator is something other than the ones listed above, `rhs` returns 0.

See also `lhs`.

Examples:

```
(%i1) e: aa + bb = cc;
(%o1)                      bb + aa = cc
```

```
(%i2) lhs (e);
(%o2)                          bb + aa
(%i3) rhs (e);
(%o3)                            cc
(%i4) [rhs (aa < bb), rhs (aa <= bb), rhs (aa >= bb),
       rhs (aa > bb)];
(%o4)                     [bb, bb, bb, bb]
(%i5) [rhs (aa = bb), rhs (aa # bb), rhs (equal (aa, bb)),
       rhs (notequal (aa, bb))];
(%o5)                     [bb, bb, bb, bb]
(%i6) e1: '(foo(x) := 2*x);
(%o6)                      foo(x) := 2 x
(%i7) e2: '(bar(y) ::= 3*y);
(%o7)                     bar(y) ::= 3 y
(%i8) e3: '(x : y);
(%o8)                           x : y
(%i9) e4: '(x :: y);
(%o9)                          x :: y
(%i10) [rhs (e1), rhs (e2), rhs (e3), rhs (e4)];
(%o10)                    [2 x, 3 y, y, y]
(%i11) infix ("][");
(%o11)                           ][
(%i12) rhs (aa ][ bb);
(%o12)                           bb
```

rootsconmode Option variable

Default value: `true`

`rootsconmode` governs the behavior of the `rootscontract` command. See `rootscontract` for details.

rootscontract (*expr*) Function

Converts products of roots into roots of products. For example, `rootscontract (sqrt(x)*y^(3/2))` yields `sqrt(x*y^3)`.

When `radexpand` is `true` and `domain` is `real`, `rootscontract` converts `abs` into `sqrt`, e.g., `rootscontract (abs(x)*sqrt(y))` yields `sqrt(x^2*y)`.

There is an option `rootsconmode` affecting `rootscontract` as follows:

Problem	Value of rootsconmode	Result of applying rootscontract
`x^(1/2)*y^(3/2)`	`false`	`(x*y^3)^(1/2)`
`x^(1/2)*y^(1/4)`	`false`	`x^(1/2)*y^(1/4)`
`x^(1/2)*y^(1/4)`	`true`	`(x*y^(1/2))^(1/2)`
`x^(1/2)*y^(1/3)`	`true`	`x^(1/2)*y^(1/3)`
`x^(1/2)*y^(1/4)`	`all`	`(x^2*y)^(1/4)`
`x^(1/2)*y^(1/3)`	`all`	`(x^3*y^2)^(1/6)`

When `rootsconmode` is `false`, `rootscontract` contracts only with respect to rational number exponents whose denominators are the same. The key to the `rootsconmode:`

`true` examples is simply that 2 divides into 4 but not into 3. `rootsconmode: all` involves taking the least common multiple of the denominators of the exponents.

`rootscontract` uses `ratsimp` in a manner similar to `logcontract`.

Examples:

```
(%i1) rootsconmode: false$
(%i2) rootscontract (x^(1/2)*y^(3/2));
                                    3
(%o2)                        sqrt(x y )
(%i3) rootscontract (x^(1/2)*y^(1/4));
                                    1/4
(%o3)                        sqrt(x) y
(%i4) rootsconmode: true$
(%i5) rootscontract (x^(1/2)*y^(1/4));
(%o5)                       sqrt(x sqrt(y))
(%i6) rootscontract (x^(1/2)*y^(1/3));
                                    1/3
(%o6)                        sqrt(x) y
(%i7) rootsconmode: all$
(%i8) rootscontract (x^(1/2)*y^(1/4));
                                 2   1/4
(%o8)                           (x  y)
(%i9) rootscontract (x^(1/2)*y^(1/3));
                                 3  2 1/6
(%o9)                           (x  y )
(%i10) rootsconmode: false$
(%i11) rootscontract (sqrt(sqrt(x) + sqrt(1 + x))
                     *sqrt(sqrt(1 + x) - sqrt(x)));
(%o11)                               1
(%i12) rootsconmode: true$
(%i13) rootscontract (sqrt(5+sqrt(5)) - 5^(1/4)*sqrt(1+sqrt(5)));
(%o13)                               0
```

rootsepsilon Option variable

Default value: 1.0e-7

`rootsepsilon` is the tolerance which establishes the confidence interval for the roots found by the `realroots` function.

solve (*expr*, *x*) Function
solve (*expr*) Function
solve ([*eqn_1*, ..., *eqn_n*], [*x_1*, ..., *x_n*]) Function

Solves the algebraic equation *expr* for the variable *x* and returns a list of solution equations in *x*. If *expr* is not an equation, the equation *expr* `= 0` is assumed in its place. *x* may be a function (e.g. `f(x)`), or other non-atomic expression except a sum or product. *x* may be omitted if *expr* contains only one variable. *expr* may be a rational expression, and may contain trigonometric functions, exponentials, etc.

The following method is used:

Let E be the expression and X be the variable. If E is linear in X then it is trivially solved for X. Otherwise if E is of the form `A*X^N + B` then the result is `(-B/A)^1/N)` times the `N`'th roots of unity.

If E is not linear in X then the gcd of the exponents of X in E (say N) is divided into the exponents and the multiplicity of the roots is multiplied by N. Then `solve` is called again on the result. If E factors then `solve` is called on each of the factors. Finally `solve` will use the quadratic, cubic, or quartic formulas where necessary.

In the case where E is a polynomial in some function of the variable to be solved for, say `F(X)`, then it is first solved for `F(X)` (call the result C), then the equation `F(X)=C` can be solved for X provided the inverse of the function F is known.

`breakup` if `false` will cause `solve` to express the solutions of cubic or quartic equations as single expressions rather than as made up of several common subexpressions which is the default.

`multiplicities` - will be set to a list of the multiplicities of the individual solutions returned by `solve`, `realroots`, or `allroots`. Try `apropos (solve)` for the switches which affect `solve`. `describe` may then by used on the individual switch names if their purpose is not clear.

`solve ([`*eqn_1*`, ..., `*eqn_n*`], [`*x_1*`, ..., `*x_n*`])` solves a system of simultaneous (linear or non-linear) polynomial equations by calling `linsolve` or `algsys` and returns a list of the solution lists in the variables. In the case of `linsolve` this list would contain a single list of solutions. It takes two lists as arguments. The first list represents the equations to be solved; the second list is a list of the unknowns to be determined. If the total number of variables in the equations is equal to the number of equations, the second argument-list may be omitted. For linear systems if the given equations are not compatible, the message `inconsistent` will be displayed (see the `solve_inconsistent_error` switch); if no unique solution exists, then `singular` will be displayed.

When `programmode` is `false`, `solve` displays solutions with intermediate expression (`%t`) labels, and returns the list of labels.

When `globalsolve` is `true` and the problem is to solve two or more linear equations, each solved-for variable is bound to its value in the solution of the equations.

Examples:

```
(%i1) solve (asin (cos (3*x))*(f(x) - 1), x);

SOLVE is using arc-trig functions to get a solution.
Some solutions will be lost.
                              %pi
(%o1)                    [x = ---, f(x) = 1]
                               6
(%i2) ev (solve (5^f(x) = 125, f(x)), solveradcan);
                                  log(125)
(%o2)                     [f(x) = --------]
                                   log(5)
(%i3) [4*x^2 - y^2 = 12, x*y - x = 2];
                         2    2
(%o3)              [4 x  - y  = 12, x y - x = 2]
```

```
(%i4) solve (%, [x, y]);
(%o4) [[x = 2, y = 2], [x = .5202594388652008 %i

 - .1331240357358706, y = .0767837852378778

 - 3.608003221870287 %i], [x = - .5202594388652008 %i

 - .1331240357358706, y = 3.608003221870287 %i

 + .0767837852378778], [x = - 1.733751846381093,

y = - .1535675710019696]]
(%i5) solve (1 + a*x + x^3, x);
                                          3
              sqrt(3) %i   1   sqrt(4 a  + 27)   1 1/3
(%o5) [x = (- ---------- - -) (--------------- - -)
                  2        2      6 sqrt(3)      2

        sqrt(3) %i   1
       (---------- - -) a
            2        2
 - --------------------------, x =
             3
     sqrt(4 a  + 27)   1 1/3
  3 (--------------- - -)
        6 sqrt(3)      2

                         3
 sqrt(3) %i   1   sqrt(4 a  + 27)   1 1/3
(---------- - -) (--------------- - -)
     2        2      6 sqrt(3)      2

         sqrt(3) %i   1
      (- ---------- - -) a
           2        2
 - --------------------------, x =
             3
     sqrt(4 a  + 27)   1 1/3
  3 (--------------- - -)
        6 sqrt(3)      2

         3
 sqrt(4 a  + 27)   1 1/3              a
(--------------- - -)    - --------------------------]
    6 sqrt(3)      2                 3
                              sqrt(4 a  + 27)   1 1/3
                            3 (--------------- - -)
                                  6 sqrt(3)      2
(%i6) solve (x^3 - 1);
```

```
                 sqrt(3) %i - 1          sqrt(3) %i + 1
(%o6)    [x = --------------, x = - --------------, x = 1]
                       2                       2
(%i7) solve (x^6 - 1);
              sqrt(3) %i + 1       sqrt(3) %i - 1
(%o7) [x = --------------, x = --------------, x = - 1,
                    2                    2

                        sqrt(3) %i + 1          sqrt(3) %i - 1
                 x = - --------------, x = - --------------, x = 1]
                              2                       2
(%i8) ev (x^6 - 1, %[1]);
                                        6
                          (sqrt(3) %i + 1)
(%o8)                     ----------------- - 1
                                 64
(%i9) expand (%);
(%o9)                               0
(%i10) x^2 - 1;
                                 2
(%o10)                          x  - 1
(%i11) solve (%, x);
(%o11)                     [x = - 1, x = 1]
(%i12) ev (%th(2), %[1]);
(%o12)                              0
```

solvedecomposes Option variable

Default value: `true`

When `solvedecomposes` is `true`, `solve` calls `polydecomp` if asked to solve polynomials.

solveexplicit Option variable

Default value: `false`

When `solveexplicit` is `true`, inhibits `solve` from returning implicit solutions, that is, solutions of the form `F(x) = 0` where `F` is some function.

solvefactors Option variable

Default value: `true`

When `solvefactors` is `false`, `solve` does not try to factor the expression. The `false` setting may be desired in some cases where factoring is not necessary.

solvenullwarn Option variable

Default value: `true`

When `solvenullwarn` is `true`, `solve` prints a warning message if called with either a null equation list or a null variable list. For example, `solve ([], [])` would print two warning messages and return `[]`.

solveradcan Option variable

Default value: `false`

When `solveradcan` is `true`, `solve` calls `radcan` which makes `solve` slower but will allow certain problems containing exponentials and logarithms to be solved.

solvetrigwarn Option variable

Default value: `true`

When `solvetrigwarn` is `true`, `solve` may print a message saying that it is using inverse trigonometric functions to solve the equation, and thereby losing solutions.

solve_inconsistent_error Option variable

Default value: `true`

When `solve_inconsistent_error` is `true`, `solve` and `linsolve` give an error if the equations to be solved are inconsistent.

If `false`, `solve` and `linsolve` return an empty list `[]` if the equations are inconsistent.

Example:

```
(%i1) solve_inconsistent_error: true$
(%i2) solve ([a + b = 1, a + b = 2], [a, b]);
Inconsistent equations:  (2)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i3) solve_inconsistent_error: false$
(%i4) solve ([a + b = 1, a + b = 2], [a, b]);
(%o4)                          []
```

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*) Function

Solves a boundary value problem for a second order differential equation. Here: *solution* is a general solution to the equation, as found by `ode2`; *xval1* specifies the value of the independent variable in a first point, in the form *x* `=` *x1*, and *yval1* gives the value of the dependent variable in that point, in the form *y* `=` *y1*. The expressions *xval2* and *yval2* give the values for these variables at a second point, using the same form.

See `ode2` for an example of its usage.

desolve (*eqn, x*) Function

desolve ([*eqn_1, ..., eqn_n*], [*x_1, ..., x_n*]) Function

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:

```
eqn_1: 'diff(f,x,2) = sin(x) + 'diff(g,x);
eqn_2: 'diff(f,x) + x^2 - f = 2*'diff(g,x,2);
```

The correct way would be:

```
eqn_1: 'diff(f(x),x,2) = sin(x) + 'diff(g(x),x);
eqn_2: 'diff(f(x),x) + x^2 - f(x) = 2*'diff(g(x),x,2);
```

The call to the function `desolve` would then be

```
desolve([eqn_1, eqn_2], [f(x),g(x)]);
```

If initial conditions at `x=0` are known, they can be supplied before calling `desolve` by using `atvalue`.

```
(%i1) 'diff(f(x),x)='diff(g(x),x)+sin(x);
                    d           d
(%o1)               -- (f(x)) = -- (g(x)) + sin(x)
                    dx          dx
(%i2) 'diff(g(x),x,2)='diff(f(x),x)-cos(x);
                     2
                    d            d
(%o2)               --- (g(x)) = -- (f(x)) - cos(x)
```

```
                    2          dx
                   dx
(%i3) atvalue('diff(g(x),x),x=0,a);
(%o3)                           a
(%i4) atvalue(f(x),x=0,1);
(%o4)                           1
(%i5) desolve([%o1,%o2],[f(x),g(x)]);
                     x
(%o5) [f(x) = a %e  - a + 1, g(x) =

                                                  x
                                       cos(x) + a %e  - a + g(0) - 1]
(%i6) [%o1,%o2],%o5,diff;
              x       x      x                x
(%o6)   [a %e  = a %e , a %e  - cos(x) = a %e  - cos(x)]
```

If `desolve` cannot obtain a solution, it returns `false`.

ic1 (*solution, xval, yval*) Function

Solves initial value problems for first order differential equations. Here *solution* is a general solution to the equation, as found by `ode2`, *xval* gives an initial value for the independent variable in the form *x* = *x0*, and *yval* gives the initial value for the dependent variable in the form *y* = *y0*.

See `ode2` for an example of its usage.

ic2 (*solution, xval, yval, dval*) Function

Solves initial value problems for second-order differential equations. Here *solution* is a general solution to the equation, as found by `ode2`, *xval* gives the initial value for the independent variable in the form *x* = *x0*, *yval* gives the initial value of the dependent variable in the form *y* = *y0*, and *dval* gives the initial value for the first derivative of the dependent variable with respect to independent variable, in the form `diff(`*y*`,`*x*`)` = *dy0* (`diff` does not have to be quoted).

See `ode2` for an example of its usage.

ode2 (*eqn, dvar, ivar*) Function

The function `ode2` solves an ordinary differential equation (ODE) of first or second order. It takes three arguments: an ODE given by *eqn*, the dependent variable *dvar*, and the independent variable *ivar*. When successful, it returns either an explicit or implicit solution for the dependent variable. `%c` is used to represent the integration constant in the case of first-order equations, and `%k1` and `%k2` the constants for second-order equations. The dependence of the dependent variable on the independent variable does not have to be written explicitly, as in the case of `desolve`, but the independent variable must always be given as the third argument.

If `ode2` cannot obtain a solution for whatever reason, it returns `false`, after perhaps printing out an error message. The methods implemented for first order equations in the order in which they are tested are: linear, separable, exact - perhaps requiring

an integrating factor, homogeneous, Bernoulli's equation, and a generalized homogeneous method. The types of second-order equations which can be solved are: constant coefficients, exact, linear homogeneous with non-constant coefficients which can be transformed to constant coefficients, the Euler or equi-dimensional equation, equations solvable by the method of variation of parameters, and equations which are free of either the independent or of the dependent variable so that they can be reduced to two first order linear equations to be solved sequentially.

In the course of solving ODE's, several variables are set purely for informational purposes: `method` denotes the method of solution used (e.g., `linear`), `intfactor` denotes any integrating factor used, `odeindex` denotes the index for Bernoulli's method or for the generalized homogeneous method, and `yp` denotes the particular solution for the variation of parameters technique.

In order to solve initial value problems (IVP) functions `ic1` and `ic2` are available for first and second order equations, and to solve second-order boundary value problems (BVP) the function `bc2` can be used.

Example:

```
(%i1) x^2*'diff(y,x) + 3*y*x = sin(x)/x;
                      2 dy           sin(x)
(%o1)                 x  -- + 3 x y = ------
                         dx            x
(%i2) ode2(%,y,x);
                              %c - cos(x)
(%o2)                      y = -----------
                                     3
                                    x
(%i3) ic1(%o2,x=%pi,y=0);
                                cos(x) + 1
(%o3)                     y = - ----------
                                     3
                                    x
(%i4) 'diff(y,x,2) + y*'diff(y,x)^3 = 0;
                           2
                          d y      dy 3
(%o4)                     --- + y (--)  = 0
                            2      dx
                          dx
(%i5) ode2(%,y,x);
                       3
                      y  + 6 %k1 y
(%o5)                 ------------ = x + %k2
                          6
(%i6) ratsimp(ic2(%o5,x=0,y=0,'diff(y,x)=2));
                             3
                            2 y  - 3 y
(%o6)                       - ---------- = x
                                 6
(%i7) bc2(%o5,x=0,y=1,x=1,y=3);
                           3
```

```
                                  y  - 10 y       3
(%o7)                             --------- = x - -
                                      6           2
```

23 Numerical

23.1 Introduction to fast Fourier transform

The `fft` package comprises functions for the numerical (not symbolic) computation of the fast Fourier transform.

23.2 Functions and Variables for fast Fourier transform

polartorect (*magnitude_array, phase_array*) Function

Translates complex values of the form `r %e^(%i t)` to the form `a + b %i`. `load ("fft")` loads this function into Maxima. See also `fft`.

The magnitude and phase, `r` and `t`, are taken from *magnitude_array* and *phase_array*, respectively. The original values of the input arrays are replaced by the real and imaginary parts, `a` and `b`, on return. The outputs are calculated as

```
a: r cos (t)
b: r sin (t)
```

The input arrays must be the same size and 1-dimensional. The array size need not be a power of 2.

`polartorect` is the inverse function of `recttopolar`.

recttopolar (*real_array, imaginary_array*) Function

Translates complex values of the form `a + b %i` to the form `r %e^(%i t)`. `load ("fft")` loads this function into Maxima. See also `fft`.

The real and imaginary parts, `a` and `b`, are taken from *real_array* and *imaginary_array*, respectively. The original values of the input arrays are replaced by the magnitude and angle, `r` and `t`, on return. The outputs are calculated as

```
r: sqrt (a^2 + b^2)
t: atan2 (b, a)
```

The computed angle is in the range `-%pi` to `%pi`.

The input arrays must be the same size and 1-dimensional. The array size need not be a power of 2.

`recttopolar` is the inverse function of `polartorect`.

ift (*real_array, imaginary_array*) Function

Fast inverse discrete Fourier transform. `load ("fft")` loads this function into Maxima.

`ift` carries out the inverse complex fast Fourier transform on 1-dimensional floating point arrays. The inverse transform is defined as

```
x[j]: sum (y[j] exp (+2 %i %pi j k / n), k, 0, n-1)
```

See `fft` for more details.

fft (*real_array*, *imaginary_array*) Function
ift (*real_array*, *imaginary_array*) Function
recttopolar (*real_array*, *imaginary_array*) Function
polartorect (*magnitude_array*, *phase_array*) Function

Fast Fourier transform and related functions. `load ("fft")` loads these functions into Maxima.

`fft` and `ift` carry out the complex fast Fourier transform and inverse transform, respectively, on 1-dimensional floating point arrays. The size of *imaginary_array* must equal the size of *real_array*.

`fft` and `ift` operate in-place. That is, on return from `fft` or `ift`, the original content of the input arrays is replaced by the output. The `fillarray` function can make a copy of an array, should it be necessary.

The discrete Fourier transform and inverse transform are defined as follows. Let `x` be the original data, with

```
x[i]: real_array[i] + %i imaginary_array[i]
```

Let `y` be the transformed data. The forward and inverse transforms are

```
y[k]: (1/n) sum (x[j] exp (-2 %i %pi j k / n), j, 0, n-1)

x[j]:       sum (y[j] exp (+2 %i %pi j k / n), k, 0, n-1)
```

Suitable arrays can be allocated by the `array` function. For example:

```
array (my_array, float, n-1)$
```

declares a 1-dimensional array with n elements, indexed from 0 through n-1 inclusive. The number of elements n must be equal to 2^m for some m.

`fft` can be applied to real data (imaginary array all zeros) to obtain sine and cosine coefficients. After calling `fft`, the sine and cosine coefficients, say `a` and `b`, can be calculated as

```
a[0]: real_array[0]
b[0]: 0
```

and

```
a[j]: real_array[j] + real_array[n-j]
b[j]: imaginary_array[j] - imaginary_array[n-j]
```

for j equal to 1 through n/2-1, and

```
a[n/2]: real_array[n/2]
b[n/2]: 0
```

`recttopolar` translates complex values of the form `a + b %i` to the form `r %e^(%i t)`. See `recttopolar`.

`polartorect` translates complex values of the form `r %e^(%i t)` to the form `a + b %i`. See `polartorect`.

`demo ("fft")` displays a demonstration of the `fft` package.

fortindent Option variable

Default value: 0

`fortindent` controls the left margin indentation of expressions printed out by the `fortran` command. 0 gives normal printout (i.e., 6 spaces), and positive values will causes the expressions to be printed farther to the right.

fortran (*expr*) Function

Prints *expr* as a Fortran statement. The output line is indented with spaces. If the line is too long, `fortran` prints continuation lines. `fortran` prints the exponentiation operator `^` as `**`, and prints a complex number `a + b %i` in the form `(a,b)`.

expr may be an equation. If so, `fortran` prints an assignment statement, assigning the right-hand side of the equation to the left-hand side. In particular, if the right-hand side of *expr* is the name of a matrix, then `fortran` prints an assignment statement for each element of the matrix.

If *expr* is not something recognized by `fortran`, the expression is printed in `grind` format without complaint. `fortran` does not know about lists, arrays, or functions.

`fortindent` controls the left margin of the printed lines. 0 is the normal margin (i.e., indented 6 spaces). Increasing `fortindent` causes expressions to be printed further to the right.

When `fortspaces` is `true`, `fortran` fills out each printed line with spaces to 80 columns.

`fortran` evaluates its arguments; quoting an argument defeats evaluation. `fortran` always returns `done`.

Examples:

```
(%i1) expr: (a + b)^12$
(%i2) fortran (expr);
      (b+a)**12
(%o2)                          done
(%i3) fortran ('x=expr);
      x = (b+a)**12
(%o3)                          done
(%i4) fortran ('x=expand (expr));
      x = b**12+12*a*b**11+66*a**2*b**10+220*a**3*b**9+495*a**4*b**8+792
     1   *a**5*b**7+924*a**6*b**6+792*a**7*b**5+495*a**8*b**4+220*a**9*b
     2   **3+66*a**10*b**2+12*a**11*b+a**12
(%o4)                          done
(%i5) fortran ('x=7+5*%i);
      x = (7,5)
(%o5)                          done
(%i6) fortran ('x=[1,2,3,4]);
      x = [1,2,3,4]
(%o6)                          done
(%i7) f(x) := x^2$
(%i8) fortran (f);
      f
(%o8)                          done
```

fortspaces Option variable

Default value: `false`

When `fortspaces` is `true`, `fortran` fills out each printed line with spaces to 80 columns.

horner (*expr*, *x*) Function
horner (*expr*) Function

Returns a rearranged representation of *expr* as in Horner's rule, using *x* as the main variable if it is specified. `x` may be omitted in which case the main variable of the canonical rational expression form of *expr* is used.

`horner` sometimes improves stability if `expr` is to be numerically evaluated. It is also useful if Maxima is used to generate programs to be run in Fortran. See also `stringout`.

```
(%i1) expr: 1e-155*x^2 - 5.5*x + 5.2e155;
                              2
(%o1)            1.0E-155 x  - 5.5 x + 5.2E+155
(%i2) expr2: horner (%, x), keepfloat: true;
(%o2)            (1.0E-155 x - 5.5) x + 5.2E+155
(%i3) ev (expr, x=1e155);
Maxima encountered a Lisp error:

 floating point overflow

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
(%i4) ev (expr2, x=1e155);
(%o4)                         7.0E+154
```

find_root (*expr*, *x*, *a*, *b*) Function
find_root (*f*, *a*, *b*) Function
find_root_error Option variable
find_root_abs Option variable
find_root_rel Option variable

Finds a root of the expression *expr* or the function *f* over the closed interval $[a, b]$. The expression *expr* may be an equation, in which case `find_root` seeks a root of `lhs(`*expr*`) - rhs(`*expr*`)`.

Given that Maxima can evaluate *expr* or *f* over $[a, b]$ and that *expr* or *f* is continuous, `find_root` is guaranteed to find the root, or one of the roots if there is more than one.

`find_root` initially applies binary search. If the function in question appears to be smooth enough, `find_root` applies linear interpolation instead.

The accuracy of `find_root` is governed by `find_root_abs` and `find_root_rel`. `find_root` stops when the function in question evaluates to something less than or equal to `find_root_abs`, or if successive approximants *x_0*, *x_1* differ by no more than `find_root_rel * max(abs(x_0), abs(x_1))`. The default values of `find_root_abs` and `find_root_rel` are both zero.

`find_root` expects the function in question to have a different sign at the endpoints of the search interval. If this condition is not met, the behavior of `find_root` is governed by `find_root_error`. When `find_root_error` is `true`, `find_root` prints an error message. Otherwise `find_root` returns the value of `find_root_error`. The default value of `find_root_error` is `true`.

If f evaluates to something other than a number at any step in the search algorithm, `find_root` returns a partially-evaluated `find_root` expression.

The order of a and b is ignored; the region in which a root is sought is $[min(a, b), max(a, b)]$.

Examples:

```
(%i1) f(x) := sin(x) - x/2;
                                          x
(%o1)                  f(x) := sin(x) - -
                                          2
(%i2) find_root (sin(x) - x/2, x, 0.1, %pi);
(%o2)                   1.895494267033981
(%i3) find_root (sin(x) = x/2, x, 0.1, %pi);
(%o3)                   1.895494267033981
(%i4) find_root (f(x), x, 0.1, %pi);
(%o4)                   1.895494267033981
(%i5) find_root (f, 0.1, %pi);
(%o5)                   1.895494267033981
(%i6) find_root (exp(x) = y, x, 0, 100);
                                x
(%o6)           find_root(%e  = y, x, 0.0, 100.0)
(%i7) find_root (exp(x) = y, x, 0, 100), y = 10;
(%o7)                   2.302585092994046
(%i8) log (10.0);
(%o8)                   2.302585092994046
```

newton (*expr*, *x*, *x_0*, *eps*) Function

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 `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 `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)                 1.2104963335033528E-4
```

```
(%i4) assume (a > 0);
(%o4)                          [a > 0]
(%i5) newton (x^2 - a^2, x, a/2, a^2/100);
(%o5)                    1.00030487804878 a
(%i6) ev (x^2 - a^2, x = %);
                                                2
(%o6)                6.098490481853958E-4 a
```

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*) Function

Returns `true` if `equal (`*x*`, `*y*`)` otherwise `false` (doesn't give an error message like `equal (x, y)` would do in this case).

remfun (*f*, *expr*) Function
remfun (*f*, *expr*, *x*) Function

`remfun (`*f*`, `*expr*`)` replaces all occurrences of *f* `(`*arg*`)` by *arg* in *expr*.

`remfun (`*f*`, `*expr*`, `*x*`)` replaces all occurrences of *f* `(`*arg*`)` by *arg* in *expr* only if *arg* contains the variable *x*.

funp (*f*, *expr*) Function
funp (*f*, *expr*, *x*) Function

`funp (`*f*`, `*expr*`)` returns `true` if *expr* contains the function *f*.

`funp (`*f*`, `*expr*`, `*x*`)` returns `true` if *expr* contains the function *f* and the variable *x* is somewhere in the argument of one of the instances of *f*.

absint (*f*, *x*, *halfplane*) Function
absint (*f*, *x*) Function
absint (*f*, *x*, *a*, *b*) Function

`absint (`*f*`, `*x*`, `*halfplane*`)` returns the indefinite integral of *f* with respect to *x* in the given halfplane (`pos`, `neg`, or `both`). *f* may contain expressions of the form `abs (x)`, `abs (sin (x))`, `abs (a) * exp (-abs (b) * abs (x))`.

`absint (`*f*`, `*x*`)` is equivalent to `absint (`*f*`, `*x*`, pos)`.

`absint (`*f*`, `*x*`, `*a*`, `*b*`)` returns the definite integral of *f* with respect to *x* from *a* to *b*. *f* may include absolute values.

fourier (*f*, *x*, *p*) Function

Returns a list of the Fourier coefficients of *f*`(`*x*`)` defined on the interval `[-p, p]`.

foursimp (*l*) Function

Simplifies `sin (n %pi)` to 0 if `sinnpiflag` is `true` and `cos (n %pi)` to `(-1)^n` if `cosnpiflag` is `true`.

sinnpiflag Option variable

Default value: `true`

See `foursimp`.

cosnpiflag Option variable

Default value: `true`

See `foursimp`.

fourexpand (*l*, *x*, *p*, *limit*) Function

Constructs and returns the Fourier series from the list of Fourier coefficients *l* up through *limit* terms (*limit* may be `inf`). *x* and *p* have same meaning as in `fourier`.

fourcos (*f*, *x*, *p*) Function

Returns the Fourier cosine coefficients for *f*(*x*) defined on `[0, p]`.

foursin (*f*, *x*, *p*) Function

Returns the Fourier sine coefficients for *f*(*x*) defined on `[0, p]`.

totalfourier (*f*, *x*, *p*) Function

Returns `fourexpand (foursimp (fourier (`*f*, *x*, *p*`)), `*x*, *p*, `'inf)`.

fourint (*f*, *x*) Function

Constructs and returns a list of the Fourier integral coefficients of *f*(*x*) defined on `[minf, inf]`.

fourintcos (*f*, *x*) Function

Returns the Fourier cosine integral coefficients for *f*(*x*) on `[0, inf]`.

fourintsin (*f*, *x*) Function

Returns the Fourier sine integral coefficients for *f*(*x*) on `[0, inf]`.

24 Arrays

24.1 Functions and Variables for Arrays

array (*name, dim_1, ..., dim_n*) Function
array (*name, type, dim_1, ..., dim_n*) Function
array ([*name_1, ..., name_m*], *dim_1, ..., 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 *dim_i*.

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

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

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

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, ..., i_n*]) Function

Evaluates *A* `[`*i_1*`, ..., `*i_n*`]`, where *A* is an array and *i_1*, ..., *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 `declared`, the number of dimensions, and the size of each dimension. The elements of the array, both bound and unbound, are returned by `listarray`.

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

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

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

Examples:

`arrayinfo` and `listarray` applied to a declared array.

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

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

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

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

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

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

```
(%i1) dd [x] (y) := y ^ x;
                                       x
```

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

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

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

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

Examples:

```
(%i1) arraymake (A, [1]);
(%o1)                             A
                                   1
(%i2) arraymake (A, [k]);
(%o2)                             A
                                   k
(%i3) arraymake (A, [i, j, 3]);
(%o3)                           A
                                 i, j, 3
(%i4) array (A, fixnum, 10);
(%o4)                              A
(%i5) fillarray (A, makelist (i^2, i, 1, 11));
(%o5)                              A
(%i6) arraymake (A, [5]);
(%o6)                             A
                                   5
(%i7) ''%;
(%o7)                             36
(%i8) L : [a, b, c, d, e];
(%o8)                      [a, b, c, d, e]
(%i9) arraymake ('L, [n]);
(%o9)                             L
                                   n
(%i10) ''%, n = 3;
(%o10)                             c
(%i11) A2 : make_array (fixnum, 10);
(%o11)           {Array:  #(0 0 0 0 0 0 0 0 0 0)}
(%i12) fillarray (A2, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
```

```
(%o12)          {Array:  #(1 2 3 4 5 6 7 8 9 10)}
(%i13) arraymake ('A2, [8]);
(%o13)                           A2
                                   8
(%i14) ''%;
(%o14)                            9
```

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

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 `j`*number*. 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.

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

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

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

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

```
(%i1) a3 : make_array (fixnum, 2, 5);
(%o1)        {Array:  #2A((0 0 0 0 0) (0 0 0 0 0))}
(%i2) fillarray (a3, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
(%o2)        {Array:  #2A((1 2 3 4 5) (6 7 8 9 10))}
(%i3) a4 : make_array (fixnum, 5, 2);
(%o3)     {Array:  #2A((0 0) (0 0) (0 0) (0 0) (0 0))}
(%i4) fillarray (a4, a3);
(%o4)     {Array:  #2A((1 2) (3 4) (5 6) (7 8) (9 10))}
```

listarray (*A*) 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.

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

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

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

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

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

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

```
(%i1) dd [x] (y) := y ^ x;
                                         x
(%o1)                      dd (y) := y
                             x
(%i2) dd [a + b];
```

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

make_array (*type, dim_1, ..., dim_n*) Function

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

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

Examples:

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

rearray (*A, dim_1, ..., dim_n*) Function

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

remarray (*A_1, ..., A_n*) Function

remarray (*all*) Function

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.

subvar (*x, i*) Function

Evaluates the subscripted expression *x*`[`*i*`]`.

`subvar` evaluates its arguments.

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

Examples:

```
(%i1) x : foo $

(%i2) i : 3 $

(%i3) subvar (x, i);
(%o3)                          foo
                                  3
(%i4) foo : [aa, bb, cc, dd, ee]$

(%i5) subvar (x, i);
(%o5)                           cc
(%i6) arraymake (x, [i]);
(%o6)                          foo
                                  3
(%i7) ''%;
(%o7)                           cc
```

use_fast_arrays Option variable

- 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 `sum (a[i]*b[i], i, 1, length(a))`. If `a` and `b` are not complex, this is the scalar product, also called the inner product or dot product, of `a` and `b`. The scalar product is defined as `conjugate(a).b` when `a` and `b` are complex; `innerproduct` in the `eigen` package provides the complex scalar product.

When the operands are more general matrices, the product is the matrix product `a` and `b`. The number of rows of `b` must equal the number of columns of `a`, and the result has number of rows equal to the number of rows of `a` and number of columns equal to the number of columns of `b`.

To distinguish `.` as an arithmetic operator from the decimal point in a floating point number, it may be necessary to leave spaces on either side. For example, `5.e3` is `5000.0` but `5 . e3` is `5` times `e3`.

There are several flags which govern the simplification of expressions involving `.`, namely `dot`, `dot0nscsimp`, `dot0simp`, `dot1simp`, `dotassoc`, `dotconstrules`, `dotdistrib`, `dotexptsimp`, `dotident`, and `dotscrules`.

25.1.2 Vectors

`vect` is a package of functions for vector analysis. `load ("vect")` loads this package, and `demo ("vect")` displays a demonstration.

The vector analysis package can combine and simplify symbolic expressions including dot products and cross products, together with the gradient, divergence, curl, and Laplacian operators. The distribution of these operators over sums or products is governed by several flags, as are various other expansions, including expansion into components in any specific orthogonal coordinate systems. There are also functions for deriving the scalar or vector potential of a field.

The `vect` package contains these functions: `vectorsimp`, `scalefactors`, `express`, `potential`, and `vectorpotential`.

Warning: the `vect` package declares the dot operator `.` to be a commutative operator.

25.1.3 eigen

The package `eigen` contains several functions devoted to the symbolic computation of eigenvalues and eigenvectors. Maxima loads the package automatically if one of the functions `eigenvalues` or `eigenvectors` is invoked. The package may be loaded explicitly as `load ("eigen")`.

`demo ("eigen")` displays a demonstration of the capabilities of this package. `batch ("eigen")` executes the same demonstration, but without the user prompt between successive computations.

The functions in the `eigen` package are `innerproduct`, `unitvector`, `columnvector`, `gramschmidt`, `eigenvalues`, `eigenvectors`, `uniteigenvectors`, and `similaritytransform`.

25.2 Functions and Variables for Matrices and Linear Algebra

addcol (*M*, *list_1*, ..., *list_n*) Function

Appends the column(s) given by the one or more lists (or matrices) onto the matrix *M*.

addrow (*M*, *list_1*, ..., *list_n*) Function

Appends the row(s) given by the one or more lists (or matrices) onto the matrix *M*.

adjoint (*M*) Function

Returns the adjoint of the matrix *M*. The adjoint matrix is the transpose of the matrix of cofactors of *M*.

augcoefmatrix ([*eqn_1*, ..., *eqn_m*], [*x_1*, ..., *x_n*]) Function

Returns the augmented coefficient matrix for the variables *x_1*, ..., *x_n* of the system of linear equations *eqn_1*, ..., *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*, ..., *x_n*).

```
(%i1) m: [2*x - (a - 1)*y = 5*b, c + b*y + a*x = 0]$
(%i2) augcoefmatrix (m, [x, y]);
                        [ 2  1 - a  - 5 b ]
(%o2)                   [                 ]
                        [ a    b      c   ]
```

charpoly (*M*, *x*) Function

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]);
                              [ 3  1 ]
(%o1)                         [      ]
                              [ 2  4 ]
(%i2) expand (charpoly (a, lambda));
                             2
(%o2)                lambda  - 7 lambda + 10
(%i3) (programmode: true, solve (%));
(%o3)               [lambda = 5, lambda = 2]
(%i4) matrix ([x1], [x2]);
```

```
                                [ x1 ]
(%o4)                           [    ]
                                [ x2 ]
(%i5) ev (a . % - lambda*%, %th(2)[1]);
                               [ x2 - 2 x1 ]
(%o5)                          [           ]
                               [ 2 x1 - x2 ]
(%i6) %[1, 1] = 0;
(%o6)                         x2 - 2 x1 = 0
(%i7) x2^2 + x1^2 = 1;
                                 2     2
(%o7)                         x2  + x1  = 1
(%i8) solve ([%th(2), %], [x1, x2]);
                     1                2
(%o8) [[x1 = - -------, x2 = - -------],
                 sqrt(5)          sqrt(5)

                                          1             2
                                   [x1 = -------, x2 = -------]]
                                         sqrt(5)       sqrt(5)
```

coefmatrix ([*eqn_1*, ..., *eqn_m*], [*x_1*, ..., *x_n*]) Function

Returns the coefficient matrix for the variables *x_1*, ..., *x_n* of the system of linear equations *eqn_1*, ..., *eqn_m*.

```
(%i1) coefmatrix([2*x-(a-1)*y+5*b = 0, b*y+a*x = 3], [x,y]);
                                  [ 2  1 - a ]
(%o1)                             [          ]
                                  [ a    b   ]
```

col (*M*, *i*) Function

Returns the *i*'th column of the matrix *M*. The return value is a matrix.

columnvector (*L*) Function

covect (*L*) Function

Returns a matrix of one column and `length` (*L*) rows, containing the elements of the list *L*.

`covect` is a synonym for `columnvector`.

`load ("eigen")` loads this function.

This is useful if you want to use parts of the outputs of the functions in this package in matrix calculations.

Example:

```
(%i1) load ("eigen")$
Warning - you are redefining the Macsyma function eigenvalues
Warning - you are redefining the Macsyma function eigenvectors
(%i2) columnvector ([aa, bb, cc, dd]);
                                  [ aa ]
                                  [    ]
```

```
                                  [ bb ]
(%o2)                             [    ]
                                  [ cc ]
                                  [    ]
                                  [ dd ]
```

conjugate (*x*) Function

Returns the complex conjugate of *x*.

```
(%i1) declare ([aa, bb], real, cc, complex, ii, imaginary);

(%o1)                           done
(%i2) conjugate (aa + bb*%i);

(%o2)                        aa - %i bb
(%i3) conjugate (cc);

(%o3)                       conjugate(cc)
(%i4) conjugate (ii);

(%o4)                           - ii
(%i5) conjugate (xx + yy);

(%o5)             conjugate(yy) + conjugate(xx)
```

copymatrix (*M*) Function

Returns a copy of the matrix *M*. This is the only way to make a copy aside from copying *M* element by element.

Note that an assignment of one matrix to another, as in `m2: m1`, does not copy `m1`. An assignment `m2 [i,j]: x` or `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*) Function

Computes the determinant of *M* by a method similar to Gaussian elimination.

The form of the result depends upon the setting of the switch `ratmx`.

There is a special routine for computing sparse determinants which is called when the switches `ratmx` and `sparse` are both `true`.

detout Option variable

Default value: `false`

When `detout` is `true`, the determinant of a matrix whose inverse is computed is factored out of the inverse.

For this switch to have an effect `doallmxops` and `doscmxops` should be `false` (see their descriptions). Alternatively this switch can be given to `ev` which causes the other two to be set correctly.

Example:

```
(%i1) m: matrix ([a, b], [c, d]);
                              [ a  b ]
(%o1)                         [      ]
                              [ c  d ]
(%i2) detout: true$
(%i3) doallmxops: false$
(%i4) doscmxops: false$
(%i5) invert (m);
                             [  d   - b ]
                             [          ]
                             [ - c   a  ]
(%o5)                        ------------
                              a d - b c
```

diagmatrix (*n*, *x*) Function

Returns a diagonal matrix of size *n* by *n* with the diagonal elements all equal to *x*. `diagmatrix (n, 1)` returns an identity matrix (same as `ident (n)`).

n must evaluate to an integer, otherwise `diagmatrix` complains with an error message.

x can be any kind of expression, including another matrix. If *x* is a matrix, it is not copied; all diagonal elements refer to the same instance, *x*.

doallmxops Option variable

Default value: `true`

When `doallmxops` is `true`, all operations relating to matrices are carried out. When it is `false` then the setting of the individual `dot` switches govern which operations are performed.

domxexpt Option variable

Default value: `true`

When `domxexpt` is `true`, a matrix exponential, `exp (M)` where *M* is a matrix, is interpreted as a matrix with element `[i,j]` equal to `exp (m[i,j])`. Otherwise `exp (M)` evaluates to `exp (`*ev(M)*.

`domxexpt` affects all expressions of the form *base*`^`*power* where *base* is an expression assumed scalar or constant, and *power* is a list or matrix.

Example:

```
(%i1) m: matrix ([1, %i], [a+b, %pi]);
                            [   1    %i  ]
(%o1)                       [            ]
                            [ b + a  %pi ]
(%i2) domxexpt: false$
(%i3) (1 - c)^m;
                               [   1    %i  ]
                               [            ]
                               [ b + a  %pi ]
(%o3)                   (1 - c)
(%i4) domxexpt: true$
```

```
(%i5) (1 - c)^m;
                    [                         %i  ]
                    [    1 - c       (1 - c)    ]
(%o5)               [                           ]
                    [         b + a          %pi ]
                    [ (1 - c)        (1 - c)    ]
```

domxmxops Option variable

Default value: `true`

When `domxmxops` is `true`, all matrix-matrix or matrix-list operations are carried out (but not scalar-matrix operations); if this switch is `false` such operations are not carried out.

domxnctimes Option variable

Default value: `false`

When `domxnctimes` is `true`, non-commutative products of matrices are carried out.

dontfactor Option variable

Default value: `[]`

`dontfactor` may be set to a list of variables with respect to which factoring is not to occur. (The list is initially empty.) Factoring also will not take place with respect to any variables which are less important, according the variable ordering assumed for canonical rational expression (CRE) form, than those on the `dontfactor` list.

doscmxops Option variable

Default value: `false`

When `doscmxops` is `true`, scalar-matrix operations are carried out.

doscmxplus Option variable

Default value: `false`

When `doscmxplus` is `true`, scalar-matrix operations yield a matrix result. This switch is not subsumed under `doallmxops`.

dot0nscsimp Option variable

Default value: `true`

When `dot0nscsimp` is `true`, a non-commutative product of zero and a nonscalar term is simplified to a commutative product.

dot0simp Option variable

Default value: `true`

When `dot0simp` is `true`, a non-commutative product of zero and a scalar term is simplified to a commutative product.

dot1simp Option variable

Default value: `true`

When `dot1simp` is `true`, a non-commutative product of one and another term is simplified to a commutative product.

dotassoc Option variable

Default value: `true`

When `dotassoc` is `true`, an expression `(A.B).C` simplifies to `A.(B.C)`.

dotconstrules Option variable

Default value: `true`

When `dotconstrules` is `true`, a non-commutative product of a constant and another term is simplified to a commutative product. Turning on this flag effectively turns on `dot0simp`, `dot0nscsimp`, and `dot1simp` as well.

dotdistrib Option variable

Default value: `false`

When `dotdistrib` is `true`, an expression `A.(B + C)` simplifies to `A.B + A.C`.

dotexptsimp Option variable

Default value: `true`

When `dotexptsimp` is `true`, an expression `A.A` simplifies to `A^^2`.

dotident Option variable

Default value: 1

`dotident` is the value returned by `X^^0`.

dotscrules Option variable

Default value: `false`

When `dotscrules` is `true`, an expression `A.SC` or `SC.A` simplifies to `SC*A` and `A.(SC*B)` simplifies to `SC*(A.B)`.

echelon (*M*) Function

Returns the echelon form of the matrix *M*, as produced by Gaussian elimination. The echelon form is computed from *M* by elementary row operations such that the first non-zero element in each row in the resulting matrix is one and the column elements under the first one in each row are all zero.

`triangularize` also carries out Gaussian elimination, but it does not normalize the leading non-zero element in each row.

`lu_factor` and `cholesky` are other functions which yield triangularized matrices.

```
(%i1) M: matrix ([3, 7, aa, bb], [-1, 8, 5, 2], [9, 2, 11, 4]);
                        [  3   7  aa  bb ]
                        [                ]
(%o1)                   [ - 1  8  5   2  ]
```

```
                          [                 ]
                          [  9   2  11  4   ]
(%i2) echelon (M);
                     [ 1  - 8  - 5      - 2     ]
                     [                          ]
                     [         28       11      ]
                     [ 0   1   --       --      ]
(%o2)                [         37       37      ]
                     [                          ]
                     [              37 bb - 119 ]
                     [ 0   0    1   ----------- ]
                     [              37 aa - 313 ]
```

eigenvalues (M) Function
eivals (M) Function

Returns a list of two lists containing the eigenvalues of the matrix M. The first sublist of the return value is the list of eigenvalues of the matrix, and the second sublist is the list of the multiplicities of the eigenvalues in the corresponding order.

`eivals` is a synonym for `eigenvalues`.

`eigenvalues` calls the function `solve` to find the roots of the characteristic polynomial of the matrix. Sometimes `solve` may not be able to find the roots of the polynomial; in that case some other functions in this package (except `innerproduct`, `unitvector`, `columnvector` and `gramschmidt`) will not work.

In some cases the eigenvalues found by `solve` may be complicated expressions. (This may happen when `solve` returns a not-so-obviously real expression for an eigenvalue which is known to be real.) It may be possible to simplify the eigenvalues using some other functions.

The package `eigen.mac` is loaded automatically when `eigenvalues` or `eigenvectors` is referenced. If `eigen.mac` is not already loaded, `load ("eigen")` loads it. After loading, all functions and variables in the package are available.

eigenvectors (M) Function
eivects (M) Function

takes a matrix M as its argument and returns a list of lists the first sublist of which is the output of `eigenvalues` and the other sublists of which are the eigenvectors of the matrix corresponding to those eigenvalues respectively.

`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 `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`.

ematrix (*m, n, x, i, j*) Function

Returns an *m* by *n* matrix, all elements of which are zero except for the `[`*i*`,` *j*`]` element which is *x*.

entermatrix (*m, n*) Function

Returns an *m* by *n* matrix, reading the elements interactively.

If *n* is equal to *m*, Maxima prompts for the type of the matrix (diagonal, symmetric, antisymmetric, or general) and for each element. Each response is terminated by a semicolon `;` or dollar sign `$`.

If *n* is not equal to *m*, Maxima prompts for each element.

The elements may be any expressions, which are evaluated. `entermatrix` evaluates its arguments.

```
(%i1) n: 3$
(%i2) m: entermatrix (n, n)$

Is the matrix  1. Diagonal  2. Symmetric  3. Antisymmetric
4. General
Answer 1, 2, 3 or 4 :
1$
Row 1 Column 1:
(a+b)^n$
Row 2 Column 2:
(a+b)^(n+1)$
Row 3 Column 3:
(a+b)^(n+2)$

Matrix entered.
(%i3) m;
                [        3                    ]
                [ (b + a)      0         0     ]
                [                              ]
(%o3)           [                  4           ]
                [    0      (b + a)      0     ]
                [                              ]
                [                           5  ]
                [    0         0      (b + a)  ]
```

genmatrix (*a*, *i_2*, *j_2*, *i_1*, *j_1*) Function
genmatrix (*a*, *i_2*, *j_2*, *i_1*) Function
genmatrix (*a*, *i_2*, *j_2*) Function

Returns a matrix generated from *a*, taking element *a*`[`*i_1*`,`*j_1*`]` as the upper-left element and *a*`[`*i_2*`,`*j_2*`]` as the lower-right element of the matrix. Here *a* is a declared array (created by `array` but not by `make_array`) or an undeclared array, or an array function, or a lambda expression of two arguments. (An array function is created like other functions with `:=` or `define`, but arguments are enclosed in square brackets instead of parentheses.)

If *j_1* is omitted, it is assumed equal to *i_1*. If both *j_1* and *i_1* are omitted, both are assumed equal to 1.

If a selected element `i,j` of the array is undefined, the matrix will contain a symbolic element *a*`[i,j]`.

Examples:

```
(%i1) h [i, j] := 1 / (i + j - 1);
                                      1
(%o1)                  h     := ---------
                        i, j    i + j - 1
(%i2) genmatrix (h, 3, 3);
                              [    1  1 ]
                              [ 1  -  - ]
                              [    2  3 ]
                              [         ]
                              [ 1  1  1 ]
(%o2)                         [ -  -  - ]
                              [ 2  3  4 ]
                              [         ]
                              [ 1  1  1 ]
                              [ -  -  - ]
                              [ 3  4  5 ]
(%i3) array (a, fixnum, 2, 2);
(%o3)                           a
(%i4) a [1, 1] : %e;
(%o4)                          %e
(%i5) a [2, 2] : %pi;
(%o5)                          %pi
(%i6) genmatrix (a, 2, 2);
                              [ %e   0  ]
(%o6)                         [         ]
                              [ 0   %pi ]
(%i7) genmatrix (lambda ([i, j], j - i), 3, 3);
                             [  0    1   2 ]
                             [             ]
(%o7)                        [ - 1   0   1 ]
                             [             ]
                             [ - 2  - 1  0 ]
(%i8) genmatrix (B, 2, 2);
                            [ B      B     ]
```

```
                          [  1, 1   1, 2 ]
(%o8)                     [              ]
                          [ B      B     ]
                          [  2, 1   2, 2 ]
```

gramschmidt (*x*) Function
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 `gramschmidt`. The inner product employed by `gramschmidt` is *F*, if present, otherwise the inner product is the function `innerproduct`.

If *x* is a matrix, the algorithm is applied to the rows of *x*. If *x* is a list of lists, the algorithm is applied to the sublists, which must have equal numbers of elements. In either case, the return value is a list of lists, the sublists of which are orthogonal and span the same space as *x*. If the dimension of the span of *x* is less than the number of rows or sublists, some sublists of the return value are zero.

`factor` is called at each stage of the algorithm to simplify intermediate results. As a consequence, the return value may contain factored integers.

`load(eigen)` loads this function.

Example:

Gram-Schmidt algorithm using default inner product function.

```
(%i1) load (eigen)$
(%i2) x: matrix ([1, 2, 3], [9, 18, 30], [12, 48, 60]);
                          [ 1   2   3  ]
                          [            ]
(%o2)                     [ 9   18  30 ]
                          [            ]
                          [ 12  48  60 ]
(%i3) y: gramschmidt (x);
                       2      2            4     3
                      3      3   3 5      2  3  2  3
(%o3)  [[1, 2, 3], [- ---, - --, ---], [- ----, ----, 0]]
                      2 7    7   2 7       5     5
(%i4) map (innerproduct, [y[1], y[2], y[3]], [y[2], y[3], y[1]]);
(%o4)                        [0, 0, 0]
```

Gram-Schmidt algorithm using a specified inner product function.

```
(%i1) load (eigen)$
(%i2) ip (f, g) := integrate (f * g, u, a, b);
(%o2)          ip(f, g) := integrate(f g, u, a, b)
(%i3) y : gramschmidt ([1, sin(u), cos(u)], ip), a= -%pi/2, b=%pi/2;
                                 %pi cos(u) - 2
(%o3)              [1, sin(u), --------------]
                                       %pi
(%i4) map (ip, [y[1], y[2], y[3]], [y[2], y[3], y[1]]), a= -%pi/2, b=%pi/2;
(%o4)                        [0, 0, 0]
```

ident (*n*) Function

Returns an *n* by *n* identity matrix.

innerproduct (*x*, *y*) Function
inprod (*x*, *y*) Function

Returns the inner product (also called the scalar product or dot product) of *x* and *y*, which are lists of equal length, or both 1-column or 1-row matrices of equal length. The return value is `conjugate (x) . y`, where `.` is the noncommutative multiplication operator.

`load ("eigen")` loads this function.

`inprod` is a synonym for `innerproduct`.

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 cre-form.

Cofactors are computed by the `determinant` function, so if `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 `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 `expand (invert (m)), detout`. If it is desirable to then divide through by the determinant this can be accomplished by `xthru (%)` or alternatively from scratch by

```
expand (adjoint (m)) / expand (determinant (m))
invert (m) := adjoint (m) / determinant (m)
```

See `^^` (noncommutative exponent) for another method of inverting a matrix.

lmxchar Option variable

Default value: `[`

`lmxchar` is the character displayed as the left delimiter of a matrix. See also `rmxchar`.

Example:

```
(%i1) lmxchar: "|"$
(%i2) matrix ([a, b, c], [d, e, f], [g, h, i]);
                               | a  b  c ]
                               |         ]
(%o2)                          | d  e  f ]
                               |         ]
                               | g  h  i ]
```

matrix (*row_1*, ..., *row_n*) Function

Returns a rectangular matrix which has the rows *row_1*, ..., *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.

```
(%i1) x: matrix ([17, 3], [-8, 11]);
                              [ 17   3  ]
(%o1)                         [         ]
                              [ - 8  11 ]
(%i2) y: matrix ([%pi, %e], [a, b]);
                              [ %pi  %e ]
(%o2)                         [         ]
                              [  a   b  ]
```

- Addition, element by element.

```
(%i3) x + y;
                        [ %pi + 17  %e + 3 ]
(%o3)                   [                  ]
                        [  a - 8    b + 11 ]
```

- Subtraction, element by element.

```
(%i4) x - y;
                        [ 17 - %pi  3 - %e ]
(%o4)                   [                  ]
                        [ - a - 8   11 - b ]
```

- Multiplication, element by element.

```
(%i5) x * y;
                          [ 17 %pi  3 %e ]
(%o5)                     [              ]
                          [ - 8 a   11 b ]
```

- Division, element by element.

```
(%i6) x / y;
                          [ 17       - 1 ]
```

```
                          [ ---  3 %e    ]
                          [ %pi          ]
(%o6)                     [              ]
                          [   8    11    ]
                          [ - -    --    ]
                          [   a    b     ]
```

- Matrix to a scalar exponent, element by element.

```
(%i7) x ^ 3;
                           [ 4913    27  ]
(%o7)                      [             ]
                           [ - 512  1331 ]
```

- Scalar base to a matrix exponent, element by element.

```
(%i8) exp(y);
                           [   %pi    %e ]
                           [ %e     %e   ]
(%o8)                      [             ]
                           [    a     b  ]
                           [  %e    %e   ]
```

- Matrix base to a matrix exponent. This is not carried out element by element.

```
(%i9) x ^ y;
                                    [ %pi  %e ]
                                    [         ]
                                    [  a   b  ]
                        [ 17   3  ]
(%o9)                   [         ]
                        [ - 8  11 ]
```

- Noncommutative matrix multiplication.

```
(%i10) x . y;
                    [ 3 a + 17 %pi  3 b + 17 %e ]
(%o10)              [                           ]
                    [ 11 a - 8 %pi  11 b - 8 %e ]
(%i11) y . x;
                  [ 17 %pi - 8 %e  3 %pi + 11 %e ]
(%o11)            [                              ]
                  [  17 a - 8 b     11 b + 3 a   ]
```

- Noncommutative matrix exponentiation. A scalar base b to a matrix power M is carried out element by element and so `b^^m` is the same as `b^m`.

```
(%i12) x ^^ 3;
                           [  3833   1719 ]
(%o12)                     [              ]
                           [ - 4584  395  ]
(%i13) %e ^^ y;
                           [   %pi    %e ]
                           [ %e     %e   ]
(%o13)                     [             ]
                           [    a     b  ]
                           [  %e    %e   ]
```

- A matrix raised to a -1 exponent with noncommutative exponentiation is the matrix inverse, if it exists.

```
(%i14) x ^^ -1;
                          [ 11      3  ]
                          [ ---  - --- ]
                          [ 211    211 ]
(%o14)                    [             ]
                          [  8    17   ]
                          [ ---   ---  ]
                          [ 211   211  ]
(%i15) x . (x ^^ -1);
                             [ 1  0 ]
(%o15)                       [      ]
                             [ 0  1 ]
```

matrixmap (*f, M*) Function

Returns a matrix with element `i,j` equal to *f*(*M*`[i,j]`).

See also `map`, `fullmap`, `fullmapl`, and `apply`.

matrixp (*expr*) Function

Returns `true` if *expr* is a matrix, otherwise `false`.

matrix_element_add Option variable

Default value: `+`

`matrix_element_add` is the operation invoked in place of addition in a matrix multiplication. `matrix_element_add` can be assigned any n-ary operator (that is, a function which handles any number of arguments). The assigned value may be the name of an operator enclosed in quote marks, the name of a function, or a lambda expression.

See also `matrix_element_mult` and `matrix_element_transpose`.

Example:

```
(%i1) matrix_element_add: "*"$
(%i2) matrix_element_mult: "^"$
(%i3) aa: matrix ([a, b, c], [d, e, f]);
                            [ a  b  c ]
(%o3)                       [         ]
                            [ d  e  f ]
(%i4) bb: matrix ([u, v, w], [x, y, z]);
                            [ u  v  w ]
(%o4)                       [         ]
                            [ x  y  z ]
(%i5) aa . transpose (bb);
                      [  u  v  w   x  y  z ]
                      [ a  b  c   a  b  c  ]
(%o5)                 [                    ]
                      [  u  v  w   x  y  z ]
                      [ d  e  f   d  e  f  ]
```

matrix_element_mult Option variable

Default value: `*`

`matrix_element_mult` is the operation invoked in place of multiplication in a matrix multiplication. `matrix_element_mult` can be assigned any binary operator. The assigned value may be the name of an operator enclosed in quote marks, the name of a function, or a lambda expression.

The dot operator `.` is a useful choice in some contexts.

See also `matrix_element_add` and `matrix_element_transpose`.

Example:

```
(%i1) matrix_element_add: lambda ([[x]], sqrt (apply ("+", x)))$
(%i2) matrix_element_mult: lambda ([x, y], (x - y)^2)$
(%i3) [a, b, c] . [x, y, z];
                          2          2          2
(%o3)         sqrt((c - z)  + (b - y)  + (a - x) )
(%i4) aa: matrix ([a, b, c], [d, e, f]);
                           [ a  b  c ]
(%o4)                      [         ]
                           [ d  e  f ]
(%i5) bb: matrix ([u, v, w], [x, y, z]);
                           [ u  v  w ]
(%o5)                      [         ]
                           [ x  y  z ]
(%i6) aa . transpose (bb);
                [             2          2          2  ]
                [ sqrt((c - w)  + (b - v)  + (a - u) ) ]
(%o6)  Col 1 = [                                       ]
                [             2          2          2  ]
                [ sqrt((f - w)  + (e - v)  + (d - u) ) ]

                          [             2          2          2  ]
                          [ sqrt((c - z)  + (b - y)  + (a - x) ) ]
                  Col 2 = [                                      ]
                          [             2          2          2  ]
                          [ sqrt((f - z)  + (e - y)  + (d - x) ) ]
```

matrix_element_transpose Option variable

Default value: `false`

`matrix_element_transpose` is the operation applied to each element of a matrix when it is transposed. `matrix_element_mult` can be assigned any unary operator. The assigned value may be the name of an operator enclosed in quote marks, the name of a function, or a lambda expression.

When `matrix_element_transpose` equals `transpose`, the `transpose` function is applied to every element. When `matrix_element_transpose` equals `nonscalars`, the `transpose` function is applied to every nonscalar element. If some element is an atom, the `nonscalars` option applies `transpose` only if the atom is declared nonscalar, while the `transpose` option always applies `transpose`.

The default value, `false`, means no operation is applied.

See also `matrix_element_add` and `matrix_element_mult`.

Examples:

```
(%i1) declare (a, nonscalar)$
(%i2) transpose ([a, b]);
                          [ transpose(a) ]
(%o2)                     [              ]
                          [      b       ]
(%i3) matrix_element_transpose: nonscalars$
(%i4) transpose ([a, b]);
                          [ transpose(a) ]
(%o4)                     [              ]
                          [      b       ]
(%i5) matrix_element_transpose: transpose$
(%i6) transpose ([a, b]);
                          [ transpose(a) ]
(%o6)                     [              ]
                          [ transpose(b) ]
(%i7) matrix_element_transpose: lambda ([x], realpart(x)
      - %i*imagpart(x))$
(%i8) m: matrix ([1 + 5*%i, 3 - 2*%i], [7*%i, 11]);
                        [ 5 %i + 1  3 - 2 %i ]
(%o8)                   [                    ]
                        [   7 %i       11    ]
(%i9) transpose (m);
                         [ 1 - 5 %i  - 7 %i ]
(%o9)                    [                  ]
                         [ 2 %i + 3    11   ]
```

mattrace (*M*) Function

Returns the trace (that is, the sum of the elements on the main diagonal) of the square matrix *M*.

`mattrace` is called by `ncharpoly`, an alternative to Maxima's `charpoly`.

`load ("nchrpl")` loads this function.

minor (*M, i, j*) Function

Returns the *i*, *j* minor of the matrix *M*. That is, *M* with row *i* and column *j* removed.

ncexpt (*a, b*) Function

If a non-commutative exponential expression is too wide to be displayed as `a^^b` it appears as `ncexpt (a,b)`.

`ncexpt` is not the name of a function or operator; the name only appears in output, and is not recognized in input.

ncharpoly (*M, x*) Function

Returns the characteristic polynomial of the matrix *M* with respect to *x*. This is an alternative to Maxima's `charpoly`.

`ncharpoly` works by computing traces of powers of the given matrix, which are known to be equal to sums of powers of the roots of the characteristic polynomial. From these quantities the symmetric functions of the roots can be calculated, which are nothing more than the coefficients of the characteristic polynomial. `charpoly` works by forming the determinant of *x* `* ident [n] - a`. Thus `ncharpoly` wins, for example, in the case of large dense matrices filled with integers, since it avoids polynomial arithmetic altogether.

`load ("nchrpl")` loads this file.

newdet (*M*, *n*) Function

Computes the determinant of the matrix or array *M* by the Johnson-Gentleman tree minor algorithm. The argument *n* is the order; it is optional if *M* is a matrix.

nonscalar Declaration

Makes atoms behave as does a list or matrix with respect to the dot operator.

nonscalarp (*expr*) Function

Returns `true` if *expr* is a non-scalar, i.e., it contains atoms declared as non-scalars, lists, or matrices.

permanent (*M*, *n*) Function

Computes the permanent of the matrix *M*. A permanent is like a determinant but with no sign changes.

rank (*M*) Function

Computes the rank of the matrix *M*. That is, the order of the largest non-singular subdeterminant of *M*.

rank may return the wrong answer if it cannot determine that a matrix element that is equivalent to zero is indeed so.

ratmx Option variable

Default value: `false`

When `ratmx` is `false`, determinant and matrix addition, subtraction, and multiplication are performed in the representation of the matrix elements and cause the result of matrix inversion to be left in general representation.

When `ratmx` is `true`, the 4 operations mentioned above are performed in CRE form and the result of matrix inverse is in CRE form. Note that this may cause the elements to be expanded (depending on the setting of `ratfac`) which might not always be desired.

row (*M*, *i*) Function

Returns the *i*'th row of the matrix *M*. The return value is a matrix.

scalarmatrixp Option variable

Default value: `true`

When `scalarmatrixp` is `true`, then whenever a 1 x 1 matrix is produced as a result of computing the dot product of matrices it is simplified to a scalar, namely the sole element of the matrix.

When `scalarmatrixp` is `all`, then all 1 x 1 matrices are simplified to scalars.

When `scalarmatrixp` is `false`, 1 x 1 matrices are not simplified to scalars.

scalefactors (*coordinatetransform*) Function

Here coordinatetransform evaluates to the form [[expression1, expression2, ...], indeterminate1, indeterminat2, ...], where indeterminate1, indeterminate2, etc. are the curvilinear coordinate variables and where a set of rectangular Cartesian components is given in terms of the curvilinear coordinates by [expression1, expression2, ...]. `coordinates` is set to the vector [indeterminate1, indeterminate2,...], and `dimension` is set to the length of this vector. SF[1], SF[2], ..., SF[DIMENSION] are set to the coordinate scale factors, and `sfprod` is set to the product of these scale factors. Initially, `coordinates` is [X, Y, Z], `dimension` is 3, and SF[1]=SF[2]=SF[3]=SFPROD=1, corresponding to 3-dimensional rectangular Cartesian coordinates. To expand an expression into physical components in the current coordinate system, there is a function with usage of the form

setelmx (*x*, *i*, *j*, *M*) Function

Assigns *x* to the (*i*, *j*)'th element of the matrix *M*, and returns the altered matrix.

M `[`*i*`,` *j*`]:` *x* has the same effect, but returns *x* instead of *M*.

similaritytransform (*M*) Function
simtran (*M*) Function

`similaritytransform` computes a similarity transform of the matrix `M`. It returns a list which is the output of the `uniteigenvectors` command. In addition if the flag `nondiagonalizable` is `false` two global matrices `leftmatrix` and `rightmatrix` are computed. These matrices have the property that `leftmatrix . `*M*` . rightmatrix` is a diagonal matrix with the eigenvalues of *M* on the diagonal. If `nondiagonalizable` is `true` the left and right matrices are not computed.

If the flag `hermitianmatrix` is `true` then `leftmatrix` is the complex conjugate of the transpose of `rightmatrix`. Otherwise `leftmatrix` is the inverse of `rightmatrix`.

`rightmatrix` is the matrix the columns of which are the unit eigenvectors of *M*. The other flags (see `eigenvalues` and `eigenvectors`) have the same effects since `similaritytransform` calls the other functions in the package in order to be able to form `rightmatrix`.

`load ("eigen")` loads this function.

`simtran` is a synonym for `similaritytransform`.

sparse Option variable

Default value: `false`

When `sparse` is `true`, and if `ratmx` is `true`, then `determinant` will use special routines for computing sparse determinants.

submatrix (*i_1, ..., i_m, M, j_1, ..., j_n*) Function
submatrix (*i_1, ..., i_m, M*) Function
submatrix (*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 ]
                          [                ]
(%o1)                     [ - 1  8  5   2  ]
                          [                ]
                          [  9   2  11  4  ]
(%i2) triangularize (M);
               [ - 1   8         5            2      ]
               [                                     ]
(%o2)          [  0   - 74     - 56         - 22     ]
               [                                     ]
               [  0    0    626 - 74 aa  238 - 74 bb ]
```

uniteigenvectors (*M*) Function
ueivects (*M*) Function

Computes unit eigenvectors of the matrix *M*. The return value is a list of lists, the first sublist of which is the output of the `eigenvalues` command, and the other sublists of which are the unit eigenvectors of the matrix corresponding to those eigenvalues respectively.

The flags mentioned in the description of the `eigenvectors` command have the same effects in this one as well.

When `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`.

unitvector (*x*) Function
uvect (*x*) Function

Returns $x/norm(x)$; this is a unit vector in the same direction as *x*.

`load ("eigen")` loads this function.

`uvect` is a synonym for `unitvector`.

vectorsimp (*expr*) Function

Applies simplifications and expansions according to the following global flags:

`expandall`, `expanddot`, `expanddotplus`, `expandcross`, `expandcrossplus`, `expandcrosscross`, `expandgrad`, `expandgradplus`, `expandgradprod`, `expanddiv`, `expanddivplus`, `expanddivprod`, `expandcurl`, `expandcurlplus`, `expandcurlcurl`, `expandlaplacian`, `expandlaplacianplus`, and `expandlaplacianprod`.

All these flags have default value `false`. The `plus` suffix refers to employing additivity or distributivity. The `prod` suffix refers to the expansion for an operand that is any kind of product.

`expandcrosscross`
: Simplifies $p\ (q\ r)$ to $(p.r) * q - (p.q) * r$.

`expandcurlcurl`
: Simplifies *curlcurlp* to $graddivp + divgradp$.

`expandlaplaciantodivgrad`
: Simplifies *laplacianp* to *divgradp*.

`expandcross`
: Enables `expandcrossplus` and `expandcrosscross`.

`expandplus`
: Enables `expanddotplus`, `expandcrossplus`, `expandgradplus`, `expanddivplus`, `expandcurlplus`, and `expandlaplacianplus`.

`expandprod`
: Enables `expandgradprod`, `expanddivprod`, and `expandlaplacianprod`.

These flags have all been declared `evflag`.

vect_cross Option variable

Default value: `false`

When `vect_cross` is `true`, it allows DIFF(X~Y,T) to work where ~ is defined in SHARE;VECT (where VECT_CROSS is set to `true`, anyway.)

zeromatrix (*m*, *n*) Function

Returns an *m* by *n* matrix, all elements of which are zero.

[Special symbol
] Special symbol

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

[and] 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 ([*expr_1, ..., expr_m*], [*x_1, ..., x_n*]) Function

Solves the simultaneous linear equations *expr_1*, ..., *expr_m* for the variables *x_1*, ..., *x_n*. Each *expr_i* may be an equation or a general expression; if given as a general expression, it is treated as an equation of the form *expr_i* `= 0`.

The return value is a list of equations of the form `[`*x_1* `=` *a_1*`, ..., `*x_n* `=` *a_n*`]` where *a_1*, ..., *a_n* are all free of *x_1*, ..., *x_n*.

`fast_linsolve` is faster than `linsolve` for system of equations which are sparse.

`load(affine)` loads this function.

grobner_basis ([*expr_1, ..., expr_m*]) Function

Returns a Groebner basis for the equations *expr_1*, ..., *expr_m*. The function `polysimp` can then be used to simplify other functions relative to the equations.

```
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 *expr_1*, ..., *expr_m*, that is, if and only if *f* is a polynomial combination of the elements of *expr_1*, ..., *expr_m*.

`load(affine)` loads this function.

set_up_dot_simplifications (*eqns, check_through_degree*) Function

set_up_dot_simplifications (*eqns*) Function

The *eqns* are polynomial equations in non commutative variables. The value of `current_variables` is the list of variables used for computing degrees. The equations must be homogeneous, in order for the procedure to terminate.

If you have checked overlapping simplifications in `dot_simplifications` above the degree of *f*, then the following is true: `dotsimp (`*f*`)` yields 0 if and only if *f* is in the ideal generated by the equations, i.e., if and only if *f* is a polynomial combination of the elements of the equations.

The degree is that returned by `nc_degree`. This in turn is influenced by the weights of individual variables.

`load(affine)` loads this function.

declare_weights (*x_1, w_1, ..., x_n, w_n*) Function

Assigns weights *w_1*, ..., *w_n* to *x_1*, ..., *x_n*, respectively. These are the weights used in computing `nc_degree`.

`load(affine)` loads this function.

nc_degree (*p*) Function

Returns the degree of a noncommutative polynomial *p*. See `declare_weights`.

`load(affine)` loads this function.

dotsimp (*f*) Function

Returns 0 if and only if *f* is in the ideal generated by the equations, i.e., if and only if *f* is a polynomial combination of the elements of the equations.

`load(affine)` loads this function.

fast_central_elements ([*x_1, ..., x_n*], *n*) Function

If `set_up_dot_simplifications` has been previously done, finds the central polynomials in the variables *x_1*, ..., *x_n* in the given degree, *n*.

For example:

```
set_up_dot_simplifications ([y.x + x.y], 3);
fast_central_elements ([x, y], 2);
[y.y, x.x];
```

`load(affine)` loads this function.

check_overlaps (*n, add_to_simps*) Function

Checks the overlaps thru degree *n*, making sure that you have sufficient simplification rules in each degree, for `dotsimp` to work correctly. This process can be speeded up if you know before hand what the dimension of the space of monomials is. If it is of finite global dimension, then `hilbert` should be used. If you don't know the monomial dimensions, do not specify a `rank_function`. An optional third argument `reset`, `false` says don't bother to query about resetting things.

`load(affine)` loads this function.

mono ([*x_1, ..., x_n*], *n*) Function

Returns the list of independent monomials relative to the current dot simplifications of degree *n* in the variables *x_1*, ..., *x_n*.

`load(affine)` loads this function.

monomial_dimensions (*n*) Function

Compute the Hilbert series through degree *n* for the current algebra.

`load(affine)` loads this function.

extract_linear_equations ([*p_1, ..., p_n*], [*m_1, ..., m_n*]) Function

Makes a list of the coefficients of the noncommutative polynomials *p_1*, ..., *p_n* of the noncommutative monomials *m_1*, ..., *m_n*. The coefficients should be scalars. Use `list_nc_monomials` to build the list of monomials.

`load(affine)` loads this function.

list_nc_monomials ([*p_1*, ..., *p_n*]) Function
list_nc_monomials (*p*) Function

Returns a list of the non commutative monomials occurring in a polynomial *p* or a list of polynomials *p_1*, ..., *p_n*.

`load(affine)` loads this function.

all_dotsimp_denoms Option variable

Default value: `false`

When `all_dotsimp_denoms` is a list, the denominators encountered by `dotsimp` are appended to the list. `all_dotsimp_denoms` may be initialized to an empty list `[]` before calling `dotsimp`.

By default, denominators are not collected by `dotsimp`.

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 the special indicial representation in `itensor`, in some cases the user may find difficulty with the specification of the metric, function definition, and the evaluation of differentiated "indexed" objects.

The `itensor` package can carry out differentiation with respect to an indexed variable, which allows one to use the package when dealing with Lagrangian and Hamiltonian formalisms. As it is possible to differentiate a field Lagrangian with respect to an (indexed) field variable, one can use Maxima to derive the corresponding Euler-Lagrange equations in indicial form. These equations can be translated into component tensor (`ctensor`) programs using the `ic_convert` function, allowing us to solve the field equations in a particular coordinate representation, or to recast the equations of motion in Hamiltonian form. See `einhil.dem` and `bradic.dem` for two comprehensive examples. The first, `einhil.dem`, uses the Einstein-Hilbert action to derive the Einstein field tensor in

the homogeneous and isotropic case (Friedmann equations) and the spherically symmetric, static case (Schwarzschild solution.) The second, `bradic.dem`, demonstrates how to compute the Friedmann equations from the action of Brans-Dicke gravity theory, and also derives the Hamiltonian associated with the theory's scalar field.

27.1.1 New tensor notation

Earlier versions of the `itensor` package in Maxima used a notation that sometimes led to incorrect index ordering. Consider the following, for instance:

```
(%i2) imetric(g);
(%o2)                                done
(%i3) ishow(g([],[j,k])*g([],[i,l])*a([i,j],[]))$
                                  i l  j k
(%t3)                            g    g    a
                                           i j
(%i4) ishow(contract(%))$
                                   k l
(%t4)                              a
```

This result is incorrect unless `a` happens to be a symmetric tensor. The reason why this happens is that although `itensor` correctly maintains the order within the set of covariant and contravariant indices, once an index is raised or lowered, its position relative to the other set of indices is lost.

To avoid this problem, a new notation has been developed that remains fully compatible with the existing notation and can be used interchangeably. In this notation, contravariant indices are inserted in the appropriate positions in the covariant index list, but with a minus sign prepended. Functions like `contract` and `ishow` are now aware of this new index notation and can process tensors appropriately.

In this new notation, the previous example yields a correct result:

```
(%i5) ishow(g([-j,-k],[])*g([-i,-l],[])*a([i,j],[]))$
                                  i l       j k
(%t5)                            g    a    g
                                        i j
(%i6) ishow(contract(%))$
                                        l k
(%t6)                              a
```

Presently, the only code that makes use of this notation is the `lc2kdt` function. Through this notation, it achieves consistent results as it applies the metric tensor to resolve Levi-Civita symbols without resorting to numeric indices.

Since this code is brand new, it probably contains bugs. While it has been tested to make sure that it doesn't break anything using the "old" tensor notation, there is a considerable chance that "new" tensors will fail to interoperate with certain functions or features. These bugs will be fixed as they are encountered... until then, caveat emptor!

27.1.2 Indicial tensor manipulation

The indicial tensor manipulation package may be loaded by `load(itensor)`. Demos are also available: try `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
                                       k l
(%i5) ishow(diff(v([i],[]),t))$
(%t5)                                  0
(%i6) depends(v,t);
```

```
(%o6)                                [v(t)]
(%i7) ishow(diff(v([i],[]),t))$
                                     d
(%t7)                                -- (v )
                                     dt   i
(%i8) ishow(idiff(v([i],[]),j))$
(%t8)                                 v
                                       i,j
(%i9) ishow(extdiff(v([i],[]),j))$
(%t9)                              v    - v
                                    j,i    i,j
                                   -----------
                                       2
(%i10) ishow(liediff(v,w([i],[])))$
                                %3          %3
(%t10)                         v   w     + v   w
                                    i,%3    ,i  %3
(%i11) ishow(covdiff(v([i],[]),j))$
                                                %4
(%t11)                         v    - v   ichr2
                                i,j    %4      i j
(%i12) ishow(ev(%,ichr2))$
                 %4 %5
(%t12) v    - (g      v   (e p       + e   p     - e p       - e    p
        i,j            %4     j %5,i    ,i  j %5      i j,%5    ,%5  i j

                                          + e p       + e   p    ))/2
                                               i %5,j    ,j  i %5
(%i13) iframe_flag:true;
(%o13)                                  true
(%i14) ishow(covdiff(v([i],[]),j))$
                                              %6
(%t14)                         v    - v   icc2
                                i,j    %6     i j
(%i15) ishow(ev(%,icc2))$
                                              %6
(%t15)                         v    - v   ifc2
                                i,j    %6     i j
(%i16) ishow(radcan(ev(%,ifc2,ifc1)))$
              %6 %7                     %6 %7
(%t16) - (ifg      v   ifb       + ifg      v   ifb       - 2 v
                    %6    j %7 i             %6    i j %7      i,j

                                              %6 %7
                                        - ifg      v   ifb      )/2
                                                    %6    %7 i j
(%i17) ishow(canform(s([i,j],[])-s([j,i])))$
(%t17)                             s    - s
                                    i j    j i
```

```
(%i18) decsym(s,2,0,[sym(all)],[]);
(%o18)                                done
(%i19) ishow(canform(s([i,j],[])-s([j,i])))$
(%t19)                                 0
(%i20) ishow(canform(a([i,j],[])+a([j,i])))$
(%t20)                              a    + a
                                     j i    i j
(%i21) decsym(a,2,0,[anti(all)],[]);
(%o21)                                done
(%i22) ishow(canform(a([i,j],[])+a([j,i])))$
(%t22)                                 0
```

27.2 Functions and Variables for itensor

27.2.1 Managing indexed objects

entertensor (*name*) Function

is a function which, by prompting, allows one to create an indexed object called *name* with any number of tensorial and derivative indices. Either a single index or a list of indices (which may be null) is acceptable input (see the example under `covdiff`).

changename (*old, new, expr*) Function

will change the name of all indexed objects called *old* to *new* in *expr*. *old* may be either a symbol or a list of the form `[`*name*`,` *m*`,` *n*`]` in which case only those indexed objects called *name* with *m* covariant and *n* contravariant indices will be renamed to *new*.

listoftens Function

Lists all tensors in a tensorial expression, complete with their indices. E.g.,

```
(%i6) ishow(a([i,j],[k])*b([u],[],v)+c([x,y],[])*d([],[])*e)$
                                          k
(%t6)                          d e c    + a    b
                                    x y    i j  u,v
(%i7) ishow(listoftens(%))$
                                 k
(%t7)                           [a   , b   , c   , d]
                                  i j   u,v   x y
```

ishow (*expr*) Function

displays *expr* with the indexed objects in it shown having their covariant indices as subscripts and contravariant indices as superscripts. The derivative indices are displayed as subscripts, separated from the covariant indices by a comma (see the examples throughout this document).

indices (*expr*) Function

Returns a list of two elements. The first is a list of the free indices in *expr* (those that occur only once). The second is the list of the dummy indices in *expr* (those that occur exactly twice) as the following example demonstrates.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(a([i,j],[k,l],m,n)*b([k,o],[j,m,p],q,r))$
                                k l      j m p
(%t2)                          a        b
                                i j,m n  k o,q r
(%i3) indices(%);
(%o3)                 [[l, p, i, n, o, q, r], [k, j, m]]
```

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*) Function

rename (*expr*, *count*) Function

Returns an expression equivalent to *expr* but with the dummy indices in each term chosen from the set `[%1, %2,...]`, if the optional second argument is omitted. Otherwise, the dummy indices are indexed beginning at the value of *count*. Each dummy index in a product will be different. For a sum, `rename` will operate upon each term in the sum resetting the counter with each term. In this way `rename` can serve as a tensorial simplifier. In addition, the indices will be sorted alphanumerically (if `allsym` is `true`) with respect to covariant or contravariant indices depending upon the value of `flipflag`. If `flipflag` is `false` then the indices will be renamed according to the order of the contravariant indices. If `flipflag` is `true` the renaming will occur according to the order of the covariant indices. It often happens that the combined effect of the two renamings will reduce an expression more than either one by itself.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) allsym:true;
(%o2)                                true
(%i3) g([],[%4,%5])*g([],[%6,%7])*ichr2([%1,%4],[%3])*
ichr2([%2,%3],[u])*ichr2([%5,%6],[%1])*ichr2([%7,r],[%2])-
g([],[%4,%5])*g([],[%6,%7])*ichr2([%1,%2],[u])*
ichr2([%3,%5],[%1])*ichr2([%4,%6],[%3])*ichr2([%7,r],[%2]),noeval$
(%i4) expr:ishow(%)$

       %4 %5  %6 %7      %3         u          %1         %2
(%t4) g      g      ichr2      ichr2      ichr2      ichr2
                         %1 %4      %2 %3      %5 %6      %7 r
```

```
        %4 %5  %6 %7      u          %1         %3         %2
     - g      g      ichr2      ichr2      ichr2      ichr2
                          %1 %2      %3 %5      %4 %6      %7 r
(%i5) flipflag:true;
(%o5)                                 true
(%i6) ishow(rename(expr))$
       %2 %5  %6 %7      %4         u          %1         %3
(%t6) g      g      ichr2      ichr2      ichr2      ichr2
                          %1 %2      %3 %4      %5 %6      %7 r

        %4 %5  %6 %7      u          %1         %3         %2
     - g      g      ichr2      ichr2      ichr2      ichr2
                          %1 %2      %3 %4      %5 %6      %7 r
(%i7) flipflag:false;
(%o7)                                 false
(%i8) rename(%th(2));
(%o8)                                   0
(%i9) ishow(rename(expr))$
       %1 %2  %3 %4      %5         %6         %7         u
(%t9) g      g      ichr2      ichr2      ichr2     ichr2
                          %1 %6      %2 %3      %4 r       %5 %7

        %1 %2  %3 %4      %6         %5         %7         u
     - g      g      ichr2      ichr2      ichr2     ichr2
                          %1 %3      %2 %6      %4 r       %5 %7
```

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*) Function

defcon (*tensor_1, tensor_2, tensor_3*) Function

gives *tensor_1* the property that the contraction of a product of *tensor_1* and *tensor_2* results in *tensor_3* with the appropriate indices. If only one argument, *tensor_1*, is given, 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 (*tensor_1, ..., tensor_n*) Function
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 (*expr*) Function

Carries out the tensorial contractions in *expr* which may be any combination of sums and products. This function uses the information given to the `defcon` function. For best results, `expr` should be fully expanded. `ratexpand` is the fastest way to expand products and powers of sums if there are no variables in the denominators of the terms. The `gcd` switch should be `false` if GCD cancellations are unnecessary.

indexed_tensor (*tensor*) Function

Must be executed before assigning components to a *tensor* for which a built in value already exists as with `ichr1`, `ichr2`, `icurvature`. See the example under `icurvature`.

components (*tensor, expr*) Function

permits one to assign an indicial value to an expression *expr* giving the values of the components of *tensor*. These are automatically substituted for the tensor whenever it occurs with all of its indices. The tensor must be of the form `t([...],[...])` where either list may be empty. *expr* can be any indexed expression involving other objects with the same free indices as *tensor*. When used to assign values to the metric tensor wherein the components contain dummy indices one must be careful to define these indices to avoid the generation of multiple dummy indices. Removal of this assignment is given to the function `remcomps`.

It is important to keep in mind that `components` cares only about the valence of a tensor, not about any particular index ordering. Thus assigning components to, say, `x([i,-j],[])`, `x([-j,i],[])`, or `x([i],[j])` all produce the same result, namely components being assigned to a tensor named `x` with valence `(1,1)`.

Components can be assigned to an indexed expression in four ways, two of which involve the use of the `components` command:

1) As an indexed expression. For instance:

```
(%i2) components(g([],[i,j]),e([],[i])*p([],[j]))$
(%i3) ishow(g([],[i,j]))$
                                        i  j
(%t3)                                  e  p
```

2) As a matrix:

```
(%i5) lg:-ident(4)$lg[1,1]:1$lg;
                              [ 1   0    0    0  ]
                              [                  ]
                              [ 0  - 1   0    0  ]
(%o5)                         [                  ]
                              [ 0   0   - 1   0  ]
```

```
                              [                  ]
                              [ 0   0    0   - 1 ]

(%i6) components(g([i,j],[]),lg);
(%o6)                                done
(%i7) ishow(g([i,j],[]))$
(%t7)                                g
                                      i j
(%i8) g([1,1],[]);
(%o8)                                 1
(%i9) g([4,4],[]);
(%o9)                                 - 1
```

3) As a function. You can use a Maxima function to specify the components of a tensor based on its indices. For instance, the following code assigns `kdelta` to `h` if `h` has the same number of covariant and contravariant indices and no derivative indices, and `g` otherwise:

```
(%i4) h(l1,l2,[l3]):=if length(l1)=length(l2) and length(l3)=0
  then kdelta(l1,l2) else apply(g,append([l1,l2], l3))$
(%i5) ishow(h([i],[j]))$
                                           j
(%t5)                                kdelta
                                           i
(%i6) ishow(h([i,j],[k],l))$
                                      k
(%t6)                                g
                                      i j,l
```

4) Using Maxima's pattern matching capabilities, specifically the `defrule` and `applyb1` commands:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) matchdeclare(l1,listp);
(%o2)                                done
(%i3) defrule(r1,m(l1,[]),(i1:idummy(),
      g([l1[1],l1[2]],[])*q([i1],[])*e([],[i1])))$

(%i4) defrule(r2,m([],l1),(i1:idummy(),
      w([],[l1[1],l1[2]])*e([i1],[])*q([],[i1])))$

(%i5) ishow(m([i,n],[])*m([],[i,m]))$
                                     i m
(%t5)                               m    m
                                         i n
(%i6) ishow(rename(applyb1(%,r1,r2)))$
                             %1  %2  %3 m
```

```
(%t6)                    e   q   w     q   e   g
                                        %1  %2  %3 n
```

remcomps (*tensor*) Function

Unbinds all values from *tensor* which were assigned with the `components` function.

showcomps (*tensor*) Function

Shows component assignments of a tensor, as made using the `components` command. This function can be particularly useful when a matrix is assigned to an indicial tensor using `components`, as demonstrated by the following example:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) load(itensor);
(%o2)      /share/tensor/itensor.lisp
(%i3) lg:matrix([sqrt(r/(r-2*m)),0,0,0],[0,r,0,0],
                [0,0,sin(theta)*r,0],[0,0,0,sqrt((r-2*m)/r)]);
               [         r                                    ]
               [ sqrt(-------)  0       0              0      ]
               [      r - 2 m                                 ]
               [                                              ]
               [       0        r       0              0      ]
(%o3)          [                                              ]
               [       0        0  r sin(theta)        0      ]
               [                                              ]
               [                                      r - 2 m ]
               [       0        0       0        sqrt(-------) ]
               [                                        r     ]
(%i4) components(g([i,j],[]),lg);
(%o4)                           done
(%i5) showcomps(g([i,j],[]));
                  [         r                                  ]
                  [ sqrt(-------)  0       0              0    ]
                  [      r - 2 m                               ]
                  [                                            ]
                  [       0        r       0              0    ]
(%t5)      g    = [                                            ]
            i j   [       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([i,j],[])` is a valid tensorial object.

kdels (*L1, L2*) Function

Symmetricized Kronecker delta, used in some calculations. For instance:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) kdelta([1,2],[2,1]);
(%o2)                                - 1
(%i3) kdels([1,2],[2,1]);
(%o3)                                 1
(%i4) ishow(kdelta([a,b],[c,d]))$
                             c       d         d       c
(%t4)                 kdelta  kdelta  - kdelta  kdelta
                             a       b         a       b
(%i4) ishow(kdels([a,b],[c,d]))$
                             c       d         d       c
(%t4)                 kdelta  kdelta  + kdelta  kdelta
                             a       b         a       b
```

levi_civita (*L*) Function

is the permutation (or Levi-Civita) tensor which yields 1 if the list *L* consists of an even permutation of integers, -1 if it consists of an odd permutation, and 0 if some indices in *L* are repeated.

lc2kdt (*expr*) Function

Simplifies expressions containing the Levi-Civita symbol, converting these to Kronecker-delta expressions when possible. The main difference between this function and simply evaluating the Levi-Civita symbol is that direct evaluation often results in Kronecker expressions containing numerical indices. This is often undesirable as it prevents further simplification. The `lc2kdt` function avoids this problem, yielding expressions that are more easily simplified with `rename` or `contract`.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) expr:ishow('levi_civita([],[i,j])
                *'levi_civita([k,l],[])*a([j],[k]))$
                                      i j  k
(%t2)                  levi_civita    a  levi_civita
                                       j            k l
(%i3) ishow(ev(expr,levi_civita))$
                                      i j  k       1 2
(%t3)                        kdelta    a  kdelta
                                      1 2  j       k l
(%i4) ishow(ev(%,kdelta))$
              i       j         j       i   k
(%t4) (kdelta  kdelta  - kdelta  kdelta ) a
              1       2         1       2   j

                                  1       2         2       1
                           (kdelta  kdelta  - kdelta  kdelta )
                                  k       l         k       l
(%i5) ishow(lc2kdt(expr))$
                      k       i       j    k       j       i
(%t5)                a  kdelta  kdelta  - a  kdelta  kdelta
                      j       k       l    j       k       l
(%i6) ishow(contract(expand(%)))$
                                    i           i
(%t6)                              a  - a kdelta
                                    l           l
```

The `lc2kdt` function sometimes makes use of the metric tensor. If the metric tensor was not defined previously with `imetric`, this results in an error.

```
(%i7) expr:ishow('levi_civita([],[i,j])
                *'levi_civita([],[k,l])*a([j,k],[]))$
                                      i j            k l
```

```
(%t7)                 levi_civita    levi_civita    a
                                                           j k
(%i8) ishow(lc2kdt(expr))$
Maxima encountered a Lisp error:

 Error in $IMETRIC [or a callee]:
 $IMETRIC [or a callee] requires less than two arguments.

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
(%i9) imetric(g);
(%o9)                                 done
(%i10) ishow(lc2kdt(expr))$
         %3 i       k   %4 j       l     %3 i       l   %4 j
(%t10) (g     kdelta   g     kdelta   - g     kdelta   g
                     %3             %4               %3
               k
        kdelta  ) a
               %4   j k
(%i11) ishow(contract(expand(%)))$
                                  l i    l i  j
(%t11)                           a    - g    a
                                             j
```

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([],[i])*a([],[j]))$
                             i  j
(%t2)                       a  a  levi_civita
                                              i j k
(%i3) el2:ishow('levi_civita([],[i,j,k])*a([i])*a([j]))$
                                        i j k
(%t3)                       levi_civita      a  a
                                              i  j
(%i4) canform(contract(expand(applyb1(el1,lc_l,lc_u))));
(%t4)                                   0
(%i5) canform(contract(expand(applyb1(el2,lc_l,lc_u))));
(%t5)                                   0
```

lc_u Function

Simplification rule used for expressions containing the unevaluated Levi-Civita symbol (`levi_civita`). Along with `lc_u`, it can be used to simplify many expressions more efficiently than the evaluation of `levi_civita`. For details, see `lc_l`.

canten (*expr*) Function

Simplifies *expr* by renaming (see `rename`) and permuting dummy indices. `rename` is restricted to sums of tensor products in which no derivatives are present. As such it is limited and should only be used if `canform` is not capable of carrying out the required simplification.

The `canten` function returns a mathematically correct result only if its argument is an expression that is fully symmetric in its indices. For this reason, `canten` returns an error if `allsym` is not set to `true`.

concan (*expr*) Function

Similar to `canten` but also performs index contraction.

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*, ...]) Function

Declares symmetry properties for *tensor* of *m* covariant and *n* contravariant indices. The *cov_i* and *contr_i* are pseudofunctions expressing symmetry relations among the covariant and contravariant indices respectively. These are of the form `symoper(`*index_1*`,` *index_2*`,...)` where `symoper` is one of `sym`, `anti` or `cyc` and the *index_i* are integers indicating the position of the index in the *tensor*. This will declare *tensor* to be symmetric, antisymmetric or cyclic respectively in the *index_i*. `symoper(all)` is also an allowable form which indicates all indices obey the symmetry condition. For example, given an object `b` with 5 covariant indices, `decsym(b,5,3,[sym(1,2),anti(3,4)],[cyc(all)])` declares `b` symmetric in its first and second and antisymmetric in its third and fourth covariant indices, and cyclic in all of its contravariant indices. Either list of symmetry declarations may be null. The function which performs the simplifications is `canform` as the example below illustrates.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) expr:contract( expand( a([i1, j1, k1], [])
           *kdels([i, j, k], [i1, j1, k1])))$
(%i3) ishow(expr)$
(%t3)         a      + a      + a      + a      + a      + a
```

```
                k j i    k i j    j k i    j i k    i k j    i j k
(%i4) decsym(a,3,0,[sym(all)],[]);
(%o4)                                done
(%i5) ishow(canform(expr))$
(%t5)                               6 a
                                       i j k
(%i6) remsym(a,3,0);
(%o6)                                done
(%i7) decsym(a,3,0,[anti(all)],[]);
(%o7)                                done
(%i8) ishow(canform(expr))$
(%t8)                                  0
(%i9) remsym(a,3,0);
(%o9)                                done
(%i10) decsym(a,3,0,[cyc(all)],[]);
(%o10)                               done
(%i11) ishow(canform(expr))$
(%t11)                        3 a      + 3 a
                                 i k j      i j k
(%i12) dispsym(a,3,0);
(%o12)                    [[cyc, [[1, 2, 3]], []]]
```

remsym (*tensor*, *m*, *n*) Function

Removes all symmetry properties from *tensor* which has *m* covariant indices and *n* contravariant indices.

canform (*expr*) Function

canform (*expr*, *rename*) Function

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*] ...]) Function

is the usual Maxima differentiation function which has been expanded in its abilities for `itensor`. It takes the derivative of *expr* with respect to *v_1* *n_1* times, with respect to *v_2* *n_2* times, etc. For the tensor package, the function has been modified so that the *v_i* may be integers from 1 up to the value of the variable `dim`. This will

cause the differentiation to be carried out with respect to the *v_i*th member of the list `vect_coords`. If `vect_coords` is bound to an atomic variable, then that variable subscripted by *v_i* will be used for the variable of differentiation. This permits an array of coordinate names or subscripted names like `x[1]`, `x[2]`, ... to be used.

A further extension adds the ability to `diff` to compute derivatives with respect to an indexed variable. In particular, the tensor package knows how to differentiate expressions containing combinations of the metric tensor and its derivatives with respect to the metric tensor and its first and second derivatives. This capability is particularly useful when considering Lagrangian formulations of a gravitational theory, allowing one to derive the Einstein tensor and field equations from the action principle.

idiff (*expr, v_1,* [*n_1,* [*v_2, n_2*] ...]) Function

Indicial differentiation. Unlike `diff`, which differentiates with respect to an independent variable, `idiff)` can be used to differentiate with respect to a coordinate. For an indexed object, this amounts to appending the *v_i* as derivative indices. Subsequently, derivative indices will be sorted, unless `iframe_flag` is set to `true`.

`idiff` can also differentiate the determinant of the metric tensor. Thus, if `imetric` has been bound to `G` then `idiff(determinant(g),k)` will return `2*determinant(g)*ichr2([%i,k],[%i])` where the dummy index `%i` is chosen appropriately.

liediff (*v, ten*) Function

Computes the Lie-derivative of the tensorial expression *ten* with respect to the vector field *v*. *ten* should be any indexed tensor expression; *v* should be the name (without indices) of a vector field. For example:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(liediff(v,a([i,j],[])*b([],[k],l)))$
       k    %2            %2          %2
(%t2) b   (v   a       + v   a     + v   a    )
       ,l       i j,%2    ,j  i %2    ,i  %2 j

                          %1  k        %1  k      %1  k
                     + (v   b      - b   v    + v   b   ) a
                             ,%1 l    ,l  ,%1    ,l  ,%1   i j
```

rediff (*ten*) Function

Evaluates all occurrences of the `idiff` command in the tensorial expression *ten*.

undiff (*expr*) Function

Returns an expression equivalent to *expr* but with all derivatives of indexed objects replaced by the noun form of the `idiff` function. Its arguments would yield that indexed object if the differentiation were carried out. This is useful when it is desired to replace a differentiated indexed object with some function definition resulting in *expr* and then carry out the differentiation by saying `ev(`*expr*`, idiff)`.

evundiff (*expr*) Function

Equivalent to the execution of `undiff`, followed by `ev` and `rediff`.

The point of this operation is to easily evalute 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))$
                                        l
(%t3)                          icurvature
                                        i j k,m
(%i4) ishow(evundiff(%))$
             l              l         %1           l           %1
(%t4) - ichr2        - ichr2     ichr2      - ichr2       ichr2
             i k,j m        %1 j      i k,m        %1 j,m      i k

             l              l         %1           l           %1
      + ichr2        + ichr2     ichr2      + ichr2       ichr2
             i j,k m        %1 k      i j,m        %1 k,m      i j
```

Note: In earlier versions of Maxima, derivative forms of the Christoffel-symbols also could not be evaluated. This has been fixed now, so `evundiff` is no longer necessary for expressions like this:

```
(%i5) imetric(g);
(%o5)                                done
(%i6) ishow(ichr2([i,j],[k],l))$
       k %3
      g     (g         - g         + g        )
              j %3,i l    i j,%3 l    i %3,j l
(%t6) ----------------------------------------
                        2

                       k %3
                      g     (g       - g       + g      )
                       ,l     j %3,i    i j,%3    i %3,j
                    + ----------------------------------
                                    2
```

flush (*expr, tensor_1, tensor_2, ...*) Function

Set to zero, in *expr*, all occurrences of the *tensor_i* that have no derivative indices.

flushd (*expr, tensor_1, tensor_2, ...*) Function

Set to zero, in *expr*, all occurrences of the *tensor_i* that have derivative indices.

flushnd (*expr, tensor, n*) Function

Set to zero, in *expr*, all occurrences of the differentiated object *tensor* that have *n* or more derivative indices as the following example demonstrates.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(a([i],[J,r],k,r)+a([i],[j,r,s],k,r,s))$
                                J r      j r s
(%t2)                          a      + a
                                i,k r    i,k r s
(%i3) ishow(flushnd(%,a,3))$
                                     J r
(%t3)                               a
                                     i,k r
```

coord (*tensor_1, tensor_2, ...*) Function

Gives *tensor_i* the coordinate differentiation property that the derivative of contravariant vector whose name is one of the *tensor_i* yields a Kronecker delta. For example, if `coord(x)` has been done then `idiff(x([],[i]),j)` gives `kdelta([i],[j])`. `coord` is a list of all indexed objects having this property.

remcoord (*tensor_1, tensor_2, ...*) 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([],[i,j])*p([m],[n],i,j)`.

conmetderiv (*expr, tensor*) Function

Simplifies expressions containing ordinary derivatives of both covariant and contravariant forms of the metric tensor (the current restriction). For example, `conmetderiv` can relate the derivative of the contravariant metric tensor with the Christoffel symbols as seen from the following:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(g([],[a,b],c))$
                                      a b
(%t2)                                g
                                      ,c
```

```
(%i3) ishow(conmetderiv(%,g))$
                           %1 b      a       %1 a      b
(%t3)                  - g     ichr2     - g     ichr2
                                      %1 c              %1 c
```

simpmetderiv (*expr*) Function
simpmetderiv (*expr*[, *stop*]) Function

Simplifies expressions containing products of the derivatives of the metric tensor. Specifically, `simpmetderiv` recognizes two identities:

$$g^{ab}_{,d}\,g_{bc} + g^{ab}_{,d}\,g_{bc,d} = (g^{ab}\,g_{bc})_{,d} = (\delta^{a}_{c})_{,d} = 0$$

hence

$$g^{ab}_{,d}\,g_{bc} = -\,g^{ab}\,g_{bc,d}$$

and

$$g^{ab}_{,j}\,g_{ab,i} = g^{ab}_{,i}\,g_{ab,j}$$

which follows from the symmetries of the Christoffel symbols.

The `simpmetderiv` function takes one optional parameter which, when present, causes the function to stop after the first successful substitution in a product expression. The `simpmetderiv` function also makes use of the global variable *flipflag* which determines how to apply a "canonical" ordering to the product indices.

Put together, these capabilities can be used to achieve powerful simplifications that are difficult or impossible to accomplish otherwise. This is demonstrated through the following example that explicitly uses the partial simplification features of `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
(%i4) ishow(canform(%))$

errexp1 has improper indices
```

```
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i5) ishow(simpmetderiv(%))$
                               a b  b c
(%t5)                         g    g    g      g
                                        a b,d  b c,e
(%i6) flipflag:not flipflag;
(%o6)                                  true
(%i7) ishow(simpmetderiv(%th(2)))$
                                 a b  b c
(%t7)                           g    g    g    g
                                 ,d   ,e   a b  b c
(%i8) flipflag:not flipflag;
(%o8)                                  false
(%i9) ishow(simpmetderiv(%th(2),stop))$
                                 a b  b c
(%t9)                         - g    g    g      g
                                       ,e   a b,d  b c
(%i10) ishow(contract(%))$
                                       b c
(%t10)                              - g    g
                                       ,e   c b,d
```

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

```
(g      + g      - g     )/2 .
  ik,j     jk,i     ij,k
```

To evaluate the Christoffel symbols for a particular metric, the variable `imetric` must be assigned a name as in the example under `chr2`.

ichr2 ([*i, j*], [*k*]) Function

Yields the Christoffel symbol of the second kind defined by the relation

```
                          ks
ichr2([i,j],[k]) = g    (g      + g      - g     )/2
                           is,j     js,i     ij,s
```

icurvature ([*i, j, k*], [*h*]) Function

Yields the Riemann curvature tensor in terms of the Christoffel symbols of the second kind (`ichr2`). The following notation is used:

```
             h             h            h         %1         h
icurvature     = - ichr2      - ichr2     ichr2    + ichr2
             i j k         i k,j        %1 j      i k        i j,k
                            h           %1
                       + ichr2      ichr2
                              %1 k       i j
```

covdiff (*expr, v_1, v_2, ...*) Function

Yields the covariant derivative of *expr* with respect to the variables *v_i* in terms of the Christoffel symbols of the second kind (`ichr2`). In order to evaluate these, one should use `ev(`*expr*`,ichr2)`.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) entertensor()$
Enter tensor name: a;
Enter a list of the covariant indices: [i,j];
Enter a list of the contravariant indices: [k];
Enter a list of the derivative indices: [];
                                      k
(%t2)                                a
                                      i j
(%i3) ishow(covdiff(%,s))$
              k         %1     k         %1     k
(%t3)     - a     ichr2    - a     ichr2    + a
              i %1      j s    %1 j      i s    i j,s

              k     %1
      + ichr2     a
              %1 s  i j
(%i4) imetric:g;
(%o4)                                  g
(%i5) ishow(ev(%th(2),ichr2))$
         %1 %4  k
        g      a     (g       - g       + g      )
                i %1   s %4,j    j s,%4    j %4,s
(%t5) - ----------------------------------------
                           2
    %1 %3  k
```

```
  g      a     (g       - g       + g      )
          %1 j   s %3,i    i s,%3    i %3,s
- ----------------------------------------
                      2
   k %2  %1
  g     a    (g        - g        + g       )
         i j   s %2,%1    %1 s,%2    %1 %2,s     k
+ ---------------------------------------- + a
                      2                         i j,s
(%i6)
```

lorentz_gauge (*expr*) Function

Imposes the Lorentz condition by substituting 0 for all indexed objects in *expr* that have a derivative index identical to a contravariant index.

igeodesic_coords (*expr*, *name*) Function

Causes undifferentiated Christoffel symbols and first derivatives of the metric tensor vanish in *expr*. The *name* in the `igeodesic_coords` function refers to the metric *name* (if it appears in *expr*) while the connection coefficients must be called with the names `ichr1` and/or `ichr2`. The following example demonstrates the verification of the cyclic identity satisfied by the Riemann curvature tensor using the `igeodesic_coords` function.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(icurvature([r,s,t],[u]))$
             u            u         %1         u
(%t2) - ichr2      - ichr2     ichr2    + ichr2
             r t,s        %1 s      r t        r s,t

                                              u         %1
                                         + ichr2     ichr2
                                               %1 t      r s
(%i3) ishow(igeodesic_coords(%,ichr2))$
                                    u           u
(%t3)                         ichr2      - ichr2
                                    r s,t       r t,s
(%i4) ishow(igeodesic_coords(icurvature([r,s,t],[u]),ichr2)+
            igeodesic_coords(icurvature([s,t,r],[u]),ichr2)+
            igeodesic_coords(icurvature([t,r,s],[u]),ichr2))$
             u            u           u           u
(%t4) - ichr2      + ichr2      + ichr2      - ichr2
             t s,r        t r,s        s t,r        s r,t

                                            u           u
                                       - ichr2     + ichr2
                                              r t,s       r s,t
(%i5) canform(%);
```

```
(%o5)                                  0
```

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 `iframe_flag` to `true`. This causes the Christoffel-symbols, `ichr1` and `ichr2`, to be replaced by the more general frame connection coefficients `icc1` and `icc2` in calculations. Speficially, the behavior of `covdiff` and `icurvature` is changed.

The frame is defined by two tensors: the inverse frame field (`ifri`, the dual basis tetrad), and the frame metric `ifg`. The frame metric is the identity matrix for orthonormal frames, or the Lorentz metric for orthonormal frames in Minkowski spacetime. The inverse frame field defines the frame base (unit vectors). Contraction properties are defined for the frame field and the frame metric.

When `iframe_flag` is true, many `itensor` expressions use the frame metric `ifg` instead of the metric defined by `imetric` for raising and lowerind indices.

IMPORTANT: Setting the variable `iframe_flag` to `true` does NOT undefine the contraction properties of a metric defined by a call to `defcon` or `imetric`. If a frame field is used, it is best to define the metric by assigning its name to the variable `imetric` and NOT invoke the `imetric` function.

Maxima uses these two tensors to define the frame coefficients (`ifc1` and `ifc2`) which form part of the connection coefficients (`icc1` and `icc2`), as the following example demonstrates:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) iframe_flag:true;
(%o2)                                true
(%i3) ishow(covdiff(v([],[i]),j))$
                               i         i     %1
(%t3)                         v   + icc2     v
                               ,j        %1 j
(%i4) ishow(ev(%,icc2))$
                               %1      i        i
(%t4)                         v   ifc2      + v
                                       %1 j     ,j
(%i5) ishow(ev(%,ifc2))$
                          %1    i %2                 i
(%t5)                    v   ifg     ifc1         + v
                                         %1 j %2     ,j
(%i6) ishow(ev(%,ifc1))$
            %1    i %2
          v   ifg     (ifb         - ifb         + ifb        )
                          j %2 %1       %2 %1 j       %1 j %2     i
(%t6)     ------------------------------------------------- + v
```

```
                                      2                                ,j
(%i7) ishow(ifb([a,b,c]))$
                                                        %3    %4
(%t7)                (ifri        - ifri       ) ifr   ifr
                          a %3,%4       a %4,%3     b     c
```

An alternate method is used to compute the frame bracket (`ifb`) if the `iframe_bracket_form` flag is set to `false`:

```
(%i8) block([iframe_bracket_form:false],ishow(ifb([a,b,c])))$
                                  %6    %5        %5      %6
(%t8)               ifri     (ifr   ifr     - ifr     ifr  )
                        a %5     b     c,%6      b,%6    c
```

iframes () Function

Since in this version of Maxima, contraction identities for `ifr` and `ifri` are always defined, as is the frame bracket (`ifb`), this function does nothing.

ifb Variable

The frame bracket. The contribution of the frame metric to the connection coefficients is expressed using the frame bracket:

$$\mathrm{ifc1}_{abc} = \frac{-\mathrm{ifb}_{cab} + \mathrm{ifb}_{bca} + \mathrm{ifb}_{abc}}{2}$$

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`:

$$\mathrm{ifb}_{abc} = \mathrm{ifr}^{d}_{b}\,\mathrm{ifr}^{e}_{c}\,(\mathrm{ifri}_{a\,d,e} - \mathrm{ifri}_{a\,e,d} - \mathrm{ifri}_{a\,f}\,\mathrm{itr}^{f}_{d\,e})$$

Otherwise:

$$\mathrm{ifb}_{abc} = (\mathrm{ifr}^{e}_{b}\,\mathrm{ifr}^{d}_{c,e} - \mathrm{ifr}^{d}_{b,e}\,\mathrm{ifr}^{e}_{c})\,\mathrm{ifri}_{a\,d}$$

icc1 Variable

Connection coefficients of the first kind. In `itensor`, defined as

```
icc1    = ichr1    - ikt1    - inmc1
    abc        abc       abc        abc
```

In this expression, if `iframe_flag` is true, the Christoffel-symbol `ichr1` is replaced with the frame connection coefficient `ifc1`. If `itorsion_flag` is `false`, `ikt1` will be omitted. It is also omitted if a frame base is used, as the torsion is already calculated as part of the frame bracket. Lastly, of `inonmet_flag` is `false`, `inmc1` will not be present.

icc2 Variable

Connection coefficients of the second kind. In `itensor`, defined as

```
    c         c        c         c
icc2   = ichr2   - ikt2   - inmc2
    ab        ab       ab        ab
```

In this expression, if `iframe_flag` is true, the Christoffel-symbol `ichr2` is replaced with the frame connection coefficient `ifc2`. If `itorsion_flag` is `false`, `ikt2` will be omitted. It is also omitted if a frame base is used, as the torsion is already calculated as part of the frame bracket. Lastly, of `inonmet_flag` is `false`, `inmc2` will not be present.

ifc1 Variable

Frame coefficient of the first kind (also known as Ricci-rotation coefficients.) This tensor represents the contribution of the frame metric to the connection coefficient of the first kind. Defined as:

```
           - ifb      + ifb      + ifb
                c a b      b c a      a b c
ifc1    = ---------------------------------
    abc                  2
```

ifc2 Variable

Frame coefficient of the first kind. This tensor represents the contribution of the frame metric to the connection coefficient of the first kind. Defined as a permutation of the frame bracket (`ifb`) with the appropriate indices raised and lowered as necessary:

```
    c       cd
ifc2   = ifg   ifc1
    ab             abd
```

ifr Variable

The frame field. Contracts with the inverse frame field (`ifri`) to form the frame metric (`ifg`).

ifri Variable

The inverse frame field. Specifies the frame base (dual basis vectors). Along with the frame metric, it forms the basis of all calculations based on frames.

ifg Variable

The frame metric. Defaults to `kdelta`, but can be changed using `components`.

ifgi Variable

The inverse frame metric. Contracts with the frame metric (`ifg`) to `kdelta`.

iframe_bracket_form Option variable

Default value: `true`

Specifies how the frame bracket (`ifb`) is computed.

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 Variable

The nonmetricity vector. Conformal nonmetricity is defined through the covariant derivative of the metric tensor. Normally zero, the metric tensor's covariant derivative will evaluate to the following when `inonmet_flag` is set to `true`:

```
g     =- g  inm
 ij;k     ij   k
```

inmc1 Variable

Covariant permutation of the nonmetricity vector components. Defined as

```
            g   inm  - inm  g   - g   inm
             ab    c      a  bc    ac    b
inmc1    = ------------------------------
     abc                 2
```

(Substitute `ifg` in place of `g` if a frame metric is used.)

inmc2 Variable

Contravariant permutation of the nonmetricity vector components. Used in the connection coefficients if `inonmet_flag` is `true`. Defined as:

```
                      c         c         cd
          -inm  kdelta  - kdelta  inm  + g   inm  g
c          a       b         a    b          d  ab
```

$$\text{inmc2}_{ab} = \frac{\text{covdiff}(\text{ichr2}_{ij},k) - \text{covdiff}(\text{ichr2}_{ik},j) + \text{ichr2}_{ij,4}\,\text{ichr2}_{k4} - \text{ichr2}_{ik,4}\,\text{ichr2}_{j4}}{2}$$

(Substitute `ifg` in place of `g` if a frame metric is used.)

ikt1 Variable

Covariant permutation of the torsion tensor (also known as contorsion). Defined as:

$$\text{ikt1}_{abc} = \frac{-g_{ad}\,\text{itr}^{d}_{cb} - g_{bd}\,\text{itr}^{d}_{ca} - \text{itr}_{ab}\,g_{cd}}{2}$$

(Substitute `ifg` in place of `g` if a frame metric is used.)

ikt2 Variable

Contravariant permutation of the torsion tensor (also known as contorsion). Defined as:

$$\text{ikt2}^{c}_{ab} = g^{cd}\,\text{ikt1}_{abd}$$

(Substitute `ifg` in place of `g` if a frame metric is used.)

itr Variable

The torsion tensor. For a metric with torsion, repeated covariant differentiation on a scalar function will not commute, as demonstrated by the following example:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) imetric:g;
(%o2)                                  g
(%i3) covdiff( covdiff( f( [], []), i), j)
                      - covdiff( covdiff( f( [], []), j), i)$
(%i4) ishow(%)$
                                    %4              %2
(%t4)                      f    ichr2    - f    ichr2
                            ,%4      j i    ,%2      i j
(%i5) canform(%);
(%o5)                                  0
(%i6) itorsion_flag:true;
(%o6)                                true
(%i7) covdiff( covdiff( f( [], []), i), j)
                      - covdiff( covdiff( f( [], []), j), i)$
(%i8) ishow(%)$
```

```
                                %8              %6
(%t8)                 f    icc2    - f    icc2    - f     + f
                       ,%8     j i    ,%6     i j    ,j i    ,i j
(%i9) ishow(canform(%))$
                                      %1              %1
(%t9)                          f    icc2    - f    icc2
                                ,%1     j i    ,%1     i j
(%i10) ishow(canform(ev(%,icc2)))$
                                      %1              %1
(%t10)                         f    ikt2    - f    ikt2
                                ,%1     i j    ,%1     j i
(%i11) ishow(canform(ev(%,ikt2)))$
                               %2 %1                    %2 %1
(%t11)              f    g      ikt1       - f    g      ikt1
                     ,%2            i j %1    ,%2            j i %1
(%i12) ishow(factor(canform(rename(expand(ev(%,ikt1))))))$
                               %3 %2            %1       %1
                          f    g      g      (itr    - itr   )
                           ,%3         %2 %1     j i      i j
(%t12)                    ------------------------------------
                                        2
(%i13) decsym(itr,2,1,[anti(all)],[]);
(%o13)                                done
(%i14) defcon(g,g,kdelta);
(%o14)                                done
(%i15) subst(g,nounify(g),%th(3))$
(%i16) ishow(canform(contract(%)))$
                                              %1
(%t16)                               - f    itr
                                        ,%1    i j
```

27.2.7 Exterior algebra

The `itensor` package can perform operations on totally antisymmetric covariant tensor fields. A totally antisymmetric tensor field of rank (0,L) corresponds with a differential L-form. On these objects, a multiplication operation known as the exterior product, or wedge product, is defined.

Unfortunately, not all authors agree on the definition of the wedge product. Some authors prefer a definition that corresponds with the notion of antisymmetrization: in these works, the wedge product of two vector fields, for instance, would be defined as

```
                a a  - a a
                 i j    j i
    a  /\ a  = -----------
     i     j        2
```

More generally, the product of a p-form and a q-form would be defined as

```
                          1     k1..kp l1..lq
  A       /\ B       = ------ D              A       B
```

```
 i1..ip     j1..jq   (p+q)!  i1..ip j1..jq  k1..kp  l1..lq
```

where `D` stands for the Kronecker-delta.

Other authors, however, prefer a "geometric" definition that corresponds with the notion of the volume element:

```
a  /\ a  = a a  - a a
 i     j    i j    j i
```

and, in the general case

```
                        1    k1..kp l1..lq
A       /\ B       = ----- D              A       B
 i1..ip     j1..jq   p! q!  i1..ip j1..jq  k1..kp  l1..lq
```

Since `itensor` is a tensor algebra package, the first of these two definitions appears to be the more natural one. Many applications, however, utilize the second definition. To resolve this dilemma, a flag has been implemented that controls the behavior of the wedge product: if `igeowedge_flag` is `false` (the default), the first, "tensorial" definition is used, otherwise the second, "geometric" definition will be applied.

`~` Operator

The wedge product operator is denoted by the tilde `~`. This is a binary operator. Its arguments should be expressions involving scalars, covariant tensors of rank one, or covariant tensors of rank `1` that have been declared antisymmetric in all covariant indices.

The behavior of the wedge product operator is controlled by the `igeowedge_flag` flag, as in the following example:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(a([i])~b([j]))$
                                  a  b  - b  a
                                   i  j    i  j
(%t2)                             -------------
                                        2
(%i3) decsym(a,2,0,[anti(all)],[]);
(%o3)                                done
(%i4) ishow(a([i,j])~b([k]))$
                          a    b  + b  a    - a    b
                           i j  k    i  j k    i k  j
(%t4)                     ---------------------------
                                    3
(%i5) igeowedge_flag:true;
(%o5)                                true
(%i6) ishow(a([i])~b([j]))$
(%t6)                             a  b  - b  a
                                   i  j    i  j
(%i7) ishow(a([i,j])~b([k]))$
(%t7)                     a    b  + b  a    - a    b
                           i j  k    i  j k    i k  j
```

| Operator

The vertical bar | denotes the "contraction with a vector" binary operation. When a totally antisymmetric covariant tensor is contracted with a contravariant vector, the result is the same regardless which index was used for the contraction. Thus, it is possible to define the contraction operation in an index-free manner.

In the `itensor` package, contraction with a vector is always carried out with respect to the first index in the literal sorting order. This ensures better simplification of expressions involving the | operator. For instance:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) decsym(a,2,0,[anti(all)],[]);
(%o2)                                done
(%i3) ishow(a([i,j],[])|v)$
                                      %1
(%t3)                                 v   a
                                           %1 j
(%i4) ishow(a([j,i],[])|v)$
                                      %1
(%t4)                                - v   a
                                           %1 j
```

Note that it is essential that the tensors used with the | operator be declared totally antisymmetric in their covariant indices. Otherwise, the results will be incorrect.

extdiff (*expr*, *i*) Function

Computes the exterior derivative of *expr* with respect to the index *i*. The exterior derivative is formally defined as the wedge product of the partial derivative operator and a differential form. As such, this operation is also controlled by the setting of `igeowedge_flag`. For instance:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(extdiff(v([i]),j))$
                                     v    - v
                                      j,i    i,j
(%t2)                                -----------
                                          2
(%i3) decsym(a,2,0,[anti(all)],[]);
(%o3)                                   done
(%i4) ishow(extdiff(a([i,j]),k))$
                              a      - a      + a
                               j k,i    i k,j    i j,k
(%t4)                         ------------------------
                                        3
(%i5) igeowedge_flag:true;
(%o5)                                   true
(%i6) ishow(extdiff(v([i]),j))$
(%t6)                                v    - v
                                      j,i    i,j
(%i7) ishow(extdiff(a([i,j]),k))$
```

```
(%t7)                      - (a      - a      + a     )
                               k j,i    k i,j    j i,k
```

hodge (*expr*) Function

Compute the Hodge-dual of *expr*. For instance:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) imetric(g);
(%o2)                            done
(%i3) idim(4);
(%o3)                            done
(%i4) icounter:100;
(%o4)                             100
(%i5) decsym(A,3,0,[anti(all)],[])$

(%i6) ishow(A([i,j,k],[]))$
(%t6)                            A
                                  i j k
(%i7) ishow(canform(hodge(%)))$
                           %1 %2 %3 %4
               levi_civita            g        A
                                       %1 %102  %2 %3 %4
(%t7)          ----------------------------------------
                                  6
(%i8) ishow(canform(hodge(%)))$
                  %1 %2 %3 %8            %4 %5 %6 %7
(%t8) levi_civita            levi_civita            g
                                                     %1 %106
                               g        g        g      A          /6
                                %2 %107  %3 %108  %4 %8  %5 %6 %7
(%i9) lc2kdt(%)$

(%i10) %,kdelta$

(%i11) ishow(canform(contract(expand(%))))$
(%t11)                     - A
                              %106 %107 %108
```

igeowedge_flag Option variable

Default value: `false`

Controls the behavior of the wedge product and exterior derivative. When set to `false` (the default), the notion of differential forms will correspond with that of a totally antisymmetric covariant tensor field. When set to `true`, differential forms will agree with the notion of the volume element.

27.2.8 Exporting TeX expressions

The `itensor` package provides limited support for exporting tensor expressions to TeX. Since `itensor` expressions appear as function calls, the regular Maxima `tex` command will not produce the expected output. You can try instead the `tentex` command, which attempts to translate tensor expressions into appropriately indexed TeX objects.

tentex (*expr*) Function

To use the `tentex` function, you must first load `tentex`, as in the following example:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) load(tentex);
(%o2)       /share/tensor/tentex.lisp
(%i3) idummyx:m;
(%o3)                                  m
(%i4) ishow(icurvature([j,k,l],[i]))$
             m1       i           m1       i           i
(%t4)  ichr2    ichr2     - ichr2    ichr2     - ichr2
             j k      m1 l        j l      m1 k        j l,k

                                                      i
                                                + ichr2
                                                      j k,l
(%i5) tentex(%)$
$$\Gamma_{j\,k}^{m_1}\,\Gamma_{l\,m_1}^{i}-\Gamma_{j\,l}^{m_1}\,
 \Gamma_{k\,m_1}^{i}-\Gamma_{j\,l,k}^{i}+\Gamma_{j\,k,l}^{i}$$
```

Note the use of the `idummyx` assignment, to avoid the appearance of the percent sign in the TeX expression, which may lead to compile errors.

NB: This version of the `tentex` function is somewhat experimental.

27.2.9 Interfacing with ctensor

The `itensor` package has the ability to generate Maxima code that can then be executed in the context of the `ctensor` package. The function that performs this task is `ic_convert`.

ic_convert (*eqn*) Function

Converts the `itensor` equation *eqn* to a `ctensor` assignment statement. Implied sums over dummy indices are made explicit while indexed objects are transformed into arrays (the array subscripts are in the order of covariant followed by contravariant indices of the indexed objects). The derivative of an indexed object will be replaced by the noun form of `diff` taken with respect to `ct_coords` subscripted by the derivative index. The Christoffel symbols `ichr1` and `ichr2` will be translated to `lcs` and `mcs`, respectively and if `metricconvert` is `true` then all occurrences of the metric with two covariant (contravariant) indices will be renamed to `lg` (`ug`). In addition, `do` loops will be introduced summing over all free indices so that the transformed assignment

statement can be evaluated by just doing ev. The following examples demonstrate the features of this function.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) eqn:ishow(t([i,j],[k])=f([],[])*g([l,m],[])*a([],[m],j)
      *b([i],[l,k]))$
                              k        m   l k
(%t2)                        t    = f a   b    g
                              i j      ,j  i    l m
(%i3) ic_convert(eqn);
(%o3) for i thru dim do (for j thru dim do (
       for k thru dim do
        t        : f sum(sum(diff(a , ct_coords ) b
         i, j, k                  m           j   i, l, k

 g    , l, 1, dim), m, 1, dim)))
  l, m
(%i4) imetric(g);
(%o4)                                done
(%i5) metricconvert:true;
(%o5)                                true
(%i6) ic_convert(eqn);
(%o6) for i thru dim do (for j thru dim do (
       for k thru dim do
        t        : f sum(sum(diff(a , ct_coords ) b
         i, j, k                  m           j   i, l, k

 lg    , l, 1, dim), m, 1, dim)))
   l, m
```

27.2.10 Reserved words

The following Maxima words are used by the itensor package internally and should not be redefined:

```
Keyword    Comments
----------------------------------------
indices2() Internal version of indices()
conti      Lists contravariant indices
covi       Lists covariant indices of a indexed object
deri       Lists derivative indices of an indexed object
name       Returns the name of an indexed object
concan
irpmon
lc0
_lc2kdt0
_lcprod
_extlc
```

28 ctensor

28.1 Introduction to ctensor

`ctensor` is a component tensor manipulation package. To use the `ctensor` package, type `load(ctensor)`. To begin an interactive session with `ctensor`, type `csetup()`. You are first asked to specify the dimension of the manifold. If the dimension is 2, 3 or 4 then the list of coordinates defaults to `[x,y]`, `[x,y,z]` or `[x,y,z,t]` respectively. These names may be changed by assigning a new list of coordinates to the variable `ct_coords` (described below) and the user is queried about this. Care must be taken to avoid the coordinate names conflicting with other object definitions.

Next, the user enters the metric either directly or from a file by specifying its ordinal position. The metric is stored in the matrix `lg`. Finally, the metric inverse is computed and stored in the matrix `ug`. One has the option of carrying out all calculations in a power series.

A sample protocol is begun below for the static, spherically symmetric metric (standard coordinates) which will be applied to the problem of deriving Einstein's vacuum equations (which lead to the Schwarzschild solution) as an example. Many of the functions in `ctensor` will be displayed for the standard metric as examples.

```
(%i1) load(ctensor);
(%o1)      /share/tensor/ctensor.mac
(%i2) csetup();
Enter the dimension of the coordinate system:
4;
Do you wish to change the coordinate names?
n;
Do you want to
1. Enter a new metric?

2. Enter a metric from a file?

3. Approximate a metric with a Taylor series?
1;

Is the matrix  1. Diagonal  2. Symmetric  3. Antisymmetric  4. General
Answer 1, 2, 3 or 4
1;
Row 1 Column 1:
a;
Row 2 Column 2:
x^2;
Row 3 Column 3:
x^2*sin(y)^2;
Row 4 Column 4:
-d;

Matrix entered.
```

```
Enter functional dependencies with the DEPENDS function or 'N' if none
depends([a,d],x);
Do you wish to see the metric?
y;
                          [ a  0       0        0  ]
                          [                        ]
                          [     2                  ]
                          [ 0  x       0        0  ]
                          [                        ]
                          [         2    2         ]
                          [ 0  0   x  sin (y)   0  ]
                          [                        ]
                          [ 0  0       0       - d ]
(%o2)                                done
(%i3) christof(mcs);
                                              a
                                             x
(%t3)                          mcs         = ---
                                  1, 1, 1    2 a

                                             1
(%t4)                           mcs        = -
                                   1, 2, 2   x

                                             1
(%t5)                           mcs        = -
                                   1, 3, 3   x

                                            d
                                           x
(%t6)                          mcs        = ---
                                  1, 4, 4   2 d

                                             x
(%t7)                          mcs        = - -
                                  2, 2, 1    a

                                          cos(y)
(%t8)                         mcs        = ------
                                 2, 3, 3   sin(y)

                                              2
                                         x sin (y)
(%t9)                      mcs        = - ---------
                              3, 3, 1         a

(%t10)                   mcs        = - cos(y) sin(y)
                            3, 3, 2
```

```
                                          d
                                           x
(%t11)                          mcs      = ---
                                   4, 4, 1   2 a
(%o11)                             done
```

28.2 Functions and Variables for ctensor

28.2.1 Initialization and setup

csetup () Function

A function in the `ctensor` (component tensor) package which initializes the package and allows the user to enter a metric interactively. See `ctensor` for more details.

cmetric (*dis*) Function
cmetric () Function

A function in the `ctensor` (component tensor) package that computes the metric inverse and sets up the package for further calculations.

If `cframe_flag` is `false`, the function computes the inverse metric `ug` from the (user-defined) matrix `lg`. The metric determinant is also computed and stored in the variable `gdet`. Furthermore, the package determines if the metric is diagonal and sets the value of `diagmetric` accordingly. If the optional argument *dis* is present and not equal to `false`, the user is prompted to see the metric inverse.

If `cframe_flag` is `true`, the function expects that the values of `fri` (the inverse frame matrix) and `lfg` (the frame metric) are defined. From these, the frame matrix `fr` and the inverse frame metric `ufg` are computed.

ct_coordsys (*coordinate_system*, *extra_arg*) Function
ct_coordsys (*coordinate_system*) Function

Sets up a predefined coordinate system and metric. The argument *coordinate_system* can be one of the following symbols:

SYMBOL	Dim	Coordinates	Description/comments
cartesian2d	2	[x,y]	Cartesian 2D coordinate system
polar	2	[r,phi]	Polar coordinate system
elliptic	2	[u,v]	Elliptic coord. system
confocalelliptic	2	[u,v]	Confocal elliptic coordinates
bipolar	2	[u,v]	Bipolar coord. system
parabolic	2	[u,v]	Parabolic coord. system
cartesian3d	3	[x,y,z]	Cartesian 3D coordinate system

polarcylindrical	3	[r,theta,z]	Polar 2D with cylindrical z
ellipticcylindrical	3	[u,v,z]	Elliptic 2D with cylindrical z
confocalellipsoidal	3	[u,v,w]	Confocal ellipsoidal
bipolarcylindrical	3	[u,v,z]	Bipolar 2D with cylindrical z
paraboliccylindrical	3	[u,v,z]	Parabolic 2D with cylindrical z
paraboloidal	3	[u,v,phi]	Paraboloidal coords.
conical	3	[u,v,w]	Conical coordinates
toroidal	3	[u,v,phi]	Toroidal coordinates
spherical	3	[r,theta,phi]	Spherical coord. system
oblatespheroidal	3	[u,v,phi]	Oblate spheroidal coordinates
oblatespheroidalsqrt	3	[u,v,phi]	
prolatespheroidal	3	[u,v,phi]	Prolate spheroidal coordinates
prolatespheroidalsqrt	3	[u,v,phi]	
ellipsoidal	3	[r,theta,phi]	Ellipsoidal coordinates
cartesian4d	4	[x,y,z,t]	Cartesian 4D coordinate system
spherical4d	4	[r,theta,eta,phi]	Spherical 4D coordinate system
exteriorschwarzschild	4	[t,r,theta,phi]	Schwarzschild metric
interiorschwarzschild	4	[t,z,u,v]	Interior Schwarzschild metric
kerr_newman	4	[t,r,theta,phi]	Charged axially symmetric metric

`coordinate_system` can also be a list of transformation functions, followed by a list containing the coordinate variables. For instance, you can specify a spherical metric as follows:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) ct_coordsys([r*cos(theta)*cos(phi),r*cos(theta)*sin(phi),
      r*sin(theta),[r,theta,phi]]);
(%o2)                                done
(%i3) lg:trigsimp(lg);
                           [ 1  0         0        ]
                           [                       ]
                           [     2                 ]
(%o3)                      [ 0  r         0        ]
                           [                       ]
                           [         2    2        ]
                           [ 0  0   r  cos (theta) ]
(%i4) ct_coords;
(%o4)                           [r, theta, phi]
(%i5) dim;
```

```
(%o5)                                3
```

Transformation functions can also be used when `cframe_flag` is `true`:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) cframe_flag:true;
(%o2)                               true
(%i3) ct_coordsys([r*cos(theta)*cos(phi),r*cos(theta)*sin(phi),
      r*sin(theta),[r,theta,phi]]);
(%o3)                               done
(%i4) fri;
(%o4)
 [cos(phi)cos(theta) -cos(phi) r sin(theta) -sin(phi) r cos(theta)]
 [                                                                 ]
 [sin(phi)cos(theta) -sin(phi) r sin(theta)  cos(phi) r cos(theta)]
 [                                                                 ]
 [    sin(theta)           r cos(theta)               0          ]

(%i5) cmetric();
(%o5)                               false
(%i6) lg:trigsimp(lg);
                           [ 1  0         0        ]
                           [                       ]
                           [     2                 ]
(%o6)                      [ 0  r         0        ]
                           [                       ]
                           [         2    2        ]
                           [ 0  0   r  cos (theta) ]
```

The optional argument *extra_arg* can be any one of the following:

`cylindrical` tells `ct_coordsys` to attach an additional cylindrical coordinate.

`minkowski` tells `ct_coordsys` to attach an additional coordinate with negative metric signature.

`all` tells `ct_coordsys` to call `cmetric` and `christof(false)` after setting up the metric.

If the global variable `verbose` is set to `true`, `ct_coordsys` displays the values of `dim`, `ct_coords`, and either `lg` or `lfg` and `fri`, depending on the value of `cframe_flag`.

init_ctensor () Function

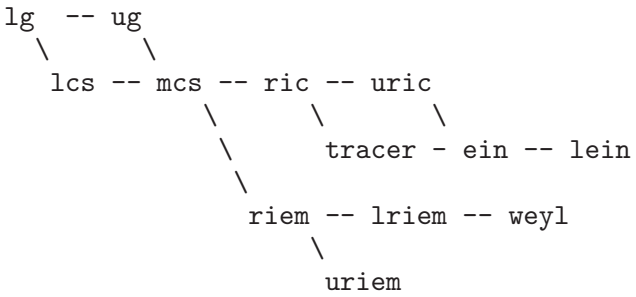
Initializes the `ctensor` package.

The `init_ctensor` function reinitializes the `ctensor` package. It removes all arrays and matrices used by `ctensor`, resets all flags, resets `dim` to 4, and resets the frame metric to the Lorentz-frame.

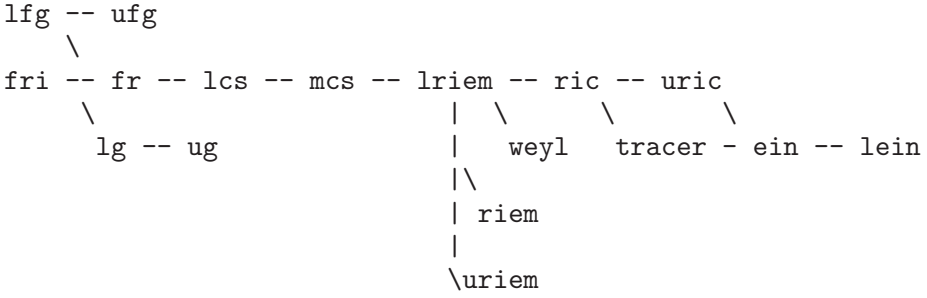
28.2.2 The tensors of curved space

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:



`ctensor` can also work using moving frames. When `cframe_flag` is set to `true`, the following tensors can be calculated:



christof (*dis*) Function

A function in the `ctensor` (component tensor) package. It computes the Christoffel symbols of both kinds. The argument *dis* determines which results are to be immediately displayed. The Christoffel symbols of the first and second kinds are stored in the arrays `lcs[i,j,k]` and `mcs[i,j,k]` respectively and defined to be symmetric in the first two indices. If the argument to `christof` is `lcs` or `mcs` then the unique non-zero values of `lcs[i,j,k]` or `mcs[i,j,k]`, respectively, will be displayed. If the argument is `all` then the unique non-zero values of `lcs[i,j,k]` and `mcs[i,j,k]` will be displayed. If the argument is `false` then the display of the elements will not occur. The array elements `mcs[i,j,k]` are defined in such a manner that the final index is contravariant.

ricci (*dis*) Function

A function in the `ctensor` (component tensor) package. `ricci` computes the covariant (symmetric) components `ric[i,j]` of the Ricci tensor. If the argument *dis* is `true`, then the non-zero components are displayed.

uricci (*dis*) Function

This function first computes the covariant components `ric[i,j]` of the Ricci tensor. Then the mixed Ricci tensor is computed using the contravariant metric tensor. If the value of the argument *dis* is `true`, then these mixed components, `uric[i,j]` (the index `i` is covariant and the index `j` is contravariant), will be displayed directly. Otherwise, `ricci(false)` will simply compute the entries of the array `uric[i,j]` without displaying the results.

scurvature () Function

Returns the scalar curvature (obtained by contracting the Ricci tensor) of the Riemannian manifold with the given metric.

einstein (*dis*) Function

A function in the `ctensor` (component tensor) package. `einstein` computes the mixed Einstein tensor after the Christoffel symbols and Ricci tensor have been obtained (with the functions `christof` and `ricci`). If the argument *dis* is `true`, then the non-zero values of the mixed Einstein tensor `ein[i,j]` will be displayed where `j` is the contravariant index. The variable `rateinstein` will cause the rational simplification on these components. If `ratfac` is `true` then the components will also be factored.

leinstein (*dis*) Function

Covariant Einstein-tensor. `leinstein` stores the values of the covariant Einstein tensor in the array `lein`. The covariant Einstein-tensor is computed from the mixed Einstein tensor `ein` by multiplying it with the metric tensor. If the argument *dis* is `true`, then the non-zero values of the covariant Einstein tensor are displayed.

riemann (*dis*) Function

A function in the `ctensor` (component tensor) package. `riemann` computes the Riemann curvature tensor from the given metric and the corresponding Christoffel symbols. The following index conventions are used:

```
                    l       _l        _l        _l    _m    _l    _m
R[i,j,k,l] =  R    = |       - |       + |     |   - |     |
                   ijk      ij,k      ik,j      mk    ij    mj    ik
```

This notation is consistent with the notation used by the `itensor` package and its `icurvature` function. If the optional argument *dis* is `true`, the non-zero components `riem[i,j,k,l]` will be displayed. As with the Einstein tensor, various switches set by the user control the simplification of the components of the Riemann tensor. If `ratriemann` is `true`, then rational simplification will be done. If `ratfac` is `true` then each of the components will also be factored.

If the variable `cframe_flag` is `false`, the Riemann tensor is computed directly from the Christoffel-symbols. If `cframe_flag` is `true`, the covariant Riemann-tensor is computed first from the frame field coefficients.

lriemann (*dis*) Function

Covariant Riemann-tensor (`lriem[]`).

Computes the covariant Riemann-tensor as the array `lriem`. If the argument *dis* is `true`, unique nonzero values are displayed.

If the variable `cframe_flag` is `true`, the covariant Riemann tensor is computed directly from the frame field coefficients. Otherwise, the (3,1) Riemann tensor is computed first.

For information on index ordering, see `riemann`.

uriemann (*dis*) Function

Computes the contravariant components of the Riemann curvature tensor as array elements `uriem[i,j,k,l]`. These are displayed if *dis* is `true`.

rinvariant () Function

Forms the Kretchmann-invariant (`kinvariant`) obtained by contracting the tensors

```
lriem[i,j,k,l]*uriem[i,j,k,l].
```

This object is not automatically simplified since it can be very large.

weyl (*dis*) Function

Computes the Weyl conformal tensor. If the argument *dis* is `true`, the non-zero components `weyl[i,j,k,l]` will be displayed to the user. Otherwise, these components will simply be computed and stored. If the switch `ratweyl` is set to `true`, then the components will be rationally simplified; if `ratfac` is `true` then the results will be factored as well.

28.2.3 Taylor series expansion

The `ctensor` package has the ability to truncate results by assuming that they are Taylor-series approximations. This behavior is controlled by the `ctayswitch` variable; when set to true, `ctensor` makes use internally of the function `ctaylor` when simplifying results.

The `ctaylor` function is invoked by the following `ctensor` functions:

```
Function     Comments
--------------------------------
christof()   For mcs only
ricci()
uricci()
einstein()
riemann()
weyl()
checkdiv()
```

ctaylor () Function

The `ctaylor` function truncates its argument by converting it to a Taylor-series using `taylor`, and then calling `ratdisrep`. This has the combined effect of dropping terms higher order in the expansion variable `ctayvar`. The order of terms that should be dropped is defined by `ctaypov`; the point around which the series expansion is carried out is specified in `ctaypt`.

As an example, consider a simple metric that is a perturbation of the Minkowski metric. Without further restrictions, even a diagonal metric produces expressions for the Einstein tensor that are far too complex:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) ratfac:true;
(%o2)                                true
(%i3) derivabbrev:true;
(%o3)                                true
(%i4) ct_coords:[t,r,theta,phi];
(%o4)                          [t, r, theta, phi]
(%i5) lg:matrix([-1,0,0,0],[0,1,0,0],[0,0,r^2,0],
               [0,0,0,r^2*sin(theta)^2]);
                       [ - 1  0  0         0        ]
                       [                            ]
                       [  0   1  0         0        ]
                       [                            ]
(%o5)                  [          2                 ]
                       [  0   0  r         0        ]
                       [                            ]
                       [              2    2        ]
                       [  0   0  0   r  sin (theta) ]
(%i6) h:matrix([h11,0,0,0],[0,h22,0,0],[0,0,h33,0],[0,0,0,h44]);
                           [ h11   0    0    0  ]
                           [                    ]
                           [  0   h22   0    0  ]
(%o6)                      [                    ]
                           [  0    0   h33   0  ]
                           [                    ]
                           [  0    0    0   h44 ]
(%i7) depends(l,r);
(%o7)                              [l(r)]
(%i8) lg:lg+l*h;
      [ h11 l - 1      0          0                 0           ]
      [                                                       ]
      [     0      h22 l + 1      0                 0           ]
      [                                                       ]
(%o8) [                         2                             ]
      [     0          0      r  + h33 l            0           ]
      [                                                       ]
      [                                    2    2             ]
      [     0          0          0       r  sin (theta) + h44 l ]
(%i9) cmetric(false);
(%o9)                              done
(%i10) einstein(false);
(%o10)                             done
(%i11) ntermst(ein);
[[1, 1], 62]
```

```
[[1, 2], 0]
[[1, 3], 0]
[[1, 4], 0]
[[2, 1], 0]
[[2, 2], 24]
[[2, 3], 0]
[[2, 4], 0]
[[3, 1], 0]
[[3, 2], 0]
[[3, 3], 46]
[[3, 4], 0]
[[4, 1], 0]
[[4, 2], 0]
[[4, 3], 0]
[[4, 4], 46]
(%o12)                                 done
```

However, if we recompute this example as an approximation that is linear in the variable `l`, we get much simpler expressions:

```
(%i14) ctayswitch:true;
(%o14)                                 true
(%i15) ctayvar:l;
(%o15)                                   l
(%i16) ctaypov:1;
(%o16)                                   1
(%i17) ctaypt:0;
(%o17)                                   0
(%i18) christof(false);
(%o18)                                 done
(%i19) ricci(false);
(%o19)                                 done
(%i20) einstein(false);
(%o20)                                 done
(%i21) ntermst(ein);
[[1, 1], 6]
[[1, 2], 0]
[[1, 3], 0]
[[1, 4], 0]
[[2, 1], 0]
[[2, 2], 13]
[[2, 3], 2]
[[2, 4], 0]
[[3, 1], 0]
[[3, 2], 2]
[[3, 3], 9]
[[3, 4], 0]
[[4, 1], 0]
[[4, 2], 0]
```

```
[[4, 3], 0]
[[4, 4], 9]
(%o21)                                done
(%i22) ratsimp(ein[1,1]);
                          2      2  4               2     2
(%o22) - (((h11 h22 - h11 ) (l )  r  - 2 h33 l    r ) sin (theta)
                               r               r r

                             2               2      4    2
                 - 2 h44 l    r  - h33 h44 (l ) )/(4 r  sin (theta))
                          r r                r
```

This capability can be useful, for instance, when working in the weak field limit far from a gravitational source.

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*) Function

The frame bracket (`fb[]`).

Computes the frame bracket according to the following definition:

```
   c          c          c         d     e
ifb   = ( ifri    - ifri    ) ifr   ifr
   ab         d,e        e,d       a     b
```

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 () Function

Computes a Newman-Penrose null tetrad (`np`) and its raised-index counterpart (`npi`). See `petrov` for an example.

The null tetrad is constructed on the assumption that a four-diemensional orthonormal frame metric with metric signature (-,+,+,+) is being used. The components of the null tetrad are related to the inverse frame matrix as follows:

```
np  = (fri  + fri ) / sqrt(2)
  1       1      2

np  = (fri  - fri ) / sqrt(2)
  2       1      2
```

```
np  = (fri  + %i fri ) / sqrt(2)
  3       3         4

np  = (fri  - %i fri ) / sqrt(2)
  4       3         4
```

psi (*dis*) Function

Computes the five Newman-Penrose coefficients `psi[0]`...`psi[4]`. If `psi` is set to `true`, the coefficients are displayed. See `petrov` for an example.

These coefficients are computed from the Weyl-tensor in a coordinate base. If a frame base is used, the Weyl-tensor is first converted to a coordinate base, which can be a computationally expensive procedure. For this reason, in some cases it may be more advantageous to use a coordinate base in the first place before the Weyl tensor is computed. Note however, that constructing a Newman-Penrose null tetrad requires a frame base. Therefore, a meaningful computation sequence may begin with a frame base, which is then used to compute `lg` (computed automatically by `cmetric` and then `ug`. At this point, you can switch back to a coordinate base by setting `cframe_flag` to false before beginning to compute the Christoffel symbols. Changing to a frame base at a later stage could yield inconsistent results, as you may end up with a mixed bag of tensors, some computed in a frame base, some in a coordinate base, with no means to distinguish between the two.

petrov () Function

Computes the Petrov classification of the metric characterized by `psi[0]`...`psi[4]`.

For example, the following demonstrates how to obtain the Petrov-classification of the Kerr metric:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) (cframe_flag:true,gcd:spmod,ctrgsimp:true,ratfac:true);
(%o2)                                true
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3)                                done
(%i4) ug:invert(lg)$
(%i5) weyl(false);
(%o5)                                done
(%i6) nptetrad(true);
(%t6) np =

[ sqrt(r - 2 m)           sqrt(r)                              ]
[---------------   ----------------------    0           0    ]
[sqrt(2) sqrt(r)   sqrt(2) sqrt(r - 2 m)                       ]
[                                                              ]
[ sqrt(r - 2 m)            sqrt(r)                             ]
[---------------  - ----------------------   0           0    ]
[sqrt(2) sqrt(r)    sqrt(2) sqrt(r - 2 m)                      ]
```

```
[                                                                    ]
[                                          r      %i r sin(theta) ]
[       0                    0          -------   --------------- ]
[                                       sqrt(2)       sqrt(2)     ]
[                                                                   ]
[                                          r       %i r sin(theta)]
[       0                    0          -------  - ---------------]
[                                       sqrt(2)        sqrt(2)    ]

                             sqrt(r)         sqrt(r - 2 m)
(%t7) npi = matrix([- ----------------------,---------------, 0, 0],
                    sqrt(2) sqrt(r - 2 m) sqrt(2) sqrt(r)

          sqrt(r)            sqrt(r - 2 m)
[- ----------------------, - ---------------, 0, 0],
   sqrt(2) sqrt(r - 2 m)    sqrt(2) sqrt(r)

           1               %i
[0, 0, ---------, --------------------],
       sqrt(2) r  sqrt(2) r sin(theta)

           1                %i
[0, 0, ---------, - --------------------])
       sqrt(2) r    sqrt(2) r sin(theta)

(%o7)                                done
(%i7) psi(true);
(%t8)                              psi  = 0
                                      0

(%t9)                              psi  = 0
                                      1

                                          m
(%t10)                             psi  = --
                                      2    3
                                           r

(%t11)                             psi  = 0
                                      3

(%t12)                             psi  = 0
                                      4
(%o12)                               done
(%i12) petrov();
(%o12)                                D
```

The Petrov classification function is based on the algorithm published in "Classifying geometries in general relativity: III Classification in practice" by Pollney, Skea, and

d'Inverno, Class. Quant. Grav. 17 2885-2902 (2000). Except for some simple test cases, the implementation is untested as of December 19, 2004, and is likely to contain errors.

28.2.6 Torsion and nonmetricity

`ctensor` has the ability to compute and include torsion and nonmetricity coefficients in the connection coefficients.

The torsion coefficients are calculated from a user-supplied tensor `tr`, which should be a rank (2,1) tensor. From this, the torsion coefficients `kt` are computed according to the following formulae:

```
                 m           m       m
         - g  tr    - g   tr    - tr   g
            im  kj     jm   ki      ij  km
kt    = -------------------------------
  ijk                 2


  k      km
kt    = g   kt
  ij          ijm
```

Note that only the mixed-index tensor is calculated and stored in the array `kt`.

The nonmetricity coefficients are calculated from the user-supplied nonmetricity vector `nm`. From this, the nonmetricity coefficients `nmc` are computed as follows:

```
                k    k         km
         -nm  D   - D  nm   + g   nm  g
   k        i  j    i   j          m  ij
nmc   = -----------------------------
   ij                2
```

where D stands for the Kronecker-delta.

When `ctorsion_flag` is set to `true`, the values of `kt` are substracted from the mixed-indexed connection coefficients computed by `christof` and stored in `mcs`. Similarly, if `cnonmet_flag` is set to `true`, the values of `nmc` are substracted from the mixed-indexed connection coefficients.

If necessary, `christof` calls the functions `contortion` and `nonmetricity` in order to compute `kt` and `nm`.

contortion (*tr*) Function

Computes the (2,1) contortion coefficients from the torsion tensor *tr*.

nonmetricity (*nm*) Function

Computes the (2,1) nonmetricity coefficients from the nonmetricity vector *nm*.

28.2.7 Miscellaneous features

ctransform (*M*) Function

A function in the `ctensor` (component tensor) package which will perform a coordinate transformation upon an arbitrary square symmetric matrix *M*. The user must input the functions which define the transformation. (Formerly called `transform`.)

findde (*A*, *n*) Function

returns a list of the unique differential equations (expressions) corresponding to the elements of the *n* dimensional square array *A*. Presently, *n* may be 2 or 3. `deindex` is a global list containing the indices of *A* corresponding to these unique differential equations. For the Einstein tensor (`ein`), which is a two dimensional array, if computed for the metric in the example below, `findde` gives the following independent differential equations:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) derivabbrev:true;
(%o2)                                true
(%i3) dim:4;
(%o3)                                  4
(%i4) lg:matrix([a, 0, 0, 0], [ 0, x^2, 0, 0],
                              [0, 0, x^2*sin(y)^2, 0], [0,0,0,-d]);
                           [ a  0       0        0  ]
                           [                        ]
                           [     2                  ]
                           [ 0  x       0        0  ]
(%o4)                      [                        ]
                           [         2    2         ]
                           [ 0  0   x  sin (y)   0  ]
                           [                        ]
                           [ 0  0       0       - d ]
(%i5) depends([a,d],x);
(%o5)                            [a(x), d(x)]
(%i6) ct_coords:[x,y,z,t];
(%o6)                            [x, y, z, t]
(%i7) cmetric();
(%o7)                               done
(%i8) einstein(false);
(%o8)                               done
(%i9) findde(ein,2);
                                                2
(%o9) [d  x - a d + d, 2 a d d    x - a (d )  x - a  d d  x
       x                     x x         x        x    x

                                                2          2
                          + 2 a d d   - 2 a  d , a  x + a  - a]
                                  x       x      x
(%i10) deindex;
```

```
(%o10)                    [[1, 1], [2, 2], [4, 4]]
```

cograd () Function

Computes the covariant gradient of a scalar function allowing the user to choose the corresponding vector name as the example under `contragrad` illustrates.

contragrad () Function

Computes the contravariant gradient of a scalar function allowing the user to choose the corresponding vector name as the example below for the Schwarzschild metric illustrates:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) derivabbrev:true;
(%o2)                                true
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3)                                done
(%i4) depends(f,r);
(%o4)                               [f(r)]
(%i5) cograd(f,g1);
(%o5)                                done
(%i6) listarray(g1);
(%o6)                           [0, f , 0, 0]
                                     r
(%i7) contragrad(f,g2);
(%o7)                                done
(%i8) listarray(g2);
                                f  r - 2 f  m
                                 r        r
(%o8)                     [0, -------------, 0, 0]
                                      r
```

dscalar () Function

computes the tensor d'Alembertian of the scalar function once dependencies have been declared upon the function. For example:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) derivabbrev:true;
(%o2)                                true
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3)                                done
(%i4) depends(p,r);
(%o4)                               [p(r)]
(%i5) factor(dscalar(p));
                           2
                  p    r  - 2 m p    r + 2 p  r - 2 m p
```

```
                          r r          r r        r          r
(%o5)                    ------------------------------------
                                          2
                                         r
```

checkdiv () Function

computes the covariant divergence of the mixed second rank tensor (whose first index must be covariant) by printing the corresponding n components of the vector field (the divergence) where n = `dim`. If the argument to the function is `g` then the divergence of the Einstein tensor will be formed and must be zero. In addition, the divergence (vector) is given the array name `div`.

cgeodesic (*dis*) Function

A function in the `ctensor` (component tensor) package. `cgeodesic` computes the geodesic equations of motion for a given metric. They are stored in the array `geod[i]`. If the argument *dis* is `true` then these equations are displayed.

bdvac (*f*) Function

generates the covariant components of the vacuum field equations of the Brans- Dicke gravitational theory. The scalar field is specified by the argument *f*, which should be a (quoted) function name with functional dependencies, e.g., `'p(x)`.

The components of the second rank covariant field tensor are represented by the array `bd`.

invariant1 () Function

generates the mixed Euler- Lagrange tensor (field equations) for the invariant density of R^2. The field equations are the components of an array named `inv1`.

invariant2 () Function

*** NOT YET IMPLEMENTED ***

generates the mixed Euler- Lagrange tensor (field equations) for the invariant density of `ric[i,j]*uriem[i,j]`. The field equations are the components of an array named `inv2`.

bimetric () Function

*** NOT YET IMPLEMENTED ***

generates the field equations of Rosen's bimetric theory. The field equations are the components of an array named `rosen`.

28.2.8 Utility functions

diagmatrixp (*M*) Function

Returns `true` if *M* is a diagonal matrix or (2D) array.

symmetricp (*M*) Function

Returns `true` if *M* is a symmetric matrix or (2D) array.

ntermst (*f*) Function

gives the user a quick picture of the "size" of the doubly subscripted tensor (array) *f*. It prints two element lists where the second element corresponds to NTERMS of the components specified by the first elements. In this way, it is possible to quickly find the non-zero expressions and attempt simplification.

cdisplay (*ten*) Function

displays all the elements of the tensor *ten*, as represented by a multidimensional array. Tensors of rank 0 and 1, as well as other types of variables, are displayed as with `ldisplay`. Tensors of rank 2 are displayed as 2-dimensional matrices, while tensors of higher rank are displayed as a list of 2-dimensional matrices. For instance, the Riemann-tensor of the Schwarzschild metric can be viewed as:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) ratfac:true;
(%o2)                                true
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3)                                done
(%i4) riemann(false);
(%o4)                                done
(%i5) cdisplay(riem);
                [ 0               0                   0           0     ]
                [                                                       ]
                [                                  2                    ]
                [      3 m (r - 2 m)   m    2 m                         ]
                [ 0  - ------------- + -- - ----      0           0     ]
                [            4          3     4                         ]
                [           r          r     r                          ]
                [                                                       ]
riem    = [                                  m (r - 2 m)               ]
    1, 1  [ 0               0               -----------       0     ]
                [                                    4                  ]
                [                                   r                   ]
                [                                                       ]
                [                                           m (r - 2 m) ]
                [ 0               0                   0     -----------]
                [                                                4      ]
                [                                               r       ]

                                      [    2 m (r - 2 m)       ]
                                      [ 0  -------------  0  0 ]
                                      [           4           ]
                                      [          r            ]
                           riem     = [                       ]
                               1, 2   [ 0         0        0  0 ]
                                      [                       ]
                                      [ 0         0        0  0 ]
                                      [                       ]
                                      [ 0         0        0  0 ]
```

$$\text{riem}_{1,3} = \begin{bmatrix} 0 & 0 & -\frac{m\,(r-2\,m)}{r^4} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{riem}_{1,4} = \begin{bmatrix} 0 & 0 & 0 & \frac{m\,(r-2\,m)}{r^4} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{riem}_{2,1} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ -\frac{2\,m}{r^2\,(r-2\,m)} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{riem}_{2,2} = \begin{bmatrix} \frac{2\,m}{r^2\,(r-2\,m)} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{m}{r^2\,(r-2\,m)} & 0 \\ 0 & 0 & 0 & -\frac{m}{r^2} \end{bmatrix}$$

```
[                                  r  (r - 2 m) ]

                  [ 0  0       0        0 ]
                  [                       ]
                  [            m          ]
                  [ 0  0  ------------  0 ]
riem      =       [        2              ]
    2, 3          [       r  (r - 2 m)    ]
                  [                       ]
                  [ 0  0       0        0 ]
                  [                       ]
                  [ 0  0       0        0 ]

                  [ 0  0  0       0       ]
                  [                       ]
                  [               m       ]
                  [ 0  0  0  ------------ ]
riem      =       [            2          ]
    2, 4          [           r  (r - 2 m) ]
                  [                       ]
                  [ 0  0  0       0       ]
                  [                       ]
                  [ 0  0  0       0       ]

                        [ 0  0  0  0 ]
                        [            ]
                        [ 0  0  0  0 ]
                        [            ]
riem      =             [ m          ]
    3, 1                [ -  0  0  0 ]
                        [ r          ]
                        [            ]
                        [ 0  0  0  0 ]

                        [ 0  0  0  0 ]
                        [            ]
                        [ 0  0  0  0 ]
                        [            ]
riem      =             [    m       ]
    3, 2                [ 0  -  0  0 ]
                        [    r       ]
                        [            ]
                        [ 0  0  0  0 ]

                    [   m                    ]
                    [ - -   0   0       0    ]
                    [   r                    ]
                    [                        ]
                    [       m                ]
```

```
                  [  0   - -  0       0      ]
                  [         r                ]
riem      = [     [                          ]
    3, 3  [     [  0    0   0       0      ]
          [     [                          ]
          [     [              2 m - r     ]
          [     [  0    0   0  ------- + 1 ]
          [     [                 r        ]

                     [ 0  0  0    0   ]
                     [                ]
                     [ 0  0  0    0   ]
                     [                ]
      riem     = [   [            2 m ]
          3, 4   [   [ 0  0  0  - --- ]
                 [   [             r  ]
                 [   [                ]
                 [   [ 0  0  0    0   ]

                  [       0        0  0  0 ]
                  [                        ]
                  [       0        0  0  0 ]
                  [                        ]
 riem      = [    [       0        0  0  0 ]
     4, 1   [    [                        ]
            [    [      2                 ]
            [    [ m sin (theta)          ]
            [    [ -------------  0  0  0 ]
            [    [       r                ]

                  [ 0        0        0  0 ]
                  [                        ]
                  [ 0        0        0  0 ]
                  [                        ]
 riem      = [    [ 0        0        0  0 ]
     4, 2   [    [                        ]
            [    [        2               ]
            [    [    m sin (theta)       ]
            [    [ 0  -------------  0  0 ]
            [    [          r             ]

                [ 0  0          0          0 ]
                [                            ]
                [ 0  0          0          0 ]
                [                            ]
riem      = [   [ 0  0          0          0 ]
    4, 3    [   [                          ]
            [   [               2            ]
            [   [        2 m sin (theta)     ]
```

```
                                                  [ 0  0  - ---------------  0 ]
                                                  [                 r          ]

                  [        2                                                  ]
                  [   m sin (theta)                                           ]
                  [ - -------------          0                  0          0 ]
                  [         r                                                 ]
                  [                                                           ]
                  [                          2                                ]
                  [                    m sin (theta)                          ]
riem      =       [        0          - -------------          0          0 ]
    4, 4          [                           r                               ]
                  [                                                           ]
                  [                                                2          ]
                  [                                         2 m sin (theta)   ]
                  [        0                 0              ---------------  0 ]
                  [                                                r          ]
                  [                                                           ]
                  [        0                 0                  0          0 ]

(%o5)                                    done
```

deleten (*L*, *n*) Function

Returns a new list consisting of *L* with the *n*'th element deleted.

28.2.9 Variables used by `ctensor`

dim Option variable

Default value: 4

An option in the `ctensor` (component tensor) package. `dim` is the dimension of the manifold with the default 4. The command `dim: n` will reset the dimension to any other value `n`.

diagmetric Option variable

Default value: `false`

An option in the `ctensor` (component tensor) package. If `diagmetric` is `true` special routines compute all geometrical objects (which contain the metric tensor explicitly) by taking into consideration the diagonality of the metric. Reduced run times will, of course, result. Note: this option is set automatically by `csetup` if a diagonal metric is specified.

ctrgsimp Option variable

Causes trigonometric simplifications to be used when tensors are computed. Presently, `ctrgsimp` affects only computations involving a moving frame.

cframe_flag Option variable

Causes computations to be performed relative to a moving frame as opposed to a holonomic metric. The frame is defined by the inverse frame array `fri` and the frame metric `lfg`. For computations using a Cartesian frame, `lfg` should be the unit matrix of the appropriate dimension; for computations in a Lorentz frame, `lfg` should have the appropriate signature.

ctorsion_flag Option variable

Causes the contortion tensor to be included in the computation of the connection coefficients. The contortion tensor itself is computed by `contortion` from the user-supplied tensor `tr`.

cnonmet_flag Option variable

Causes the nonmetricity coefficients to be included in the computation of the connection coefficients. The nonmetricity coefficients are computed from the user-supplied nonmetricity vector `nm` by the function `nonmetricity`.

ctayswitch Option variable

If set to `true`, causes some `ctensor` computations to be carried out using Taylor-series expansions. Presently, `christof`, `ricci`, `uricci`, `einstein`, and `weyl` take into account this setting.

ctayvar Option variable

Variable used for Taylor-series expansion if `ctayswitch` is set to `true`.

ctaypov Option variable

Maximum power used in Taylor-series expansion when `ctayswitch` is set to `true`.

ctaypt Option variable

Point around which Taylor-series expansion is carried out when `ctayswitch` is set to `true`.

gdet System variable

The determinant of the metric tensor `lg`. Computed by `cmetric` when `cframe_flag` is set to `false`.

ratchristof Option variable

Causes rational simplification to be applied by `christof`.

rateinstein Option variable

Default value: `true`

If `true` rational simplification will be performed on the non-zero components of Einstein tensors; if `ratfac` is `true` then the components will also be factored.

ratriemann Option variable

Default value: `true`

One of the switches which controls simplification of Riemann tensors; if `true`, then rational simplification will be done; if `ratfac` is `true` then each of the components will also be factored.

ratweyl Option variable

Default value: `true`

If `true`, this switch causes the `weyl` function to apply rational simplification to the values of the Weyl tensor. If `ratfac` is `true`, then the components will also be factored.

lfg Variable

The covariant frame metric. By default, it is initialized to the 4-dimensional Lorentz frame with signature (+,+,+,-). Used when `cframe_flag` is `true`.

ufg Variable

The inverse frame metric. Computed from `lfg` when `cmetric` is called while `cframe_flag` is set to `true`.

riem Variable

The (3,1) Riemann tensor. Computed when the function `riemann` is invoked. For information about index ordering, see the description of `riemann`.

If `cframe_flag` is `true`, `riem` is computed from the covariant Riemann-tensor `lriem`.

lriem Variable

The covariant Riemann tensor. Computed by `lriemann`.

uriem Variable

The contravariant Riemann tensor. Computed by `uriemann`.

ric Variable

The mixed Ricci-tensor. Computed by `ricci`.

uric Variable

The contravariant Ricci-tensor. Computed by `uricci`.

lg Variable

The metric tensor. This tensor must be specified (as a `dim` by `dim` matrix) before other computations can be performed.

ug Variable

The inverse of the metric tensor. Computed by `cmetric`.

weyl Variable

The Weyl tensor. Computed by `weyl`.

fb Variable

Frame bracket coefficients, as computed by `frame_bracket`.

kinvariant Variable

The Kretchmann invariant. Computed by `rinvariant`.

np Variable

A Newman-Penrose null tetrad. Computed by `nptetrad`.

npi Variable

The raised-index Newman-Penrose null tetrad. Computed by `nptetrad`. Defined as `ug.np`. The product `np.transpose(npi)` is constant:

```
(%i39) trigsimp(np.transpose(npi));
                                [  0   - 1  0  0 ]
                                [                ]
                                [ - 1   0   0  0 ]
(%o39)                          [                ]
                                [  0    0   0  1 ]
                                [                ]
                                [  0    0   1  0 ]
```

tr Variable

User-supplied rank-3 tensor representing torsion. Used by `contortion`.

kt Variable

The contortion tensor, computed from `tr` by `contortion`.

nm Variable

User-supplied nonmetricity vector. Used by `nonmetricity`.

nmc Variable

The nonmetricity coefficients, computed from `nm` by `nonmetricity`.

tensorkill System variable

Variable indicating if the tensor package has been initialized. Set and used by `csetup`, reset by `init_ctensor`.

ct_coords Option variable

Default value: `[]`

An option in the `ctensor` (component tensor) package. `ct_coords` contains a list of coordinates. While normally defined when the function `csetup` is called, one may redefine the coordinates with the assignment `ct_coords: [j1, j2, ..., jn]` where the j's are the new coordinate names. See also `csetup`.

28.2.10 Reserved names

The following names are used internally by the `ctensor` package and should not be redefined:

Name	Description
_lg()	Evaluates to lfg if frame metric used, lg otherwise
_ug()	Evaluates to ufg if frame metric used, ug otherwise
cleanup()	Removes items drom the deindex list
contract4()	Used by psi()
filemet()	Used by csetup() when reading the metric from a file
findde1()	Used by findde()
findde2()	Used by findde()
findde3()	Used by findde()
kdelt()	Kronecker-delta (not generalized)
newmet()	Used by csetup() for setting up a metric interactively
setflags()	Used by init_ctensor()
readvalue()	
resimp()	
sermet()	Used by csetup() for entering a metric as Taylor-series
txyzsum()	
tmetric()	Frame metric, used by cmetric() when cframe_flag:true
triemann()	Riemann-tensor in frame base, used when cframe_flag:true
tricci()	Ricci-tensor in frame base, used when cframe_flag:true
trrc()	Ricci rotation coefficients, used by christof()
yesp()	

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.

New Name	Old Name	Description
ctaylor()	DLGTAYLOR()	Taylor-series expansion of an expression
lgeod[]	EM	Geodesic equations
ein[]	G[]	Mixed Einstein-tensor
ric[]	LR[]	Mixed Ricci-tensor
ricci()	LRICCICOM()	Compute the mixed Ricci-tensor
ctaypov	MINP	Maximum power in Taylor-series expansion
cgeodesic()	MOTION	Compute geodesic equations
ct_coords	OMEGA	Metric coordinates
ctayvar	PARAM	Taylor-series expansion variable
lriem[]	R[]	Covariant Riemann-tensor
uriemann()	RAISERIEMANN()	Compute the contravariant Riemann-tensor
ratriemann	RATRIEMAN	Rational simplif. of the Riemann-tensor
uric[]	RICCI[]	Contravariant Ricci-tensor
uricci()	RICCICOM()	Compute the contravariant Ricci-tensor
cmetric()	SETMETRIC()	Set up the metric

ctaypt	TAYPT	Point for Taylor-series expansion
ctayswitch	TAYSWITCH	Taylor-series setting switch
csetup()	TSETUP()	Start interactive setup session
ctransform()	TTRANSFORM()	Interactive coordinate transformation
uriem[]	UR[]	Contravariant Riemann-tensor
weyl[]	W[]	(3,1) Weyl-tensor

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     j = v     k = v  . v
             1         2         1    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);
                                [ 0  0  0  0 ]
                                [            ]
                                [ 0  0  0  0 ]
(%o5)                           [            ]
                                [ 0  0  0  0 ]
                                [            ]
                                [ 0  0  0  0 ]
(%i6) q[1,1]:1;
(%o6)                                 1
(%i7) for i thru adim do q[1,i+1]:q[i+1,1]:v[i];
(%o7)                                done
(%i8) q[1,4]:q[4,1]:v[1].v[2];
(%o8)                               v  . v
                                     1    2
(%i9) for i from 2 thru 4 do for j from 2 thru 4 do
      q[i,j]:atensimp(q[i,1].q[1,j]);
(%o9)                                done
(%i10) q;
                  [    1        v         v      v  . v  ]
```

```
                    [              1          2      1    2 ]
                    [                                      ]
                    [   v          - 1      v  . v    - v   ]
                    [    1                    1    2      2  ]
(%o10)              [                                      ]
                    [   v       - v  . v      - 1      v    ]
                    [    2         1    2                1   ]
                    [                                      ]
                    [ v  . v       v         - v        - 1  ]
                    [  1    2      2           1             ]
```

`atensor` recognizes as base vectors indexed symbols, where the symbol is that stored in `asymbol` and the index runs between 1 and `adim`. For indexed symbols, and indexed symbols only, the bilinear forms `sf`, `af`, and `av` are evaluated. The evaluation substitutes the value of `aform[i,j]` in place of `fun(v[i],v[j])` where `v` represents the value of `asymbol` and `fun` is either `af` or `sf`; or, it substitutes `v[aform[i,j]]` in place of `av(v[i],v[j])`.

Needless to say, the functions `sf`, `af` and `av` can be redefined.

When the `atensor` package is loaded, the following flags are set:

```
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, degenerate, and negative dimensions of the algebra, respectively. If any *opt_dims* values are supplied, `atensor` will configure the values of `adim` and `aform` appropriately. Otherwise, `adim` will default to 0 and `aform` will not be defined.

`symmetric`: The symmetric algebra is defined by the commutation relation `u.v-v.u=0`.

`symplectic`: The symplectic algebra is defined by the commutation relation `u.v-v.u=2*af(u,v)` where `af` is an antisymmetric scalar-valued function. For the symplectic algebra, *opt_dims* can be up to two nonnegative integers, representing the nondegenerate and degenerate dimensions, respectively. If any *opt_dims* values are

supplied, `atensor` will configure the values of `adim` and `aform` appropriately. Otherwise, `adim` will default to 0 and `aform` will not be defined.

`lie_envelop`: The algebra of the Lie envelope is defined by the commutation relation `u.v-v.u=2*av(u,v)` where `av` is an antisymmetric function.

The `init_atensor` function also recognizes several predefined algebra types:

`complex` implements the algebra of complex numbers as the Clifford algebra Cl(0,1). The call `init_atensor(complex)` is equivalent to `init_atensor(clifford,0,0,1)`.

`quaternion` implements the algebra of quaternions. The call `init_atensor(quaternion)` is equivalent to `init_atensor(clifford,0,0,2)`.

`pauli` implements the algebra of Pauli-spinors as the Clifford-algebra Cl(3,0). A call to `init_atensor(pauli)` is equivalent to `init_atensor(clifford,3)`.

`dirac` implements the algebra of Dirac-spinors as the Clifford-algebra Cl(3,1). A call to `init_atensor(dirac)` is equivalent to `init_atensor(clifford,3,0,1)`.

atensimp (*expr*) Function

Simplifies an algebraic tensor expression *expr* according to the rules configured by a call to `init_atensor`. Simplification includes recursive application of commutation relations and resolving calls to `sf`, `af`, and `av` where applicable. A safeguard is used to ensure that the function always terminates, even for complex expressions.

alg_type Function

The algebra type. Valid values are `universal`, `grassmann`, `clifford`, `symmetric`, `symplectic` and `lie_envelop`.

adim Variable

Default value: 0

The dimensionality of the algebra. `atensor` uses the value of `adim` to determine if an indexed object is a valid base vector. See `abasep`.

aform Variable

Default value: `ident(3)`

Default values for the bilinear forms `sf`, `af`, and `av`. The default is the identity matrix `ident(3)`.

asymbol Variable

Default value: `v`

The symbol for base vectors.

sf (*u*, *v*) Function

A symmetric scalar function that is used in commutation relations. The default implementation checks if both arguments are base vectors using `abasep` and if that is the case, substitutes the corresponding value from the matrix `aform`.

af (*u*, *v*) Function

An antisymmetric scalar function that is used in commutation relations. The default implementation checks if both arguments are base vectors using `abasep` and if that is the case, substitutes the corresponding value from the matrix `aform`.

av (*u*, *v*) Function

An antisymmetric function that is used in commutation relations. The default implementation checks if both arguments are base vectors using `abasep` and if that is the case, substitutes the corresponding value from the matrix `aform`.

For instance:

```
(%i1) load(atensor);
(%o1)       /share/tensor/atensor.mac
(%i2) adim:3;
(%o2)                                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*) Function

Checks if its argument is an `atensor` base vector. That is, if it is an indexed symbol, with the symbol being the same as the value of `asymbol`, and the index having a numeric value between 1 and `adim`.

30 Series

30.1 Introduction to Series

Maxima contains functions `taylor` and `powerseries` for finding the series of differentiable functions. It also has tools such as `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

cauchysum Option variable

Default value: `false`

When multiplying together sums with `inf` as their upper limit, if `sumexpand` is `true` and `cauchysum` is `true` then the Cauchy product will be used rather than the usual product. In the Cauchy product the index of the inner summation is a function of the index of the outer one rather than varying independently.

Example:

```
(%i1) sumexpand: false$
(%i2) cauchysum: false$
(%i3) s: sum (f(i), i, 0, inf) * sum (g(j), j, 0, inf);
                       inf          inf
                       ====         ====
                       \            \
(%o3)                  ( >    f(i)) >    g(j)
                       /            /
                       ====         ====
                       i = 0        j = 0
(%i4) sumexpand: true$
(%i5) cauchysum: true$
(%i6) ''s;
                  inf     i1
                  ====   ====
                  \      \
(%o6)              >      >     g(i1 - i2) f(i2)
                  /      /
                  ====   ====
                  i1 = 0 i2 = 0
```

deftaylor (*f_1*(*x_1*), *expr_1*, ..., *f_n*(*x_n*), *expr_n*) Function

For each function *f_i* of one variable *x_i*, `deftaylor` defines *expr_i* as the Taylor series about zero. *expr_i* is typically a polynomial in *x_i* or a summation; more general expressions are accepted by `deftaylor` without complaint.

`powerseries` (*f_i*(*x_i*), *x_i*, 0) returns the series defined by `deftaylor`.

`deftaylor` returns a list of the functions *f_1*, ..., *f_n*. `deftaylor` evaluates its arguments.

Example:

```
(%i1) deftaylor (f(x), x^2 + sum(x^i/(2^i*i!^2), i, 4, inf));
(%o1)                          [f]
(%i2) powerseries (f(x), x, 0);
                       inf
                       ====      i1
                       \        x         2
(%o2)                   >     -------- + x
                       /       i1    2
                       ====   2   i1!
                       i1 = 4
(%i3) taylor (exp (sqrt (f(x))), x, 0, 4);
                       2         3          4
                      x    3073 x    12817 x
(%o3)/T/     1 + x + -- + ------- + -------- + . . .
                      2     18432     307200
```

maxtayorder Option variable

Default value: `true`

When `maxtayorder` is `true`, then during algebraic manipulation of (truncated) Taylor series, `taylor` tries to retain as many terms as are known to be correct.

niceindices (*expr*) Function

Renames the indices of sums and products in *expr*. `niceindices` attempts to rename each index to the value of `niceindicespref[1]`, unless that name appears in the summand or multiplicand, in which case `niceindices` tries the succeeding elements of `niceindicespref` in turn, until an unused variable is found. If the entire list is exhausted, additional indices are constructed by appending integers to the value of `niceindicespref[1]`, e.g., `i0`, `i1`, `i2`, ...

`niceindices` returns an expression. `niceindices` evaluates its argument.

Example:

```
(%i1) niceindicespref;
(%o1)                   [i, j, k, l, m, n]
(%i2) product (sum (f (foo + i*j*bar), foo, 1, inf), bar, 1, inf);
                 inf    inf
                /===\   ====
                 ! !    \
(%o2)            ! !     >      f(bar i j + foo)
                 ! !    /
                bar = 1 ====
                        foo = 1
(%i3) niceindices (%);
                     inf  inf
                    /===\ ====
                     ! !  \
```

```
(%o3)                ! !   >    f(i j l + k)
                     ! !  /
                    l = 1 ====
                          k = 1
```

niceindicespref Option variable

Default value: `[i, j, k, l, m, n]`

`niceindicespref` is the list from which `niceindices` takes the names of indices for sums and products.

The elements of `niceindicespref` are typically names of variables, although that is not enforced by `niceindices`.

Example:

```
(%i1) niceindicespref: [p, q, r, s, t, u]$
(%i2) product (sum (f (foo + i*j*bar), foo, 1, inf), bar, 1, inf);
                  inf    inf
                 /===\   ====
                  ! !    \
(%o2)             ! !     >      f(bar i j + foo)
                  ! !    /
                 bar = 1 ====
                         foo = 1
(%i3) niceindices (%);
                     inf  inf
                    /===\ ====
                     ! !  \
(%o3)                ! !   >    f(i j q + p)
                     ! !  /
                    q = 1 ====
                          p = 1
```

nusum (*expr, x, i_0, i_1*) Function

Carries out indefinite hypergeometric summation of *expr* with respect to *x* using a decision procedure due to R.W. Gosper. *expr* and the result must be expressible as products of integer powers, factorials, binomials, and rational functions.

The terms "definite" and "indefinite summation" are used analogously to "definite" and "indefinite integration". To sum indefinitely means to give a symbolic result for the sum over intervals of variable length, not just e.g. 0 to inf. Thus, since there is no formula for the general partial sum of the binomial series, `nusum` can't do it.

`nusum` and `unsum` know a little about sums and differences of finite products. See also `unsum`.

Examples:

```
(%i1) nusum (n*n!, n, 0, n);

Dependent equations eliminated:  (1)
(%o1)                      (n + 1)! - 1
(%i2) nusum (n^4*4^n/binomial(2*n,n), n, 0, n);
```

```
                      4        3       2              n
      2 (n + 1) (63 n  + 112 n  + 18 n  - 22 n + 3) 4      2
(%o2) ------------------------------------------------ - ------
                    693 binomial(2 n, n)                3 11 7
(%i3) unsum (%, n);
                               4  n
                              n  4
(%o3)                    ----------------
                         binomial(2 n, n)
(%i4) unsum (prod (i^2, i, 1, n), n);
                     n - 1
                     /===\
                      ! !   2
(%o4)                ( ! !  i ) (n - 1) (n + 1)
                      ! !
                     i = 1
(%i5) nusum (%, n, 1, n);

Dependent equations eliminated:  (2 3)
                              n
                            /===\
                             ! !   2
(%o5)                        ! !  i  - 1
                             ! !
                            i = 1
```

pade (*taylor_series*, *numer_deg_bound*, *denom_deg_bound*) Function

Returns a list of all rational functions which have the given Taylor series expansion where the sum of the degrees of the numerator and the denominator is less than or equal to the truncation level of the power series, i.e. are "best" approximants, and which additionally satisfy the specified degree bounds.

taylor_series is a univariate Taylor series. *numer_deg_bound* and *denom_deg_bound* are positive integers specifying degree bounds on the numerator and denominator.

taylor_series can also be a Laurent series, and the degree bounds can be `inf` which causes all rational functions whose total degree is less than or equal to the length of the power series to be returned. Total degree is defined as *numer_deg_bound* + *denom_deg_bound*. Length of a power series is defined as `"truncation level" + 1 - min(0, "order of series")`.

```
(%i1) taylor (1 + x + x^2 + x^3, x, 0, 3);
                                2    3
(%o1)/T/             1 + x + x  + x  + . . .
(%i2) pade (%, 1, 1);
                                 1
(%o2)                        [- -----]
                               x - 1
(%i3) t: taylor(-(83787*x^10 - 45552*x^9 - 187296*x^8
                  + 387072*x^7 + 86016*x^6 - 1507328*x^5
                  + 1966080*x^4 + 4194304*x^3 - 25165824*x^2
```

```
                    + 67108864*x - 134217728)
       /134217728, x, 0, 10);
                      2    3       4       5       6        7
              x   3 x    x    15 x    23 x    21 x    189 x
(%o3)/T/ 1 - - + ---- - -- - ----- + ----- - ----- - ------
              2    16    32   1024    2048    32768   65536

                                    8         9          10
                               5853 x    2847 x    83787 x
                            + ------- + ------- - --------- + . . .
                               4194304   8388608   134217728
(%i4) pade (t, 4, 4);
(%o4)                          []
```

There is no rational function of degree 4 numerator/denominator, with this power series expansion. You must in general have degree of the numerator and degree of the denominator adding up to at least the degree of the power series, in order to have enough unknown coefficients to solve.

```
(%i5) pade (t, 5, 5);
                       5                 4                   3
(%o5) [- (520256329 x  - 96719020632 x  - 489651410240 x

                     2
 - 1619100813312 x  - 2176885157888 x - 2386516803584)

                  5                  4                    3
/(47041365435 x  + 381702613848 x  + 1360678489152 x

                     2
 + 2856700692480 x  + 3370143559680 x + 2386516803584)]
```

powerdisp Option variable

Default value: `false`

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

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

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).

If `powerseries` is unable to expand *expr*, `taylor` may give the first several terms of the series.

When `verbose` is `true`, `powerseries` prints progress messages.

```
(%i1) verbose: true$
(%i2) powerseries (log(sin(x)/x), x, 0);
can't expand
```

```
                                    log(sin(x))
so we'll try again after applying the rule:
                                          d
                                       / -- (sin(x))
                                       [ dx
                          log(sin(x)) = i ----------- dx
                                       ]   sin(x)
                                       /
in the first simplification we have returned:
                                /
                                [
                                i cot(x) dx - log(x)
                                ]
                                /
                        inf
                        ====        i1  2 i1             2 i1
                        \      (- 1)   2     bern(2 i1) x
                         >     ------------------------------
                        /                i1 (2 i1)!
                        ====
                        i1 = 1
(%o2)                    ------------------------------------
                                        2
```

psexpand Option variable

Default value: `false`

When `psexpand` is `true`, an extended rational function expression is displayed fully expanded. The switch `ratexpand` has the same effect.

When `psexpand` is `false`, a multivariate expression is displayed just as in the rational function package.

When `psexpand` is `multi`, then terms with the same total degree in the variables are grouped together.

revert (*expr*, *x*) Function

revert2 (*expr*, *x*, *n*) Function

These functions return the reversion of *expr*, a Taylor series about zero in the variable *x*. `revert` returns a polynomial of degree equal to the highest power in *expr*. `revert2` returns a polynomial of degree *n*, which may be greater than, equal to, or less than the degree of *expr*.

`load ("revert")` loads these functions.

Examples:

```
(%i1) load ("revert")$
(%i2) t: taylor (exp(x) - 1, x, 0, 6);
                     2    3    4    5     6
                    x    x    x    x     x
(%o2)/T/      x + -- + -- + -- + --- + --- + . . .
                    2    6    24   120   720
```

```
(%i3) revert (t, x);
                6       5       4       3       2
           10 x  - 12 x  + 15 x  - 20 x  + 30 x  - 60 x
(%o3)/R/ - ----------------------------------------------
                                60
(%i4) ratexpand (%);
                     6    5    4    3    2
                    x    x    x    x    x
(%o4)             - -- + -- - -- + -- - -- + x
                    6    5    4    3    2
(%i5) taylor (log(x+1), x, 0, 6);
                    2    3    4    5    6
                   x    x    x    x    x
(%o5)/T/       x - -- + -- - -- + -- - -- + . . .
                   2    3    4    5    6
(%i6) ratsimp (revert (t, x) - taylor (log(x+1), x, 0, 6));
(%o6)                           0
(%i7) revert2 (t, x, 4);
                           4    3    2
                          x    x    x
(%o7)                   - -- + -- - -- + x
                          4    3    2
```

taylor (*expr*, *x*, *a*, *n*) Function
taylor (*expr*, [*x_1*, *x_2*, ...], *a*, *n*) Function
taylor (*expr*, [*x*, *a*, *n*, *'asymp*]) Function
taylor (*expr*, [*x_1*, *x_2*, ...], [*a_1*, *a_2*, ...], [*n_1*, *n_2*, ...]) Function
taylor (*expr*, [*x_1*, *a_1*, *n_1*], [*x_2*, *a_2*, *n_2*], ...) Function

`taylor (`*expr*`, `*x*`, `*a*`, `*n*`)` expands the expression *expr* in a truncated Taylor or Laurent series in the variable *x* around the point *a*, containing terms through `(`*x* `-` *a*`)^`*n*.

If *expr* is of the form *f*`(`*x*`)/`*g*`(`*x*`)` and *g*`(`*x*`)` has no terms up to degree *n* then `taylor` attempts to expand *g*`(`*x*`)` up to degree `2` *n*. If there are still no nonzero terms, `taylor` doubles the degree of the expansion of *g*`(`*x*`)` so long as the degree of the expansion is less than or equal to *n* `2^taylordepth`.

`taylor (`*expr*`, [`*x_1*`, `*x_2*`, ...], `*a*`, `*n*`)` returns a truncated power series of degree *n* in all variables *x_1*, *x_2*, ... about the point `(`*a*`, `*a*`, ...)`.

`taylor (`*expr*`, [`*x_1*`, `*a_1*`, `*n_1*`], [`*x_2*`, `*a_2*`, `*n_2*`], ...)` returns a truncated power series in the variables *x_1*, *x_2*, ... about the point `(`*a_1*`, `*a_2*`, ...)`, truncated at *n_1*, *n_2*, ...

`taylor (`*expr*`, [`*x_1*`, `*x_2*`, ...], [`*a_1*`, `*a_2*`, ...], [`*n_1*`, `*n_2*`, ...])` returns a truncated power series in the variables *x_1*, *x_2*, ... about the point `(`*a_1*`, `*a_2*`, ...)`, truncated at *n_1*, *n_2*, ...

`taylor (`*expr*`, [`*x*`, `*a*`, `*n*`, `'asymp`])` returns an expansion of *expr* in negative powers of *x* `-` *a*. The highest order term is `(`*x* `-` *a*`)^-`*n*.

When `maxtayorder` is `true`, then during algebraic manipulation of (truncated) Taylor series, `taylor` tries to retain as many terms as are known to be correct.

When `psexpand` is `true`, an extended rational function expression is displayed fully expanded. The switch `ratexpand` has the same effect. When `psexpand` is `false`, a multivariate expression is displayed just as in the rational function package. When `psexpand` is `multi`, then terms with the same total degree in the variables are grouped together.

See also the `taylor_logexpand` switch for controlling expansion.

Examples:

```
(%i1) taylor (sqrt (sin(x) + a*x + 1), x, 0, 3);
                           2             2
              (a + 1) x   (a  + 2 a + 1) x
(%o1)/T/ 1 + --------- - -----------------
                  2              8

                                    3      2             3
                                (3 a  + 9 a  + 9 a - 1) x
                             + ------------------------- + . . .
                                          48
(%i2) %^2;
                                     3
                                    x
(%o2)/T/           1 + (a + 1) x - -- + . . .
                                    6
(%i3) taylor (sqrt (x + 1), x, 0, 5);
                         2    3      4      5
                  x   x    x    5 x    7 x
(%o3)/T/      1 + - - -- + -- - ---- + ---- + . . .
                  2   8    16   128    256
(%i4) %^2;
(%o4)/T/                  1 + x + . . .
(%i5) product ((1 + x^i)^2.5, i, 1, inf)/(1 + x^2);
                        inf
                       /===\
                        ! !    i     2.5
                        ! !  (x  + 1)
                        ! !
                        i = 1
(%o5)                   -----------------
                              2
                             x  + 1
(%i6) ev (taylor(%, x,  0, 3), keepfloat);
                                2           3
(%o6)/T/    1 + 2.5 x + 3.375 x  + 6.5625 x  + . . .
(%i7) taylor (1/log (x + 1), x, 0, 3);
                                2       3
                1   1   x    x    19 x
(%o7)/T/        - + - - -- + -- - ----- + . . .
                x   2   12   24    720
(%i8) taylor (cos(x) - sec(x), x, 0, 5);
```

```
                                4
                           2   x
(%o8)/T/                - x  - -- + . . .
                           6
(%i9) taylor ((cos(x) - sec(x))^3, x, 0, 5);
(%o9)/T/                    0 + . . .
(%i10) taylor (1/(cos(x) - sec(x))^3, x, 0, 5);
                                                  2          4
           1     1       11      347    6767 x    15377 x
(%o10)/T/ - -- + ---- + ------ - ----- - ------- - --------
           6      4        2   15120   604800    7983360
          x    2 x    120 x

                                                          + . . .
(%i11) taylor (sqrt (1 - k^2*sin(x)^2), x, 0, 6);
             2  2       4      2   4
            k  x    (3 k  - 4 k ) x
(%o11)/T/ 1 - ----- - ---------------
              2           24

                                  6       4       2   6
                            (45 k  - 60 k  + 16 k ) x
                          - ------------------------ + . . .
                                     720
(%i12) taylor ((x + 1)^n, x, 0, 4);
                     2       2     3      2         3
                  (n  - n) x    (n  - 3 n  + 2 n) x
(%o12)/T/ 1 + n x + ---------- + --------------------
                        2                 6

                              4      3       2         4
                           (n  - 6 n  + 11 n  - 6 n) x
                         + --------------------------- + . . .
                                       24
(%i13) taylor (sin (y + x), x, 0, 3, y, 0, 3);
              3                 2
             y                 y
(%o13)/T/ y - -- + . . . + (1 - -- + . . .) x
             6                 2

                    3                       2
               y   y             2      1   y             3
         + (- - + -- + . . .) x  + (- - + -- + . . .) x  + . . .
               2   12                    6   12
(%i14) taylor (sin (y + x), [x, y], 0, 3);
                     3        2      2      3
                    x  + 3 y x  + 3 y  x + y
(%o14)/T/   y + x - ------------------------ + . . .
                           6
```

```
(%i15) taylor (1/sin (y + x), x, 0, 3, y, 0, 3);
          1   y             1    1              1           2
(%o15)/T/ - + - + . . . + (- -- + - + . . .) x + (-- + . . .) x
          y   6              2   6              3
                            y                  y

                                          1           3
                                     + (- -- + . . .) x  + . . .
                                          4
                                         y
(%i16) taylor (1/sin (y + x), [x, y], 0, 3);
                              3         2       2        3
            1     x + y   7 x  + 21 y x  + 21 y  x + 7 y
(%o16)/T/ ----- + ----- + ------------------------------ + . . .
          x + y     6                  360
```

taylordepth 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 `2^taylordepth`.

taylorinfo (*expr*) Function

Returns information about the Taylor series *expr*. The return value is a list of lists. Each list comprises the name of a variable, the point of expansion, and the degree of the expansion.

`taylorinfo` returns `false` if *expr* is not a Taylor series.

Example:

```
(%i1) taylor ((1 - y^2)/(1 - x), x, 0, 3, [y, a, inf]);
                    2                        2
(%o1)/T/ - (y - a)  - 2 a (y - a) + (1 - a )

          2                          2
 + (1 - a  - 2 a (y - a) - (y - a) ) x

          2                          2   2
 + (1 - a  - 2 a (y - a) - (y - a) ) x

          2                          2   3
 + (1 - a  - 2 a (y - a) - (y - a) ) x  + . . .
(%i2) taylorinfo(%);
(%o2)               [[y, a, inf], [x, 0, 3]]
```

taylorp (*expr*) Function

Returns `true` if *expr* is a Taylor series, and `false` otherwise.

taylor_logexpand Option variable

Default value: `true`

`taylor_logexpand` controls expansions of logarithms in `taylor` series.

When `taylor_logexpand` is `true`, all logarithms are expanded fully so that zero-recognition problems involving logarithmic identities do not disturb the expansion process. However, this scheme is not always mathematically correct since it ignores branch information.

When `taylor_logexpand` is set to `false`, then the only expansion of logarithms that occur is that necessary to obtain a formal power series.

taylor_order_coefficients Option variable

Default value: `true`

`taylor_order_coefficients` controls the ordering of coefficients in a Taylor series.

When `taylor_order_coefficients` is `true`, coefficients of taylor series are ordered canonically.

taylor_simplifier (*expr*) Function

Simplifies coefficients of the power series *expr*. `taylor` calls this function.

taylor_truncate_polynomials Option variable

Default value: `true`

When `taylor_truncate_polynomials` is `true`, polynomials are truncated based upon the input truncation levels.

Otherwise, polynomials input to `taylor` are considered to have infinite precison.

taytorat (*expr*) Function

Converts *expr* from `taylor` form to canonical rational expression (CRE) form. The effect is the same as `rat (ratdisrep (`*expr*`))`, but faster.

trunc (*expr*) Function

Annotates the internal representation of the general expression *expr* so that it is displayed as if its sums were truncated Taylor series. *expr* is not otherwise modified.

Example:

```
(%i1) expr: x^2 + x + 1;
                              2
(%o1)                        x  + x + 1
(%i2) trunc (expr);
                                    2
(%o2)                    1 + x + x  + . . .
(%i3) is (expr = trunc (expr));
(%o3)                           true
```

unsum (*f*, *n*) Function

Returns the first backward difference *f*`(`*n*`) - `*f*`(`*n*` - 1)`. Thus `unsum` in a sense is the inverse of `sum`.

See also `nusum`.

Examples:

```
(%i1) g(p) := p*4^n/binomial(2*n,n);
                                      n
                                   p 4
(%o1)                 g(p) := ----------------
                                 binomial(2 n, n)
(%i2) g(n^4);
                                  4  n
                                 n  4
(%o2)                          ----------------
                               binomial(2 n, n)
(%i3) nusum (%, n, 0, n);
                     4        3       2              n
      2 (n + 1) (63 n  + 112 n  + 18 n  - 22 n + 3) 4      2
(%o3) ------------------------------------------------ - ------
                     693 binomial(2 n, n)                3 11 7
(%i4) unsum (%, n);
                                  4  n
                                 n  4
(%o4)                          ----------------
                               binomial(2 n, n)
```

verbose Option variable

Default value: `false`

When `verbose` is `true`, `powerseries` prints progress messages.

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 `zerobern` is `false`.

See also `burn`.

```
(%i1) zerobern: true$
(%i2) map (bern, [0, 1, 2, 3, 4, 5, 6, 7, 8]);
                  1  1       1      1        1
(%o2)       [1, - -, -, 0, - --, 0, --, 0, - --]
                  2  6       30     42       30
(%i3) zerobern: false$
(%i4) map (bern, [0, 1, 2, 3, 4, 5, 6, 7, 8]);
            1  1    1   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 (bfloat); *n* is the number of digits in the return value.

bfhzeta (*s*, *h*, *n*) Function

Returns the Hurwitz zeta function for the arguments *s* and *h*. The return value is a big float (bfloat); *n* is the number of digits in the return value.

The Hurwitz zeta function is defined as

```
sum ((k+h)^-s, k, 0, inf)
```

`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:

```
(%i1) binomial (11, 7);
(%o1)                          330
(%i2) 11! / 7! / (11 - 7)!;
(%o2)                          330
(%i3) binomial (x, 7);
        (x - 6) (x - 5) (x - 4) (x - 3) (x - 2) (x - 1) x
```

```
(%o3)   -----------------------------------------------
                                5040
(%i4) binomial (x + 7, x);
      (x + 1) (x + 2) (x + 3) (x + 4) (x + 5) (x + 6) (x + 7)
(%o4) -------------------------------------------------------
                                 5040
(%i5) binomial (11, y);
(%o5)                     binomial(11, y)
```

burn (*n*) Function

Returns the *n*'th Bernoulli number for integer *n*. `burn` may be more efficient than `bern` for large, isolated *n* (perhaps *n* greater than 105 or so), as `bern` computes all the Bernoulli numbers up to index *n* before returning.

`burn` exploits the observation that (rational) Bernoulli numbers can be approximated by (transcendental) zetas with tolerable efficiency.

`load ("bffac")` loads this function.

cf (*expr*) Function

Converts *expr* into a continued fraction. *expr* is an expression comprising continued fractions and square roots of integers. Operands in the expression may be combined with arithmetic operators. Aside from continued fractions and square roots, factors in the expression must be integer or rational numbers. Maxima does not know about operations on continued fractions outside of `cf`.

`cf` evaluates its arguments after binding `listarith` to `false`. `cf` returns a continued fraction, represented as a list.

A continued fraction `a + 1/(b + 1/(c + ...))` is represented by the list `[a, b, c, ...]`. The list elements `a`, `b`, `c`, ... must evaluate to integers. *expr* may also contain `sqrt (n)` where `n` is an integer. In this case `cf` will give as many terms of the continued fraction as the value of the variable `cflength` times the period.

A continued fraction can be evaluated to a number by evaluating the arithmetic representation returned by `cfdisrep`. See also `cfexpand` for another way to evaluate a continued fraction.

See also `cfdisrep`, `cfexpand`, and `cflength`.

Examples:

- *expr* is an expression comprising continued fractions and square roots of integers.

```
(%i1) cf ([5, 3, 1]*[11, 9, 7] + [3, 7]/[4, 3, 2]);
(%o1)               [59, 17, 2, 1, 1, 1, 27]
(%i2) cf ((3/17)*[1, -2, 5]/sqrt(11) + (8/13));
(%o2)        [0, 1, 1, 1, 3, 2, 1, 4, 1, 9, 1, 9, 2]
```

- `cflength` controls how many periods of the continued fraction are computed for algebraic, irrational numbers.

```
(%i1) cflength: 1$
(%i2) cf ((1 + sqrt(5))/2);
(%o2)                     [1, 1, 1, 1, 2]
(%i3) cflength: 2$
```

```
(%i4) cf ((1 + sqrt(5))/2);
(%o4)               [1, 1, 1, 1, 1, 1, 1, 2]
(%i5) cflength: 3$
(%i6) cf ((1 + sqrt(5))/2);
(%o6)           [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
```

- A continued fraction can be evaluated by evaluating the arithmetic representation returned by `cfdisrep`.

```
(%i1) cflength: 3$
(%i2) cfdisrep (cf (sqrt (3)))$
(%i3) ev (%, numer);
(%o3)                   1.731707317073171
```

- Maxima does not know about operations on continued fractions outside of `cf`.

```
(%i1) cf ([1,1,1,1,1,2] * 3);
(%o1)                      [4, 1, 5, 2]
(%i2) cf ([1,1,1,1,1,2]) * 3;
(%o2)                  [3, 3, 3, 3, 3, 6]
```

cfdisrep (*list*) Function

Constructs and returns an ordinary arithmetic expression of the form `a + 1/(b + 1/(c + ...))` from the list representation of a continued fraction `[a, b, c, ...]`.

```
(%i1) cf ([1, 2, -3] + [1, -2, 1]);
(%o1)                      [1, 1, 1, 2]
(%i2) cfdisrep (%);
                                    1
(%o2)                      1 + ---------
                                      1
                                 1 + -----
                                        1
                                    1 + -
                                        2
```

cfexpand (*x*) Function

Returns a matrix of the numerators and denominators of the last (column 1) and next-to-last (column 2) convergents of the continued fraction *x*.

```
(%i1) cf (rat (ev (%pi, numer)));

'rat' replaced 3.141592653589793 by 103993/33102 =3.141592653011902
(%o1)                    [3, 7, 15, 1, 292]
(%i2) cfexpand (%);
                           [ 103993  355 ]
(%o2)                      [             ]
                           [ 33102   113 ]
(%i3) %[1,1]/%[2,1], numer;
(%o3)                    3.141592653011902
```

cflength Option variable

Default value: 1

`cflength` controls the number of terms of the continued fraction the function `cf` will give, as the value `cflength` times the period. Thus the default is to give one period.

```
(%i1) cflength: 1$
(%i2) cf ((1 + sqrt(5))/2);
(%o2)                    [1, 1, 1, 1, 2]
(%i3) cflength: 2$
(%i4) cf ((1 + sqrt(5))/2);
(%o4)               [1, 1, 1, 1, 1, 1, 1, 2]
(%i5) cflength: 3$
(%i6) cf ((1 + sqrt(5))/2);
(%o6)           [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
```

divsum (*n*, *k*) Function
divsum (*n*) 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 (*n*) 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 ....

factorial (*x*) Function

Represents the factorial function. Maxima treats `factorial (x)` the same as *x*`!`. See `!`.

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.

```
(%i1) map (fib, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
(%o1)         [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

fibtophi (*expr*) Function

Expresses Fibonacci numbers in *expr* in terms of the constant `%phi`, which is `(1 + sqrt(5))/2`, approximately 1.61803399.

Examples:

```
(%i1) fibtophi (fib (n));
                            n             n
                         %phi  - (1 - %phi)
(%o1)                    ------------------
                             2 %phi - 1
(%i2) fib (n-1) + fib (n) - fib (n+1);
(%o2)          - fib(n + 1) + fib(n) + fib(n - 1)
(%i3) fibtophi (%);
             n + 1              n + 1       n             n
          %phi      - (1 - %phi)        %phi  - (1 - %phi)
(%o3) - -------------------------- + ------------------
                2 %phi - 1                2 %phi - 1
                                          n - 1             n - 1
                                       %phi      - (1 - %phi)
                                     + --------------------------
                                                2 %phi - 1
(%i4) ratsimp (%);
(%o4)                           0
```

ifactors (*n*) Function

For a positive integer *n* returns the factorization of *n*. If `n=p1^e1..pk^nk` is the decomposition of *n* into prime factors, ifactors returns `[[p1, e1], ... , [pk, ek]]`.

Factorization methods used are trial divisions by primes up to 9973, Pollard's rho 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)!;
                                  n!
(%o1)                          --------
                               (n + 2)!
(%i2) minfactorial (%);
                                   1
(%o2)                     ---------------
                          (n + 1) (n + 2)
```

next_prime (*n*) Function

Returns the smallest prime bigger than *n*.

```
(%i1) next_prime(27);
(%o1)                         29
```

partfrac (*expr*, *var*) Function

Expands the expression *expr* in partial fractions with respect to the main variable *var*. `partfrac` does a complete partial fraction decomposition. The algorithm employed is based on the fact that the denominators of the partial fraction expansion (the factors of the original denominator) are relatively prime. The numerators can be written as linear combinations of denominators, and the expansion falls out.

```
(%i1) 1/(1+x)^2 - 2/(1+x) + 2/(2+x);
                        2       2        1
(%o1)                 ----- - ----- + --------
                      x + 2   x + 1          2
                                      (x + 1)
(%i2) ratsimp (%);
                                 x
(%o2)                  - ------------------
                            3      2
                           x  + 4 x  + 5 x + 2
(%i3) partfrac (%, x);
                        2       2        1
(%o3)                 ----- - ----- + --------
```

```
                        x + 2   x + 1          2
                                      (x + 1)
```

power_mod (*a*, *n*, *m*) Function

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.

```
(%i1) power_mod(3, 15, 5);
(%o1)                          2
(%i2) mod(3^15,5);
(%o2)                          2
(%i3) power_mod(2, -1, 5);
(%o3)                          3
(%i4) inv_mod(2,5);
(%o4)                          3
```

primep (*n*) Function

Primality test. If `primep (n)` returns `false`, *n* is a composite number and if it returns `true`, *n* is a prime number with very high probability.

For *n* less than 341550071728321 a deterministic version of Miller-Rabin's test is used. If `primep (n)` returns `true`, then *n* is a prime number.

For *n* bigger than 34155071728321 `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* beeing 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*) Function

Returns the greatest prime smaller than *n*.

```
(%i1) prev_prime(27);
(%o1)                         23
```

qunit (*n*) Function

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

```
(%i1) qunit (17);
(%o1)                      sqrt(17) + 4
(%i2) expand (% * (sqrt(17) - 4));
(%o2)                            1
```

totient (*n*) Function

Returns the number of integers less than or equal to *n* which are relatively prime to *n*.

zerobern Option variable

Default value: `true`

When `zerobern` is `false`, `bern` excludes the Bernoulli numbers which are equal to zero. See `bern`.

zeta (*n*) Function

Returns the Riemann zeta function if *x* is a negative integer, 0, 1, or a positive even number, and returns a noun form `zeta (`*n*`)` for all other arguments, including rational noninteger, floating point, and complex arguments.

See also `bfzeta` and `zeta%pi`.

```
(%i1) map (zeta, [-4, -3, -2, -1, 0, 1, 2, 3, 4, 5]);
                                        2             4
          1        1     1       %pi            %pi
(%o1) [0, ---, 0, - --, - -, inf, ----, zeta(3), ----, zeta(5)]
         120       12    2        6              90
```

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`.

```
(%i1) zeta%pi: true$
(%i2) zeta (4);
                                   4
                                %pi
(%o2)                           ----
                                 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.

32.2 Functions and Variables for Symmetries

32.2.1 Changing bases

comp2pui (*n, L*) Function

implements passing from the complete symmetric functions given in the list *L* to the elementary symmetric functions from 0 to *n*. If the list *L* contains fewer than *n+1* elements, it will be completed with formal values of the type *h1*, *h2*, etc. If the first element of the list *L* exists, it specifies the size of the alphabet, otherwise the size is set to *n*.

```
(%i1) comp2pui (3, [4, g]);
                              2                       2
(%o1)     [4, g, 2 h2 - g , 3 h3 - g h2 + g (g  - 2 h2)]
```

ele2pui (*m, L*) Function

goes from the elementary symmetric functions to the complete functions. Similar to `comp2ele` and `comp2pui`.

Other functions for changing bases: `comp2ele`.

ele2comp (*m, L*) Function

Goes from the elementary symmetric functions to the compete functions. Similar to `comp2ele` and `comp2pui`.

Other functions for changing bases: `comp2ele`.

elem (*ele, sym, lvar*) Function

decomposes the symmetric polynomial *sym*, in the variables contained in the list *lvar*, in terms of the elementary symmetric functions given in the list *ele*. If the first element of *ele* is given, it will be the size of the alphabet, otherwise the size will be the degree of the polynomial *sym*. If values are missing in the list *ele*, formal values of the type *e1*, *e2*, etc. will be added. The polynomial *sym* may be given in three different forms: contracted (`elem` should then be 1, its default value), partitioned (`elem` should be 3), or extended (i.e. the entire polynomial, and `elem` should then be 2). The function `pui` is used in the same way.

On an alphabet of size 3 with *e1*, the first elementary symmetric function, with value 7, the symmetric polynomial in 3 variables whose contracted form (which here depends on only two of its variables) is *x^4-2*x*y* decomposes as follows in elementary symmetric functions:

```
(%i1) elem ([3, 7], x^4 - 2*x*y, [x, y]);
(%o1) 7 (e3 - 7 e2 + 7 (49 - e2)) + 21 e3

                                          + (- 2 (49 - e2) - 2) e2
(%i2) ratsimp (%);
                                2
(%o2)             28 e3 + 2 e2  - 198 e2 + 2401
```

Other functions for changing bases: `comp2ele`.

mon2schur (*L*) Function

The list L represents the Schur function S_L: we have $L = [i_1, i_2, \ldots, i_q]$, with $i_1 \leq i_2 \leq \ldots \leq i_q$. The Schur function $S_{i_1,i_2,\ldots,i_q}$ is the minor of the infinite matrix h_{i-j}, $i \geq 1, j \geq 1$, consisting of the q first rows and the columns $i_1 + 1, i_2 + 2, \ldots, i_q + q$.

This Schur function can be written in terms of monomials by using `treinat` and `kostka`. The form returned is a symmetric polynomial in a contracted representation in the variables $x_1, x_2, \ldots$.

```
(%i1) mon2schur ([1, 1, 1]);
(%o1)                        x1 x2 x3
(%i2) mon2schur ([3]);
                                    2        3
(%o2)                x1 x2 x3 + x1  x2 + x1
(%i3) mon2schur ([1, 2]);
                                       2
(%o3)                  2 x1 x2 x3 + x1  x2
```

which means that for 3 variables this gives:

```
2 x1 x2 x3 + x1^2 x2 + x2^2 x1 + x1^2 x3 + x3^2 x1
 + x2^2 x3 + x3^2 x2
```

Other functions for changing bases: `comp2ele`.

multi_elem (*l_elem, multi_pc, l_var*) Function

decomposes a multi-symmetric polynomial in the multi-contracted form *multi_pc* in the groups of variables contained in the list of lists *l_var* in terms of the elementary symmetric functions contained in *l_elem*.

```
(%i1) multi_elem ([[2, e1, e2], [2, f1, f2]], a*x + a^2 + x^3,
      [[x, y], [a, b]]);
                                                      3
(%o1)         - 2 f2 + f1 (f1 + e1) - 3 e1 e2 + e1
(%i2) ratsimp (%);
                           2                          3
(%o2)         - 2 f2 + f1  + e1 f1 - 3 e1 e2 + e1
```

Other functions for changing bases: `comp2ele`.

multi_pui Function

is to the function `pui` what the function `multi_elem` is to the function `elem`.

```
(%i1) multi_pui ([[2, p1, p2], [2, t1, t2]], a*x + a^2 + x^3,
      [[x, y], [a, b]]);
                                                3
                                     3 p1 p2   p1
(%o1)              t2 + p1 t1 + ------- - ---
                                        2      2
```

pui (*L*, *sym*, *lvar*) Function

decomposes the symmetric polynomial *sym*, in the variables in the list *lvar*, in terms of the power functions in the list *L*. If the first element of *L* is given, it will be the size of the alphabet, otherwise the size will be the degree of the polynomial *sym*. If values are missing in the list *L*, formal values of the type *p1*, *p2* , etc. will be added. The polynomial *sym* may be given in three different forms: contracted (`elem` should then be 1, its default value), partitioned (`elem` should be 3), or extended (i.e. the entire polynomial, and `elem` should then be 2). The function `pui` is used in the same way.

```
(%i1) pui;
(%o1)                              1
(%i2) pui ([3, a, b], u*x*y*z, [x, y, z]);
                          2
                  a (a  - b) u   (a b - p3) u
(%o2)             ------------ - ------------
                       6              3
(%i3) ratsimp (%);
                                      3
                     (2 p3 - 3 a b + a ) u
(%o3)                ---------------------
                                6
```

Other functions for changing bases: `comp2ele`.

pui2comp (*n*, *lpui*) Function

renders the list of the first *n* complete functions (with the length first) in terms of the power functions given in the list *lpui*. If the list *lpui* is empty, the cardinal is *n*, otherwise it is its first element (as in `comp2ele` and `comp2pui`).

```
(%i1) pui2comp (2, []);
                                       2
                                  p2 + p1
(%o1)                  [2, p1, --------]
                                    2
(%i2) pui2comp (3, [2, a1]);
                                                 2
                                      a1 (p2 + a1 )
                           2  p3 + ------------- + a1 p2
                  p2 + a1                2
(%o2)     [2, a1, --------, -------------------------]
                     2                   3
(%i3) ratsimp (%);
                              2                     3
```

```
                          p2 + a1   2 p3 + 3 a1 p2 + a1
(%o3)          [2, a1, --------, --------------------]
                             2               6
```

Other functions for changing bases: `comp2ele`.

pui2ele (*n*, *lpui*) Function

effects the passage from power functions to the elementary symmetric functions. If the flag `pui2ele` is `girard`, it will return the list of elementary symmetric functions from 1 to *n*, and if the flag is `close`, it will return the *n*-th elementary symmetric function.

Other functions for changing bases: `comp2ele`.

puireduc (*n*, *lpui*) Function

lpui is a list whose first element is an integer *m*. `puireduc` gives the first *n* power functions in terms of the first *m*.

```
(%i1) puireduc (3, [2]);
                                          2
                                       p1 (p1  - p2)
(%o1)           [2, p1, p2, p1 p2 - -------------]
                                          2
(%i2) ratsimp (%);
                                            3
                                  3 p1 p2 - p1
(%o2)               [2, p1, p2, -------------]
                                        2
```

schur2comp (*P*, *l_var*) Function

P is a polynomial in the variables of the list *l_var*. Each of these variables represents a complete symmetric function. In *l_var* the *i*-th complete symmetric function is represented by the concatenation of the letter `h` and the integer *i*: `h`*i*. This function expresses *P* in terms of Schur functions.

```
(%i1) schur2comp (h1*h2 - h3, [h1, h2, h3]);
(%o1)                          s
                                1, 2
(%i2) schur2comp (a*h3, [h3]);
(%o2)                          s  a
                                3
```

32.2.2 Changing representations

cont2part (*pc*, *lvar*) Function

returns the partitioned polynomial associated to the contracted form *pc* whose variables are in *lvar*.

```
(%i1) pc: 2*a^3*b*x^4*y + x^5;
                              3    4      5
(%o1)                    2 a  b x  y + x
```

```
(%i2) cont2part (pc, [x, y]);
                                      3
(%o2)              [[1, 5, 0], [2 a  b, 4, 1]]
```

contract (*psym, lvar*) Function

returns a contracted form (i.e. a monomial orbit under the action of the symmetric group) of the polynomial *psym* in the variables contained in the list *lvar*. The function `explose` performs the inverse operation. The function `tcontract` tests the symmetry of the polynomial.

```
(%i1) psym: explose (2*a^3*b*x^4*y, [x, y, z]);
        3     4     3     4     3    4        3    4
(%o1) 2 a  b y z  + 2 a  b x z  + 2 a  b y  z + 2 a  b x  z

                                          3     4     3    4
                                       + 2 a  b x y  + 2 a  b x  y
(%i2) contract (psym, [x, y, z]);
                               3    4
(%o2)                     2 a  b x  y
```

explose (*pc, lvar*) Function

returns the symmetric polynomial associated with the contracted form *pc*. The list *lvar* contains the variables.

```
(%i1) explose (a*x + 1, [x, y, z]);
(%o1)                  a z + a y + a x + 1
```

part2cont (*ppart, lvar*) Function

goes from the partitioned form to the contracted form of a symmetric polynomial. The contracted form is rendered with the variables in *lvar*.

```
(%i1) part2cont ([[2*a^3*b, 4, 1]], [x, y]);
                               3    4
(%o1)                     2 a  b x  y
```

partpol (*psym, lvar*) Function

psym is a symmetric polynomial in the variables of the list *lvar*. This function retturns its partitioned representation.

```
(%i1) partpol (-a*(x + y) + 3*x*y, [x, y]);
(%o1)               [[3, 1, 1], [- a, 1, 0]]
```

tcontract (*pol, lvar*) Function

tests if the polynomial *pol* is symmetric in the variables of the list *lvar*. If so, it returns a contracted representation like the function `contract`.

tpartpol (*pol, lvar*) Function

tests if the polynomial *pol* is symmetric in the variables of the list *lvar*. If so, it returns its partitioned representation like the function `partpol`.

32.2.3 Groups and orbits

direct ([*p_1, ..., p_n*], *y*, *f*, [*lvar_1, ..., lvar_n*]) Function

calculates the direct image (see M. Giusti, D. Lazard et A. Valibouze, ISSAC 1988, Rome) associated to the function *f*, in the lists of variables *lvar_1*, ..., *lvar_n*, and in the polynomials *p_1*, ..., *p_n* in a variable *y*. The arity of the function *f* is important for the calulation. Thus, if the expression for *f* does not depend on some variable, it is useless to include this variable, and not including it will also considerably reduce the amount of computation.

```
(%i1) direct ([z^2  - e1* z + e2, z^2  - f1* z + f2],
              z, b*v + a*u, [[u, v], [a, b]]);
       2
(%o1) y  - e1 f1 y

                                  2           2             2   2
                    - 4 e2 f2 - (e1  - 2 e2) (f1  - 2 f2) + e1  f1
                  + ----------------------------------------------
                                        2
(%i2) ratsimp (%);
              2                2                    2
(%o2)        y  - e1 f1 y + (e1  - 4 e2) f2 + e2 f1
(%i3) ratsimp (direct ([z^3-e1*z^2+e2*z-e3,z^2  - f1* z + f2],
              z, b*v + a*u, [[u, v], [a, b]]));
       6           5         2                          2    2   4
(%o3) y  - 2 e1 f1 y  + ((2 e1  - 6 e2) f2 + (2 e2 + e1 ) f1 ) y

                         3                                  3   3
 + ((9 e3 + 5 e1 e2 - 2 e1 ) f1 f2 + (- 2 e3 - 2 e1 e2) f1 ) y

         2       2        4    2
 + ((9 e2  - 6 e1  e2 + e1 ) f2

                    2       2       2                   2    4
 + (- 9 e1 e3 - 6 e2  + 3 e1  e2) f1  f2 + (2 e1 e3 + e2 ) f1 )

  2          2                       2     3          2
 y  + (((9 e1  - 27 e2) e3 + 3 e1 e2  - e1  e2) f1 f2

                 2           2    3               5
 + ((15 e2 - 2 e1 ) e3 - e1 e2 ) f1  f2 - 2 e2 e3 f1 ) y

           2                   3           3     2   2    3
 + (- 27 e3  + (18 e1 e2 - 4 e1 ) e3 - 4 e2  + e1  e2 ) f2

         2     3                   3    2   2
 + (27 e3  + (e1  - 9 e1 e2) e3 + e2 ) f1  f2

                   2    4        2   6
```

```
 + (e1 e2 e3 - 9 e3 ) f1  f2 + e3  f1
```

Finding the polynomial whose roots are the sums $a+u$ where a is a root of $z^2-e_1z+e_2$ 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
(%o1) y  + (- 2 f1 - 2 e1) y  + (2 f2 + f1  + 3 e1 f1 + 2 e2

    2   2                                2              2
 + e1 ) y  + ((- 2 f1 - 2 e1) f2 - e1 f1  + (- 2 e2 - e1 ) f1

                  2                     2           2
 - 2 e1 e2) y + f2  + (e1 f1 - 2 e2 + e1 ) f2 + e2 f1  + e1 e2 f1

    2
 + e2
```

`direct` accepts two flags: `elementaires` and `puissances` (default) which allow decomposing the symmetric polynomials appearing in the calculation into elementary symmetric functions, or power functions, respectively.

Functions of `sym` used in this function:

`multi_orbit` (so `orbit`), `pui_direct`, `multi_elem` (so `elem`), `multi_pui` (so `pui`), `pui2ele`, `ele2pui` (if the flag `direct` is in `puissances`).

multi_orbit (*P*, [*lvar_1*, *lvar_2*,..., *lvar_p*]) Function

P is a polynomial in the set of variables contained in the lists *lvar_1*, *lvar_2*, ..., *lvar_p*. This function returns the orbit of the polynomial *P* under the action of the product of the symmetric groups of the sets of variables represented in these *p* lists.

```
(%i1) multi_orbit (a*x + b*y, [[x, y], [a, b]]);
(%o1)                [b y + a x, a y + b x]
(%i2) multi_orbit (x + y + 2*a, [[x, y], [a, b, c]]);
(%o2)        [y + x + 2 c, y + x + 2 b, y + x + 2 a]
```

Also see: `orbit` for the action of a single symmetric group.

multsym (*ppart_1*, *ppart_2*, *n*) Function

returns the product of the two symmetric polynomials in *n* variables by working only modulo the action of the symmetric group of order *n*. The polynomials are in their partitioned form.

Given the 2 symmetric polynomials in *x*, *y*: `3*(x + y) + 2*x*y` and `5*(x^2 + y^2)` whose partitioned forms are `[[3, 1], [2, 1, 1]]` and `[[5, 2]]`, their product will be

```
(%i1) multsym ([[3, 1], [2, 1, 1]], [[5, 2]], 2);
(%o1)         [[10, 3, 1], [15, 3, 0], [15, 2, 1]]
```

that is `10*(x^3*y + y^3*x) + 15*(x^2*y + y^2*x) + 15*(x^3 + y^3)`.

Functions for changing the representations of a symmetric polynomial:

`contract`, `cont2part`, `explose`, `part2cont`, `partpol`, `tcontract`, `tpartpol`.

orbit (*P*, *lvar*) Function

computes the orbit of the polynomial P in the variables in the list *lvar* under the action of the symmetric group of the set of variables in the list *lvar*.

```
(%i1) orbit (a*x + b*y, [x, y]);
(%o1)                [a y + b x, b y + a x]
(%i2) orbit (2*x + x^2, [x, y]);
                         2         2
(%o2)                 [y  + 2 y, x  + 2 x]
```

See also `multi_orbit` for the action of a product of symmetric groups on a polynomial.

pui_direct (*orbite*, [*lvar_1*, ..., *lvar_n*], [*d_1*, *d_2*, ..., *d_n*]) Function

Let f be a polynomial in n blocks of variables *lvar_1*, ..., *lvar_n*. Let *c_i* be the number of variables in *lvar_i*, and SC be the product of n symmetric groups of degree *c_1*, ..., *c_n*. This group acts naturally on f. The list *orbite* is the orbit, denoted $SC(f)$, of the function f under the action of SC. (This list may be obtained by the function `multi_orbit`.) The *di* are integers s.t. $c_1 \leq d_1, c_2 \leq d_2, \ldots, c_n \leq d_n$.

Let SD be the product of the symmetric groups $S_{d_1} \times S_{d_2} \times \cdots \times S_{d_n}$. The function `pui_direct` returns the first n power functions of $SD(f)$ deduced from the power functions of $SC(f)$, where n is the size of $SD(f)$.

The result is in multi-contracted form w.r.t. SD, i.e. only one element is kept per orbit, under the action of SD.

```
(%i1) l: [[x, y], [a, b]];
(%o1)                    [[x, y], [a, b]]
(%i2) pui_direct (multi_orbit (a*x + b*y, l), l, [2, 2]);
                                      2  2
(%o2)               [a x, 4 a b x y + a  x ]
(%i3) pui_direct (multi_orbit (a*x + b*y, l), l, [3, 2]);
                             2  2    2   2       3  3
(%o3) [2 a x, 4 a b x y + 2 a  x , 3 a  b x  y + 2 a  x ,

   2  2  2  2     3   3       4  4
12 a  b  x  y  + 4 a  b x  y + 2 a  x ,

   3  2  3  2     4   4       5  5
10 a  b  x  y  + 5 a  b x  y + 2 a  x ,

   3  3  3  3      4  2  4  2     5   5       6  6
40 a  b  x  y  + 15 a  b  x  y  + 6 a  b x  y + 2 a  x ]
(%i4) pui_direct ([y + x + 2*c, y + x + 2*b, y + x + 2*a],
      [[x, y], [a, b, c]], [2, 3]);
                               2            2
(%o4) [3 x + 2 a, 6 x y + 3 x  + 4 a x + 4 a ,

                  2                  3       2       2       3
              9 x  y + 12 a x y + 3 x  + 6 a x  + 12 a  x + 8 a ]
```

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
```

lgtreillis (*n*, *m*) Function

returns the list of partitions of weight *n* and length *m*.

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

Also see: `ltreillis`, `treillis` and `treinat`.

ltreillis (*n*, *m*) Function

returns the list of partitions of weight *n* and length less than or equal to *m*.

```
(%i1) ltreillis (4, 2);
(%o1)               [[4, 0], [3, 1], [2, 2]]
```

Also see: `lgtreillis`, `treillis` and `treinat`.

treillis (*n*) Function

returns all partitions of weight *n*.

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

See also: `lgtreillis`, `ltreillis` and `treinat`.

treinat (*part*) Function

retruns the list of partitions inferior to the partition *part* w.r.t. the natural order.

```
(%i1) treinat ([5]);
(%o1)                          [[5]]
(%i2) treinat ([1, 1, 1, 1, 1]);
(%o2) [[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1],

                                                   [1, 1, 1, 1, 1]]
(%i3) treinat ([3, 2]);
(%o3)                [[5], [4, 1], [3, 2]]
```

See also: `lgtreillis`, `ltreillis` and `treillis`.

32.2.5 Polynomials and their roots

ele2polynome (*L*, *z*) Function

returns the polynomial in *z* s.t. the elementary symmetric functions of its roots are in the list *L* = [*n*, *e_1*, ..., *e_n*], where *n* is the degree of the polynomial and *e_i* the *i*-th elementary symmetric function.

```
(%i1) ele2polynome ([2, e1, e2], z);
                           2
(%o1)                     z  - e1 z + e2
(%i2) polynome2ele (x^7 - 14*x^5 + 56*x^3  - 56*x + 22, x);
(%o2)          [7, 0, - 14, 0, 56, 0, - 56, - 22]
(%i3) ele2polynome ([7, 0, -14, 0, 56, 0, -56, -22], x);
                    7       5       3
(%o3)             x  - 14 x  + 56 x  - 56 x + 22
```

The inverse: `polynome2ele` (*P*, *z*).

Also see: `polynome2ele`, `pui2polynome`.

polynome2ele (*P*, *x*) Function

gives the list *l* = [*n*, *e_1*, ..., *e_n*] where *n* is the degree of the polynomial *P* in the variable *x* and *e_i* is the *i*-the elementary symmetric function of the roots of *P*.

```
(%i1) polynome2ele (x^7 - 14*x^5 + 56*x^3 - 56*x + 22, x);
(%o1)          [7, 0, - 14, 0, 56, 0, - 56, - 22]
(%i2) ele2polynome ([7, 0, -14, 0, 56, 0, -56, -22], x);
                    7       5       3
(%o2)             x  - 14 x  + 56 x  - 56 x + 22
```

The inverse: `ele2polynome` (*l*, *x*)

prodrac (*L*, *k*) Function

L is a list containing the elementary symmetric functions on a set *A*. `prodrac` returns the polynomial whose roots are the *k* by *k* products of the elements of *A*.

Also see `somrac`.

pui2polynome (*x*, *lpui*) Function

calculates the polynomial in *x* whose power functions of the roots are given in the list *lpui*.

```
(%i1) pui;
(%o1)                          1
(%i2) kill(labels);
(%o0)                        done
(%i1) polynome2ele (x^3 - 4*x^2 + 5*x - 1, x);
(%o1)                     [3, 4, 5, 1]
(%i2) ele2pui (3, %);
(%o2)                     [3, 4, 6, 7]
(%i3) pui2polynome (x, %);
                        3      2
(%o3)                  x  - 4 x  + 5 x - 1
```

See also: `polynome2ele`, `ele2polynome`.

somrac (*L*, *k*) Function

The list *L* contains elementary symmetric functions of a polynomial *P* . The function computes the polynomial whose roots are the *k* by *k* distinct sums of the roots of *P*.

Also see `prodrac`.

32.2.6 Resolvents

resolvante (*P*, *x*, *f*, [*x_1*,..., *x_d*]) Function

calculates the resolvent of the polynomial P in x of degree n >= d by the function f expressed in the variables *x_1*, ..., *x_d*. For efficiency of computation it is important to not include in the list `[x_1, ..., x_d]` variables which do not appear in the transformation function *f*.

To increase the efficiency of the computation one may set flags in `resolvante` so as to use appropriate algorithms:

If the function f is unitary:

- A polynomial in a single variable,
- linear,
- alternating,
- a sum,
- symmetric,
- a product,
- the function of the Cayley resolvent (usable up to degree 5)

```
(x1*x2 + x2*x3 + x3*x4 + x4*x5 + x5*x1 -
     (x1*x3 + x3*x5 + x5*x2 + x2*x4 + x4*x1))^2
```

  general,

the flag of `resolvante` may be, respectively:

- unitaire,
- lineaire,
- alternee,
- somme,
- produit,
- cayley,
- generale.

```
(%i1) resolvante: unitaire$
(%i2) resolvante (x^7 - 14*x^5 + 56*x^3 - 56*x + 22, x, x^3 - 1,
      [x]);

" resolvante unitaire " [7, 0, 28, 0, 168, 0, 1120, - 154, 7840,
                         - 2772, 56448, - 33880,

413952, - 352352, 3076668, - 3363360, 23114112, - 30494464,

175230832, - 267412992, 1338886528, - 2292126760]
  3       6      3       9      6      3
[x  - 1, x  - 2 x  + 1, x  - 3 x  + 3 x  - 1,

 12      9      6      3       15      12       9       6      3
```

```
x   - 4 x  + 6 x  - 4 x  + 1, x   - 5 x   + 10 x  - 10 x  + 5 x

       18      15       12       9       6      3
 - 1, x   - 6 x   + 15 x   - 20 x  + 15 x  - 6 x  + 1,

 21      18       15       12       9       6      3
x   - 7 x   + 21 x   - 35 x   + 35 x  - 21 x  + 7 x  - 1]
[- 7, 1127, - 6139, 431767, - 5472047, 201692519, - 3603982011]
       7      6       5        4         3          2
(%o2) y  + 7 y  - 539 y  - 1841 y  + 51443 y  + 315133 y

                                            + 376999 y + 125253
(%i3) resolvante: lineaire$
(%i4) resolvante (x^4 - 1, x, x1 + 2*x2 + 3*x3, [x1, x2, x3]);

" resolvante lineaire "
       24       20         16            12             8
(%o4) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                  4
                                          + 344489984 y  + 655360000
(%i5) resolvante: general$
(%i6) resolvante (x^4 - 1, x, x1 + 2*x2 + 3*x3, [x1, x2, x3]);

" resolvante generale "
       24       20         16            12             8
(%o6) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                  4
                                          + 344489984 y  + 655360000
(%i7) resolvante (x^4 - 1, x, x1 + 2*x2 + 3*x3, [x1, x2, x3, x4]);

" resolvante generale "
       24       20         16            12             8
(%o7) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                  4
                                          + 344489984 y  + 655360000
(%i8) direct ([x^4 - 1], x, x1 + 2*x2 + 3*x3, [[x1, x2, x3]]);
       24       20         16            12             8
(%o8) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                  4
                                          + 344489984 y  + 655360000
(%i9) resolvante :lineaire$
(%i10) resolvante (x^4 - 1, x, x1 + x2 + x3, [x1, x2, x3]);

" resolvante lineaire "
                            4
```

```
(%o10)                        y  - 1
(%i11) resolvante: symetrique$
(%i12) resolvante (x^4 - 1, x, x1 + x2 + x3, [x1, x2, x3]);

" resolvante symetrique "
                               4
(%o12)                        y  - 1
(%i13) resolvante (x^4 + x + 1, x, x1 - x2, [x1, x2]);

" resolvante symetrique "
                            6      2
(%o13)                     y  - 4 y  - 1
(%i14) resolvante: alternee$
(%i15) resolvante (x^4 + x + 1, x, x1 - x2, [x1, x2]);

" resolvante alternee "
           12      8       6        4        2
(%o15)     y   + 8 y  + 26 y  - 112 y  + 216 y  + 229
(%i16) resolvante: produit$
(%i17) resolvante (x^7 - 7*x + 3, x, x1*x2*x3, [x1, x2, x3]);

" resolvante produit "
       35      33         29        28         27        26
(%o17) y   - 7 y   - 1029 y   + 135 y   + 7203 y   - 756 y

         24           23          22            21           20
 + 1323 y   + 352947 y   - 46305 y   - 2463339 y   + 324135 y

          19           18             17              15
 - 30618 y   - 453789 y   - 40246444 y   + 282225202 y

             14              12             11            10
 - 44274492 y   + 155098503 y   + 12252303 y   + 2893401 y

              9            8            7             6
 - 171532242 y  + 6751269 y  + 2657205 y  - 94517766 y

            5             3
 - 3720087 y  + 26040609 y  + 14348907
(%i18) resolvante: symetrique$
(%i19) resolvante (x^7 - 7*x + 3, x, x1*x2*x3, [x1, x2, x3]);

" resolvante symetrique "
       35      33         29        28         27        26
(%o19) y   - 7 y   - 1029 y   + 135 y   + 7203 y   - 756 y

         24           23          22            21           20
 + 1323 y   + 352947 y   - 46305 y   - 2463339 y   + 324135 y
```

```
          19           18             17              15
 - 30618 y   - 453789 y   - 40246444 y   + 282225202 y

             14              12             11            10
 - 44274492 y   + 155098503 y   + 12252303 y   + 2893401 y

              9           8           7             6
 - 171532242 y  + 6751269 y  + 2657205 y  - 94517766 y

            5            3
 - 3720087 y  + 26040609 y  + 14348907
(%i20) resolvante: cayley$
(%i21) resolvante (x^5 - 4*x^2 + x + 1, x, a, []);

" resolvante de Cayley "
        6       5         4          3           2
(%o21) x  - 40 x  + 4080 x  - 92928 x  + 3772160 x  + 37880832 x

                                                          + 93392896
```

For the Cayley resolvent, the 2 last arguments are neutral and the input polynomial must necessarily be of degree 5.

See also:

`resolvante_bipartite`, `resolvante_produit_sym`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`.

resolvante_alternee1 (P, x) Function

calculates the transformation $P(x)$ of degree n by the function $\prod_{1\leq i<j\leq n-1}(x_i - x_j)$.

See also:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante` , `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`, `resolvante_bipartite`.

resolvante_bipartite (P, x) Function

calculates the transformation of $P(x)$ of even degree n by the function $x_1x_2\cdots x_{n/2} + x_{n/2+1}\cdots x_n$.

See also:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante` , `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`, `resolvante_alternee1`.

```
(%i1) resolvante_bipartite (x^6 + 108, x);
              10        8          6             4
(%o1)        y   - 972 y  + 314928 y  - 34012224 y
```

See also:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante`, `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`, `resolvante_alternee1`.

resolvante_diedrale (*P*, *x*) Function

calculates the transformation of *P*(*x*) by the function *x_1 x_2* + *x_3 x_4*.

```
(%i1) resolvante_diedrale (x^5 - 3*x^4 + 1, x);
       15       12       11       10        9         8         7
(%o1) x   - 21 x   - 81 x   - 21 x   + 207 x  + 1134 x  + 2331 x

        6         5          4          3          2
 - 945 x  - 4970 x  - 18333 x  - 29079 x  - 20745 x  - 25326 x

 - 697
```

See also:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante`.

resolvante_klein (*P*, *x*) Function

calculates the transformation of *P*(*x*) by the function *x_1 x_2 x_4* + *x_4*.

See also:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`.

resolvante_klein3 (*P*, *x*) Function

calculates the transformation of *P*(*x*) by the function *x_1 x_2 x_4* + *x_4*.

See also:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante_klein`, `resolvante`, `resolvante_vierer`, `resolvante_diedrale`.

resolvante_produit_sym (*P*, *x*) Function

calculates the list of all product resolvents of the polynomial *P*(*x*).

```
(%i1) resolvante_produit_sym (x^5 + 3*x^4 + 2*x - 1, x);
       5      4             10      8       7       6       5
(%o1) [y  + 3 y  + 2 y - 1, y   - 2 y  - 21 y  - 31 y  - 14 y

    4       3      2       10      8       7    6       5       4
 - y  + 14 y  + 3 y  + 1, y   + 3 y  + 14 y  - y  - 14 y  - 31 y

       3      2       5      4
 - 21 y  - 2 y  + 1, y  - 2 y  - 3 y - 1, y - 1]
(%i2) resolvante: produit$
(%i3) resolvante (x^5 + 3*x^4 + 2*x - 1, x, a*b*c, [a, b, c]);

" resolvante produit "
       10      8       7    6        5       4       3     2
(%o3) y   + 3 y  + 14 y  - y  - 14 y  - 31 y  - 21 y  - 2 y  + 1
```

See also:

`resolvante`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`.

resolvante_unitaire (*P*, *Q*, *x*) Function

computes the resolvent of the polynomial `P(x)` by the polynomial `Q(x)`.

See also:

`resolvante_produit_sym`, `resolvante`, `resolvante_alternee1`, `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`.

resolvante_vierer (*P*, *x*) Function

computes the transformation of `P(x)` by the function *x_1 x_2* - *x_3 x_4*.

See also:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante_klein`, `resolvante_klein3`, `resolvante`, `resolvante_diedrale`.

32.2.7 Miscellaneous

multinomial (*r*, *part*) Function

where *r* is the weight of the partition *part*. This function returns the associate multinomial coefficient: if the parts of *part* are *i_1*, *i_2*, ..., *i_k*, the result is `r!/(`*i_1*`!` *i_2*`!` ... *i_k*`!)`.

permut (*L*) Function

returns the list of permutations of the list *L*.

33 Groups

33.1 Functions and Variables for Groups

todd_coxeter (*relations*, *subgroup*) Function
todd_coxeter (*relations*) Function

Find the order of G/H where G is the Free Group modulo *relations*, and H is the subgroup of G generated by *subgroup*. *subgroup* is an optional argument, defaulting to []. In doing this it produces a multiplication table for the right action of G on G/H, where the cosets are enumerated [H,Hg2,Hg3,...]. This can be seen internally in the variable `todd_coxeter_state`.

Example:

```
(%i1) symet(n):=create_list(
        if (j - i) = 1 then (p(i,j))^^3 else
            if (not i = j) then (p(i,j))^^2 else
                p(i,i) , j, 1, n-1, i, 1, j);
                                                          <3>
(%o1) symet(n) := create_list(if j - i = 1 then p(i, j)

                                    <2>
 else (if not i = j then p(i, j)    else p(i, i)), j, 1, n - 1,

i, 1, j)
(%i2) p(i,j) := concat(x,i).concat(x,j);
(%o2)         p(i, j) := concat(x, i) . concat(x, j)
(%i3) symet(5);
         <2>           <3>    <2>           <2>           <3>
(%o3) [x1   , (x1 . x2)   , x2   , (x1 . x3)   , (x2 . x3)   ,

            <2>           <2>           <2>           <3>    <2>
          x3   , (x1 . x4)   , (x2 . x4)   , (x3 . x4)   , x4   ]
(%i4) todd_coxeter(%o3);

Rows tried 426
(%o4)                          120
(%i5) todd_coxeter(%o3,[x1]);

Rows tried 213
(%o5)                          60
(%i6) todd_coxeter(%o3,[x1,x2]);

Rows tried 71
(%o6)                          20
```

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:

```
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 (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 Declaration

Maxima understands two distinct types of features, system features and features which apply to mathematical expressions. See also `status` for information about system features. See also `features` and `featurep` for information about mathematical features.

`feature` itself is not the name of a function or variable.

featurep (*a*, *f*) Function

Attempts to determine whether the object *a* has the feature *f* on the basis of the facts in the current database. If so, it returns `true`, else `false`.

Note that `featurep` returns `false` when neither *f* nor the negation of *f* can be established.

`featurep` evaluates its argument.

See also `declare` and `features`.

```
(%i1) declare (j, even)$
(%i2) featurep (j, integer);
(%o2)                          true
```

maxima_tempdir System variable

`maxima_tempdir` names the directory in which Maxima creates some temporary files. In particular, temporary files for plotting are created in `maxima_tempdir`.

The initial value of `maxima_tempdir` is the user's home directory, if Maxima can locate it; otherwise Maxima makes a guess about a suitable directory.

`maxima_tempdir` may be assigned a string which names a directory.

maxima_userdir System variable

`maxima_userdir` names a directory which Maxima searches to find Maxima and Lisp files. (Maxima searches some other directories as well; `file_search_maxima` and `file_search_lisp` are the complete lists.)

The initial value of `maxima_userdir` is a subdirectory of the user's home directory, if Maxima can locate it; otherwise Maxima makes a guess about a suitable directory.

`maxima_userdir` may be assigned a string which names a directory. However, assigning to `maxima_userdir` does not automatically change `file_search_maxima` and `file_search_lisp`; those variables must be changed separately.

room () Function
room (*true*) Function
room (*false*) Function

Prints out a description of the state of storage and stack management in Maxima. `room` calls the Lisp function of the same name.

- `room ()` prints out a moderate description.
- `room (true)` prints out a verbose description.
- `room (false)` prints out a terse description.

status (*feature*) Function
status (*feature*, *putative_feature*) Function
status (*status*) 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,` *putative_feature*`)` returns `true` if *putative_feature* is on the list of items returned by `status (feature)` and `false` otherwise. `status` quotes the argument *putative_feature*. 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")`.
- `status (status)` returns a two-element list `[feature, status]`. `feature` and `status` are the two arguments accepted by the `status` function; it is unclear if this list has additional significance.

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 `showtime: true` to make Maxima print out the computation time and elapsed time with each output line.

timedate () Function

Returns a string representing the current time and date. The string has the format `HH:MM:SS Day, mm/dd/yyyy (GMT-n)`, where the fields are hours, minutes, seconds, day of week, month, day of month, year, and hours different from GMT.

Example:

```
(%i1) d: timedate ();
(%o1) 08:05:09 Wed, 11/02/2005 (GMT-7)
(%i2) print ("timedate reports current time", d)$
timedate reports current time 08:05:09 Wed, 11/02/2005 (GMT-7)
```

absolute_real_time () Function

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 () Function

Returns the number of seconds (including fractions of a second) since Maxima was most recently started or restarted. The return value is a floating-point number.

See also `absolute_real_time` and `elapsed_run_time`.

Example:

```
(%i1) elapsed_real_time ();
(%o1)                       2.559324
(%i2) expand ((a + b)^500)$
(%i3) elapsed_real_time ();
(%o3)                       7.552087
```

elapsed_run_time () Function

Returns an estimate of the number of seconds (including fractions of a second) which Maxima has spent in computations since Maxima was most recently started or restarted. The return value is a floating-point number.

See also `absolute_real_time` and `elapsed_real_time`.

Example:

```
(%i1) elapsed_run_time ();
(%o1)                         0.04
(%i2) expand ((a + b)^500)$
(%i3) elapsed_run_time ();
(%o3)                         1.26
```

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 System variable

Default value: `[]`

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

alphabetic Declaration

`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 61.

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~]
```

apropos (*string*) Function

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

args (*expr*) Function

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

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

`args (`*expr*`)` is equivalent to `substpart ("[",` *expr*`, 0)`. See also `substpart`, 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.

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`

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

```
                  domain:real

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

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

numberp (*expr*) Function

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

`numberp` returns `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:

```
(%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]
```

properties (*a*) Function

Returns a list of the names of all the properties associated with the atom *a*.

props Special symbol

`props` are atoms which have any property other than those explicitly mentioned in `infolists`, such as specified by `atvalue`, `matchdeclare`, etc., as well as properties specified in the `declare` function.

propvars (*prop*) Function

Returns a list of those atoms on the `props` list which have the property indicated by *prop*. Thus `propvars (atvalue)` returns a list of atoms which have atvalues.

put (*atom, value, indicator*) Function

Assigns *value* to the property (specified by *indicator*) of *atom*. *indicator* may be the name of any property, not just a system-defined property.

`put` evaluates its arguments. `put` returns *value*.

Examples:

```
(%i1) put (foo, (a+b)^5, expr);
                                        5
(%o1)                          (b + a)
(%i2) put (foo, "Hello", str);
(%o2)                          Hello
(%i3) properties (foo);
```

```
(%o3)            [[user properties, str, expr]]
(%i4) get (foo, expr);
                                      5
(%o4)                         (b + a)
(%i5) get (foo, str);
(%o5)                           Hello
```

qput (*atom, value, indicator*) Function

Assigns *value* to the property (specified by *indicator*) of *atom*. This is the same as `put`, except that the arguments are quoted.

Example:

```
(%i1) foo: aa$
(%i2) bar: bb$
(%i3) baz: cc$
(%i4) put (foo, bar, baz);
(%o4)                           bb
(%i5) properties (aa);
(%o5)                [[user properties, cc]]
(%i6) get (aa, cc);
(%o6)                           bb
(%i7) qput (foo, bar, baz);
(%o7)                           bar
(%i8) properties (foo);
(%o8)            [value, [user properties, baz]]
(%i9) get ('foo, 'baz);
(%o9)                           bar
```

rem (*atom, indicator*) Function

Removes the property indicated by *indicator* from *atom*.

remove (*a_1, p_1, ..., a_n, p_n*) Function
remove ([*a_1, ..., a_m*], [*p_1, ..., p_n*], ...) Function
remove ("*a*", *operator*) Function
remove (*a, transfun*) Function
remove (*all, p*) Function

Removes properties associated with atoms.

`remove (a_1, p_1, ..., a_n, p_n)` removes property `p_k` from atom `a_k`.

`remove ([a_1, ..., a_m], [p_1, ..., p_n], ...)` removes properties *p_1*, ..., *p_n* from atoms *a_1*, ..., *a_m*. There may be more than one pair of lists.

`remove (all, p)` removes the property *p* from all atoms which have it.

The removed properties may be system-defined properties such as `function`, `macro`, or `mode_declare`, or user-defined properties.

A property may be `transfun` to remove the translated Lisp version of a function. After executing this, the Maxima version of the function is executed rather than the translated version.

`remove ("a", operator)` or, equivalently, `remove ("a", op)` removes from *a* the operator properties declared by `prefix`, `infix`, `nary`, `postfix`, `matchfix`, or `nofix`. Note that the name of the operator must be written as a quoted string.

`remove` always returns `done` whether or not an atom has a specified property. This behavior is unlike the more specific remove functions `remvalue`, `remarray`, `remfunction`, and `remrule`.

remvalue (*name_1, ..., name_n*) Function
remvalue (*all*) Function

Removes the values of user variables *name_1*, ..., *name_n* (which can be subscripted) from the system.

`remvalue (all)` removes the values of all variables in `values`, the list of all variables given names by the user (as opposed to those which are automatically assigned by Maxima).

See also `values`.

rncombine (*expr*) Function

Transforms *expr* by combining all terms of *expr* that have identical denominators or denominators that differ from each other by numerical factors only. This is slightly different from the behavior of `combine`, which collects terms that have identical denominators.

Setting `pfeformat: true` and using `combine` yields results similar to those that can be obtained with `rncombine`, but `rncombine` takes the additional step of cross-multiplying numerical denominator factors. This results in neater forms, and the possibility of recognizing some cancellations.

`load(rncomb)` loads this function.

scalarp (*expr*) Function

Returns `true` if *expr* is a number, constant, or variable declared `scalar` with `declare`, or composed entirely of numbers, constants, and such variables, but not containing matrices or lists.

setup_autoload (*filename, function_1, ..., function_n*) Function

Specifies that if any of *function_1*, ..., *function_n* are referenced and not yet defined, *filename* is loaded via `load`. *filename* usually contains definitions for the functions specified, although that is not enforced.

`setup_autoload` does not work for array functions.

`setup_autoload` quotes its arguments.

Example:

```
(%i1) legendre_p (1, %pi);
(%o1)                   legendre_p(1, %pi)
(%i2) setup_autoload ("specfun.mac", legendre_p, ultraspherical);
(%o2)                          done
(%i3) ultraspherical (2, 1/2, %pi);
Warning - you are redefining the Macsyma function ultraspherical
```

```
Warning - you are redefining the Macsyma function legendre_p
                                2
                  3 (%pi - 1)
(%o3)             ------------ + 3 (%pi - 1) + 1
                       2
(%i4) legendre_p (1, %pi);
(%o4)                           %pi
(%i5) legendre_q (1, %pi);
                                %pi + 1
                       %pi log(-------)
                                1 - %pi
(%o5)                  ---------------- - 1
                              2
```

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 (*progname, pattern, x_1, ..., x_n*) Function
defmatch (*progname, pattern*) Function

Defines a function *progname*(*expr*, *x_1*, ..., *x_n*) which tests *expr* to see if it matches *pattern*.

pattern is an expression containing the pattern arguments *x_1*, ..., *x_n* (if any) and some pattern variables (if any). The pattern arguments are given explicitly as arguments to `defmatch` while the pattern variables are declared by the `matchdeclare` function. Any variable not declared as a pattern variable in `matchdeclare` or as a pattern argument in `defmatch` matches only itself.

The first argument to the created function *progname* is an expression to be matched against the pattern and the other arguments are the actual arguments which correspond to the dummy variables *x_1*, ..., *x_n* in the pattern.

If the match is successful, *progname* returns a list of equations whose left sides are the pattern arguments and pattern variables, and whose right sides are the subexpressions which the pattern arguments and variables matched. The pattern variables, but not the pattern arguments, are assigned the subexpressions they match. If the match fails, *progname* returns `false`.

A literal pattern (that is, a pattern which contains neither pattern arguments nor pattern variables) returns `true` if the match succeeds.

See also `matchdeclare`, `defrule`, `tellsimp`, and `tellsimpafter`.

Examples:

Define a function `linearp(expr, x)` which tests `expr` to see if it is of the form `a*x + b` such that `a` and `b` do not contain `x` and `a` is nonzero. This match function matches expressions which are linear in any variable, because the pattern argument `x` is given to `defmatch`.

```
(%i1) matchdeclare (a, lambda ([e], e#0 and freeof(x, e)), b,
                    freeof(x));
(%o1)                          done
(%i2) defmatch (linearp, a*x + b, x);
(%o2)                         linearp
(%i3) linearp (3*z + (y + 1)*z + y^2, z);
                          2
(%o3)              [b = y , a = y + 4, x = z]
(%i4) a;
(%o4)                          y + 4
(%i5) b;
                                 2
(%o5)                           y
(%i6) x;
(%o6)                            x
```

Define a function `linearp(expr)` which tests `expr` to see if it is of the form `a*x + b` such that `a` and `b` do not contain `x` and `a` is nonzero. This match function only matches expressions linear in `x`, not any other variable, because no pattern argument is given to `defmatch`.

```
(%i1) matchdeclare (a, lambda ([e], e#0 and freeof(x, e)), b,
                    freeof(x));
(%o1)                          done
(%i2) defmatch (linearp, a*x + b);
(%o2)                         linearp
(%i3) linearp (3*z + (y + 1)*z + y^2);
(%o3)                          false
(%i4) linearp (3*x + (y + 1)*x + y^2);
                               2
(%o4)                  [b = y , a = y + 4]
```

Define a function `checklimits(expr)` which tests `expr` to see if it is a definite integral.

```
(%i1) matchdeclare ([a, f], true);
(%o1)                          done
(%i2) constinterval (l, h) := constantp (h - l);
(%o2)        constinterval(l, h) := constantp(h - l)
(%i3) matchdeclare (b, constinterval (a));
(%o3)                          done
(%i4) matchdeclare (x, atom);
(%o4)                          done
(%i5) simp : false;
(%o5)                          false
(%i6) defmatch (checklimits, 'integrate (f, x, a, b));
(%o6)                       checklimits
(%i7) simp : true;
(%o7)                          true
(%i8) 'integrate (sin(t), t, %pi + x, 2*%pi + x);
                        x + 2 %pi
                       /
                       [
```

```
(%o8)                  I          sin(t) dt
                       ]
                       /
                        x + %pi
(%i9) checklimits (%);
(%o9)    [b = x + 2 %pi, a = x + %pi, x = t, f = sin(t)]
```

defrule (*rulename, pattern, replacement*) Function

Defines and names a replacement rule for the given pattern. If the rule named *rulename* is applied to an expression (by `apply1`, `applyb1`, or `apply2`), every subexpression matching the pattern will be replaced by the replacement. All variables in the replacement which have been assigned values by the pattern match are assigned those values in the replacement which is then simplified.

The rules themselves can be treated as functions which transform an expression by one operation of the pattern match and replacement. If the match fails, the rule function returns `false`.

disprule (*rulename_1, ..., rulename_2*) Function
disprule (*all*) Function

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) -> []]
```

let (*prod, repl, predname, arg_1, ..., arg_n*) Function
let ([*prod, repl, predname, arg_1, ..., arg_n*], *package_name*) Function

Defines a substitution rule for `letsimp` such that *prod* is replaced by *repl*. *prod* is a product of positive or negative powers of the following terms:

- Atoms which `letsimp` will search for literally unless previous to calling `letsimp` the `matchdeclare` function is used to associate a predicate with the atom. In this case `letsimp` will match the atom to any term of a product satisfying the predicate.
- Kernels such as `sin(x)`, `n!`, `f(x,y)`, etc. As with atoms above `letsimp` will look for a literal match unless `matchdeclare` is used to associate a predicate with the argument of the kernel.

A term to a positive power will only match a term having at least that power. A term to a negative power on the other hand will only match a term with a power at least as negative. In the case of negative powers in *prod* the switch `letrat` must be set to `true`. See also `letrat`.

If a predicate is included in the `let` function followed by a list of arguments, a tentative match (i.e. one that would be accepted if the predicate were omitted) is accepted only if `predname (arg_1', ..., arg_n')` evaluates to `true` where *arg_i'* is the value matched to *arg_i*. The *arg_i* may be the name of any atom or the argument of any kernel appearing in *prod*. *repl* may be any rational expression. If any of the atoms or arguments from *prod* appear in *repl* the appropriate substitutions are made.

The global flag `letrat` controls the simplification of quotients by `letsimp`. When `letrat` is `false`, `letsimp` simplifies the numerator and denominator of *expr* separately, and does not simplify the quotient. Substitutions such as `n!/n` goes to `(n-1)!` then fail. When `letrat` is `true`, then the numerator, denominator, and the quotient are simplified in that order.

These substitution functions allow you to work with several rule packages at once. Each rule package can contain any number of `let` rules and is referenced by a user-defined name. `let ([`*prod*`,` *repl*`,` *predname*`,` *arg_1*`, ..., `*arg_n*`],` *package_name*`)` adds the rule *predname* to the rule package *package_name*. `letsimp (`*expr*`,` *package_name*`)` applies the rules in *package_name*. `letsimp (`*expr*`,` *package_name1*`,` *package_name2*`, ...)` is equivalent to `letsimp (`*expr*`,` *package_name1*`)` followed by `letsimp (%,` *package_name2*`)`, ...

`current_let_rule_package` is the name of the rule package that is presently being used. This variable may be assigned the name of any rule package defined via the `let` command. Whenever any of the functions comprising the `let` package are called with no package name, the package named by `current_let_rule_package` is used. If a call such as `letsimp (`*expr*`,` *rule_pkg_name*`)` is made, the rule package *rule_pkg_name* is used for that `letsimp` command only, and `current_let_rule_package` is not changed. If not otherwise specified, `current_let_rule_package` defaults to `default_let_rule_package`.

```
(%i1) matchdeclare ([a, a1, a2], true)$
(%i2) oneless (x, y) := is (x = y-1)$
(%i3) let (a1*a2!, a1!, oneless, a2, a1);
(%o3)         a1 a2! --> a1! where oneless(a2, a1)
```

```
(%i4) letrat: true$
(%i5) let (a1!/a1, (a1-1)!);
                         a1!
(%o5)                    --- --> (a1 - 1)!
                         a1
(%i6) letsimp (n*m!*(n-1)!/m);
(%o6)                       (m - 1)! n!
(%i7) let (sin(a)^2, 1 - cos(a)^2);
                         2               2
(%o7)                 sin (a) --> 1 - cos (a)
(%i8) letsimp (sin(x)^4);
                         4           2
(%o8)                 cos (x) - 2 cos (x) + 1
```

letrat Option variable

Default value: `false`

When `letrat` is `false`, `letsimp` simplifies the numerator and denominator of a ratio separately, and does not simplify the quotient.

When `letrat` is `true`, the numerator, denominator, and their quotient are simplified in that order.

```
(%i1) matchdeclare (n, true)$
(%i2) let (n!/n, (n-1)!);
                          n!
(%o2)                     -- --> (n - 1)!
                          n
(%i3) letrat: false$
(%i4) letsimp (a!/a);
                                a!
(%o4)                           --
                                a
(%i5) letrat: true$
(%i6) letsimp (a!/a);
(%o6)                         (a - 1)!
```

letrules () Function
letrules (*package_name*) 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 expression, except for match variables which are operands of addition `+` or multiplication `*`. Only addition and multiplication are handled specially; other n-ary operators (both built-in and user-defined) are treated like ordinary functions.

In the case of addition and multiplication, the match variable may be assigned a single expression which satisfies the match predicate, or a sum or product (respectively) of such expressions. Such multiple-term matching is greedy: predicates are evaluated in the order in which their associated variables appear in the match pattern, and a term which satisfies more than one predicate is taken by the first predicate which it satisfies. Each predicate is tested against all operands of the sum or product before the next predicate is evaluated. In addition, if 0 or 1 (respectively) satisfies a match predicate, and there are no other terms which satisfy the predicate, 0 or 1 is assigned to the match variable associated with the predicate.

The algorithm for processing addition and multiplication patterns makes some match results (for example, a pattern in which a "match anything" variable appears) dependent on the ordering of terms in the match pattern and in the expression to be matched. However, if all match predicates are mutually exclusive, the match result is insensitive to ordering, as one match predicate cannot accept terms matched by another.

Calling `matchdeclare` with a variable *a* as an argument changes the `matchdeclare` property for *a*, if one was already declared; only the most recent `matchdeclare` is in effect when a rule is defined, Later changes to the `matchdeclare` property (via `matchdeclare` or `remove`) do not affect existing rules.

`propvars (matchdeclare)` returns the list of all variables for which there is a `matchdeclare` property. `printprops (`*a*`, matchdeclare)` returns the predicate for variable `a`. `printprops (all, matchdeclare)` returns the list of predicates for all `matchdeclare` variables. `remove (`*a*`, matchdeclare)` removes the `matchdeclare` property from *a*.

The functions `defmatch`, `defrule`, `tellsimp`, `tellsimpafter`, and `let` construct rules which test expressions against patterns.

`matchdeclare` quotes its arguments. `matchdeclare` always returns `done`.

Examples:

A predicate is the name of a function, or a lambda expression, or a function call or lambda call missing the last argument, or `true` or `all`.

```
(%i1) matchdeclare (aa, integerp);
(%o1)                         done
(%i2) matchdeclare (bb, lambda ([x], x > 0));
(%o2)                         done
(%i3) matchdeclare (cc, freeof (%e, %pi, %i));
(%o3)                         done
(%i4) matchdeclare (dd, lambda ([x, y], gcd (x, y) = 1) (1728));
(%o4)                         done
(%i5) matchdeclare (ee, true);
(%o5)                         done
(%i6) matchdeclare (ff, all);
(%o6)                         done
```

If an expression satisfies a match predicate, the match variable is assigned the expression.

```
(%i1) matchdeclare (aa, integerp, bb, atom);
(%o1)                         done
(%i2) defrule (r1, bb^aa, ["integer" = aa, "atom" = bb]);
                     aa
(%o2)         r1 : bb   -> [integer = aa, atom = bb]
(%i3) r1 (%pi^8);
(%o3)                [integer = 8, atom = %pi]
```

In the case of addition and multiplication, the match variable may be assigned a single expression which satisfies the match predicate, or a sum or product (respectively) of such expressions.

```
(%i1) matchdeclare (aa, atom, bb, lambda ([x], not atom(x)));
(%o1)                         done
(%i2) defrule (r1, aa + bb, ["all atoms" = aa, "all nonatoms" =
                bb]);
bb + aa partitions 'sum'
(%o2)  r1 : bb + aa -> [all atoms = aa, all nonatoms = bb]
(%i3) r1 (8 + a*b + sin(x));
```

```
(%o3)     [all atoms = 8, all nonatoms = sin(x) + a b]
(%i4) defrule (r2, aa * bb, ["all atoms" = aa, "all nonatoms" =
               bb]);
bb aa partitions 'product'
(%o4)   r2 : aa bb -> [all atoms = aa, all nonatoms = bb]
(%i5) r2 (8 * (a + b) * sin(x));
(%o5)    [all atoms = 8, all nonatoms = (b + a) sin(x)]
```

When matching arguments of `+` and `*`, if all match predicates are mutually exclusive, the match result is insensitive to ordering, as one match predicate cannot accept terms matched by another.

```
(%i1) matchdeclare (aa, atom, bb, lambda ([x], not atom(x)));
(%o1)                         done
(%i2) defrule (r1, aa + bb, ["all atoms" = aa, "all nonatoms" =
               bb]);
bb + aa partitions 'sum'
(%o2)  r1 : bb + aa -> [all atoms = aa, all nonatoms = bb]
(%i3) r1 (8 + a*b + %pi + sin(x) - c + 2^n);
                                                          n
(%o3) [all atoms = %pi + 8, all nonatoms = sin(x) + 2  - c + a b]
(%i4) defrule (r2, aa * bb, ["all atoms" = aa, "all nonatoms" =
               bb]);
bb aa partitions 'product'
(%o4)   r2 : aa bb -> [all atoms = aa, all nonatoms = bb]
(%i5) r2 (8 * (a + b) * %pi * sin(x) / c * 2^n);
                                                   n
                                           (b + a) 2  sin(x)
(%o5) [all atoms = 8 %pi, all nonatoms = -----------------]
                                                 c
```

The functions `propvars` and `printprops` return information about match variables.

```
(%i1) matchdeclare ([aa, bb, cc], atom, [dd, ee], integerp);
(%o1)                         done
(%i2) matchdeclare (ff, floatnump, gg, lambda ([x], x > 100));
(%o2)                         done
(%i3) propvars (matchdeclare);
(%o3)             [aa, bb, cc, dd, ee, ff, gg]
(%i4) printprops (ee, matchdeclare);
(%o4)                     [integerp(ee)]
(%i5) printprops (gg, matchdeclare);
(%o5)              [lambda([x], x > 100, gg)]
(%i6) printprops (all, matchdeclare);
(%o6) [lambda([x], x > 100, gg), floatnump(ff), integerp(ee),
                    integerp(dd), atom(cc), atom(bb), atom(aa)]
```

matchfix (*ldelimiter*, *rdelimiter*) Function
matchfix (*ldelimiter*, *rdelimiter*, *arg_pos*, *pos*) Function

Declares a matchfix operator with left and right delimiters *ldelimiter* and *rdelimiter*. The delimiters are specified as strings.

A "matchfix" operator is a function of any number of arguments, such that the arguments occur between matching left and right delimiters. The delimiters may be any strings, so long as the parser can distinguish the delimiters from the operands and other expressions and operators. In practice this rules out unparseable delimiters such as `%`, `,`, `$` and `;`, and may require isolating the delimiters with white space. The right delimiter can be the same or different from the left delimiter.

A left delimiter can be associated with only one right delimiter; two different matchfix operators cannot have the same left delimiter.

An existing operator may be redeclared as a matchfix operator without changing its other properties. In particular, built-in operators such as addition `+` can be declared matchfix, but operator functions cannot be defined for built-in operators.

`matchfix` (*ldelimiter*, *rdelimiter*, *arg_pos*, *pos*) declares the argument part-of-speech *arg_pos* and result part-of-speech *pos*, and the delimiters *ldelimiter* and *rdelimiter*.

"Part of speech", in reference to operator declarations, means expression type. Three types are recognized: `expr`, `clause`, and `any`, indicating an algebraic expression, a Boolean expression, or any kind of expression, respectively. Maxima can detect some syntax errors by comparing the declared part of speech to an actual expression.

The function to carry out a matchfix operation is an ordinary user-defined function. The operator function is defined in the usual way with the function definition operator `:=` or `define`. The arguments may be written between the delimiters, or with the left delimiter as a quoted string and the arguments following in parentheses. `dispfun` (*ldelimiter*) displays the function definition.

The only built-in matchfix operator is the list constructor `[ ]`. Parentheses `( )` and double-quotes `" "` act like matchfix operators, but are not treated as such by the Maxima parser.

`matchfix` evaluates its arguments. `matchfix` returns its first argument, *ldelimiter*.

Examples:

- Delimiters may be almost any strings.

```
(%i1) matchfix ("@@", "~");
(%o1)                           @@
(%i2) @@ a, b, c ~;
(%o2)                       @@a, b, c~
(%i3) matchfix (">>", "<<");
(%o3)                           >>
(%i4) >> a, b, c <<;
(%o4)                       >>a, b, c<<
(%i5) matchfix ("foo", "oof");
(%o5)                           foo
(%i6) foo a, b, c oof;
(%o6)                     fooa, b, coof
(%i7) >> w + foo x, y oof + z << / @@ p, q ~;
                     >>z + foox, yoof + w<<
(%o7)                ----------------------
                            @@p, q~
```

- Matchfix operators are ordinary user-defined functions.

```
(%i1) matchfix ("!-", "-!");
(%o1)                          "!-"
(%i2) !- x, y -! := x/y - y/x;
                                        x   y
(%o2)                  !-x, y-! := - - -
                                        y   x
(%i3) define (!-x, y-!, x/y - y/x);
                                        x   y
(%o3)                  !-x, y-! := - - -
                                        y   x
(%i4) define ("!-" (x, y), x/y - y/x);
                                        x   y
(%o4)                  !-x, y-! := - - -
                                        y   x
(%i5) dispfun ("!-");
                                        x   y
(%t5)                  !-x, y-! := - - -
                                        y   x

(%o5)                         done
(%i6) !-3, 5-!;
                                  16
(%o6)                           - --
                                  15
(%i7) "!-" (3, 5);
                                  16
(%o7)                           - --
                                  15
```

remlet (*prod, name*) Function
remlet () Function
remlet (*all*) Function
remlet (*all, name*) Function

Deletes the substitution rule, *prod* –> 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*`,` *rulename*`)` removes the rule with the name *rulename* from the operator *op*. When *op* is a built-in or user-defined operator (as defined by `infix`, `prefix`, etc.), *op* and *rulename* must be enclosed in double quote marks.

`remrule (`*op*`, all)` removes all rules for the operator *op*.

See also `remlet`, which removes a rule defined by `let`.

Examples:

```
(%i1) tellsimp (foo (aa, bb), bb - aa);
(%o1)                    [foorule1, false]
(%i2) tellsimpafter (aa + bb, special_add (aa, bb));
(%o2)                    [+rule1, simplus]
(%i3) infix ("@@");
(%o3)                             @@
(%i4) tellsimp (aa @@ bb, bb/aa);
(%o4)                    [@@rule1, false]
(%i5) tellsimpafter (quux (%pi, %e), %pi - %e);
(%o5)                   [quuxrule1, false]
(%i6) tellsimpafter (quux (%e, %pi), %pi + %e);
(%o6)              [quuxrule2, quuxrule1, false]
(%i7) [foo (aa, bb), aa + bb, aa @@ bb, quux (%pi, %e),
       quux (%e, %pi)];
                                          bb
(%o7) [bb - aa, special_add(aa, bb), --, %pi - %e, %pi + %e]
                                          aa
(%i8) remrule (foo, foorule1);
(%o8)                            foo
(%i9) remrule ("+", "+rule1");
(%o9)                             +
(%i10) remrule ("@@", "@@rule1");
(%o10)                            @@
(%i11) remrule (quux, all);
(%o11)                           quux
(%i12) [foo (aa, bb), aa + bb, aa @@ bb, quux (%pi, %e),
        quux (%e, %pi)];
(%o12) [foo(aa, bb), bb + aa, aa @@ bb, quux(%pi, %e),
                                         quux(%e, %pi)]
```

tellsimp (*pattern, replacement*) Function

is similar to `tellsimpafter` but places new information before old so that it is applied before the built-in simplification rules.

`tellsimp` is used when it is important to modify the expression before the simplifier works on it, for instance if the simplifier "knows" something about the expression, but what it returns is not to your liking. If the simplifier "knows" something about the main operator of the expression, but is simply not doing enough for you, you probably want to use `tellsimpafter`.

The pattern may not be a sum, product, single variable, or number.

`rules` is the list of rules defined by `defrule`, `defmatch`, `tellsimp`, and `tellsimpafter`.

Examples:

```
(%i1) matchdeclare (x, freeof (%i));
(%o1)                         done
(%i2) %iargs: false$
(%i3) tellsimp (sin(%i*x), %i*sinh(x));
(%o3)                 [sinrule1, simp-%sin]
(%i4) trigexpand (sin (%i*y + x));
(%o4)         sin(x) cos(%i y) + %i cos(x) sinh(y)
(%i5) %iargs:true$
(%i6) errcatch(0^0);
 0
0  has been generated
(%o6)                           []
(%i7) ev (tellsimp (0^0, 1), simp: false);
(%o7)                  [^rule1, simpexpt]
(%i8) 0^0;
(%o8)                           1
(%i9) remrule ("^", %th(2)[1]);
(%o9)                           ^
(%i10) tellsimp (sin(x)^2, 1 - cos(x)^2);
(%o10)                 [^rule2, simpexpt]
(%i11) (1 + sin(x))^2;
                                       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 (trace $foorule1)` traces the function, and `:lisp (symbol-function '$foorule1` displays its definition.

`tellsimpafter` quotes its arguments. `tellsimpafter` returns the list of rules for the main operator of *pattern*, including the newly established rule.

See also `matchdeclare`, `defmatch`, `defrule`, `tellsimp`, `let`, `kill`, `remrule`, and `clear_rules`.

Examples:

pattern may be any nonatomic expression in which the main operator is not a pattern variable.

```
(%i1) matchdeclare (aa, atom, [ll, mm], listp, xx, true)$
(%i2) tellsimpafter (sin (ll), map (sin, ll));
```

```
(%o2)                  [sinrule1, simp-%sin]
(%i3) sin ([1/6, 1/4, 1/3, 1/2, 1]*%pi);
                    1  sqrt(2)  sqrt(3)
(%o3)              [-, -------, -------, 1, 0]
                    2     2        2
(%i4) tellsimpafter (ll^mm, map ("^", ll, mm));
(%o4)                  [^rule1, simpexpt]
(%i5) [a, b, c]^[1, 2, 3];
                                 2   3
(%o5)                       [a, b , c ]
(%i6) tellsimpafter (foo (aa (xx)), aa (foo (xx)));
(%o6)                    [foorule1, false]
(%i7) foo (bar (u - v));
(%o7)                     bar(foo(u - v))
```

Rules are applied in the order they were defined. If two rules can match an expression, the rule which was defined first is applied.

```
(%i1) matchdeclare (aa, integerp);
(%o1)                          done
(%i2) tellsimpafter (foo (aa), bar_1 (aa));
(%o2)                    [foorule1, false]
(%i3) tellsimpafter (foo (aa), bar_2 (aa));
(%o3)               [foorule2, foorule1, false]
(%i4) foo (42);
(%o4)                        bar_1(42)
```

Pattern variables are treated as local variables in simplification rules. (Compare to `defmatch`, which treats pattern variables as global variables.)

```
(%i1) matchdeclare (aa, integerp, bb, atom);
(%o1)                          done
(%i2) tellsimpafter (foo(aa, bb), bar('aa=aa, 'bb=bb));
(%o2)                    [foorule1, false]
(%i3) bb: 12345;
(%o3)                          12345
(%i4) foo (42, %e);
(%o4)                  bar(aa = 42, bb = %e)
(%i5) bb;
(%o5)                          12345
```

As with all atoms, properties of pattern variables are global even though values are local. In this example, an assignment property is declared via `define_variable`. This is a property of the atom `bb` throughout Maxima.

```
(%i1) matchdeclare (aa, integerp, bb, atom);
(%o1)                          done
(%i2) tellsimpafter (foo(aa, bb), bar('aa=aa, 'bb=bb));
(%o2)                    [foorule1, false]
(%i3) foo (42, %e);
(%o3)                  bar(aa = 42, bb = %e)
(%i4) define_variable (bb, true, boolean);
(%o4)                          true
(%i5) foo (42, %e);
```

```
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);
(%o11)                           ^
(%i12) rules;
(%o12)         [foorule2, foorule3, +rule1, *rule1]
```

A worked example: anticommutative multiplication.

```
(%i1) gt (i, j) := integerp(j) and i < j;
(%o1)           gt(i, j) := integerp(j) and i < j
(%i2) matchdeclare (i, integerp, j, gt(i));
(%o2)                          done
(%i3) tellsimpafter (s[i]^^2, 1);
(%o3)                  [^^rule1, simpncexpt]
(%i4) tellsimpafter (s[i] . s[j], -s[j] . s[i]);
(%o4)                    [.rule1, simpnct]
(%i5) s[1] . (s[1] + s[2]);
(%o5)                     s  . (s  + s )
                           1     2    1
(%i6) expand (%);
(%o6)                       1 - s  . s
                                2    1
(%i7) factor (expand (sum (s[i], i, 0, 9)^^5));
(%o7) 100 (s  + s  + s  + s  + s  + s  + s  + s  + s  + s )
           9    8    7    6    5    4    3    2    1    0
```

clear_rules () Function

Executes `kill (rules)` and then resets the next rule number to 1 for addition `+`, multiplication `*`, and exponentiation `^`.

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

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

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

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

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

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

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

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

37.2 Functions and Variables for Lists

append (*list_1, ..., list_n*) Function

Returns a single list of the elements of *list_1* followed by the elements of *list_2*, ... `append` also works on general expressions, e.g. `append (f(a,b), f(c,d,e));` yields `f(a,b,c,d,e)`.

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

assoc (*key, list, default*) Function

assoc (*key, list*) Function

This function searches for the *key* in the left hand side of the input *list* of the form `[x,y,z,...]` where each of the *list* elements is an expression of a binary operand and 2 elements. For example `x=1`, `2^3`, `[a,b]` etc. The *key* is checked againts 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 (*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` (asuming `a[1]` and `x` are unbound).

cons (*expr, 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 (*list*) Function

Returns a copy of the list *list*.

create_list (*form, x_1, list_1, ..., x_n, list_n*) Function

Create a list by evaluating *form* with *x_1* bound to each element of *list_1*, and for each such binding bind *x_2* to each element of *list_2*, ... The number of elements in the result will be the product of the number of elements in each list. Each variable *x_i* must actually be a symbol – it will not be evaluated. The list arguments will be evaluated once at the beginning of the iteration.

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

With a double iteration:

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

Instead of *list_i* two args may be supplied each of which should evaluate to a number. These will be the inclusive lower and upper bounds for the iteration.

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

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

delete (*expr_1, expr_2*) Function
delete (*expr_1, expr_2, n*) Function

Removes all occurrences of *expr_1* from *expr_2*. *expr_1* may be a term of *expr_2* (if it is a sum) or a factor of *expr_2* (if it is a product).

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

`delete`(*expr_1*, *expr_2*, *n*) removes the first *n* occurrences of *expr_1* from *expr_2*. If there are fewer than *n* occurrences of *expr_1* in *expr_2* then all occurrences will be deleted.

```
(%i1) delete(a, f(a,b,c,d,a));
(%o1)                       f(b, c, d)
(%i2) delete(a, f(a,b,a,c,d,a), 2);
(%o2)                      f(b, c, d, a)
```

eighth (*expr*) Function

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

endcons (*expr, list*) Function

Returns a new list consisting of the elements of `list` followed by *expr*. `endcons` also works on general expressions, e.g. `endcons(x, f(a,b,c));` -> `f(a,b,c,x)`.

fifth (*expr*) Function

Returns the 5'th item of expression or list *expr*. See `first` for more details.

first (*expr*) Function

Returns the first part of *expr* which may result in the first element of a list, the first row of a matrix, the first term of a sum, etc. Note that `first` and its related functions, `rest` and `last`, work on the form of *expr* which is displayed not the form which is typed on input. If the variable `inflag` is set to `true` however, these functions will look at the internal form of *expr*. Note that the simplifier re-orders expressions. Thus `first(x+y)` will be `x` if `inflag` is `true` and `y` if `inflag` is `false` (`first(y+x)` gives the same results). The functions `second` .. `tenth` yield the second through the tenth part of their input argument.

fourth (*expr*) Function

Returns the 4'th item of expression or list *expr*. See `first` for more details.

get (*a*, *i*) Function

Retrieves the user property indicated by *i* associated with atom *a* or returns `false` if a doesn't have property *i*.

`get` evaluates its arguments.

```
(%i1) put (%e, 'transcendental, 'type);
(%o1)                     transcendental
(%i2) put (%pi, 'transcendental, 'type)$
(%i3) put (%i, 'algebraic, 'type)$
(%i4) typeof (expr) := block ([q],
        if numberp (expr)
        then return ('algebraic),
        if not atom (expr)
        then return (maplist ('typeof, expr)),
        q: get (expr, 'type),
        if q=false
        then errcatch (error(expr,"is not numeric.")) else q)$
(%i5) typeof (2*%e + x*%pi);
x is not numeric.
(%o5)  [[transcendental, []], [algebraic, transcendental]]
(%i6) typeof (2*%e + %pi);
(%o6)     [transcendental, [algebraic, transcendental]]
```

join (*l*, *m*) Function

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

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

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

Examples:

```
(%i1) L1: [a, sin(b), c!, d - 1];
(%o1)                [a, sin(b), c!, d - 1]
(%i2) join (L1, [1, 2, 3, 4]);
```

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

last (*expr*) Function

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

length (*expr*) Function

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

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

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

Examples:

```
(%i1) makelist(concat(x,i),i,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.

unique (*L*) Function

Returns the unique elements of the list *L*.

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

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

Example:

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

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

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.

reverse (*list*) Function

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

second (*expr*) Function

Returns the 2'nd item of expression or list *expr*. See `first` for more details.

seventh (*expr*) Function

Returns the 7'th item of expression or list *expr*. See `first` for more details.

sixth (*expr*) Function

Returns the 6'th item of expression or list *expr*. See `first` for more details.

sublist_indices (*L*, *P*) Function

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

Examples:

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

tenth (*expr*) Function

Returns the 10'th item of expression or list *expr*. See `first` for more details.

third (*expr*) Function

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, ..., a_n`, write `set(a_1, ..., a_n)` or `{a_1, ..., a_n}`; to construct the empty set, write `set()` or `{}`. In input, `set(...)` and `{ ... }` are equivalent. Sets are always displayed with curly braces.

If a member is listed more than once, simplification eliminates the redundant member.

```
(%i1) set();
(%o1)                           {}
(%i2) set(a, b, a);
(%o2)                         {a, b}
(%i3) set(a, set(b));
(%o3)                        {a, {b}}
(%i4) set(a, [b]);
(%o4)                        {a, [b]}
(%i5) {};
(%o5)                           {}
(%i6) {a, b, a};
(%o6)                         {a, b}
(%i7) {a, {b}};
(%o7)                        {a, {b}}
(%i8) {a, [b]};
(%o8)                        {a, [b]}
```

Two would-be elements x and y are redundant (i.e., considered the same for the purpose of set construction) if and only if `is(`x `=` y`)` yields `true`. Note that `is(equal(`x`,` y`))` can yield `true` while `is(`x `=` y`)` yields `false`; in that case the elements x and y are considered distinct.

```
(%i1) x: a/c + b/c;
                                b   a
(%o1)                           - + -
                                c   c
(%i2) y: a/c + b/c;
                                b   a
(%o2)                           - + -
                                c   c
```

```
(%i3) z: (a + b)/c;
                                b + a
(%o3)                           -----
                                  c
(%i4) is (x = y);
(%o4)                           true
(%i5) is (y = z);
(%o5)                           false
(%i6) is (equal (y, z));
(%o6)                           true
(%i7) y - z;
                             b + a   b   a
(%o7)                      - ----- + - + -
                               c     c   c
(%i8) ratsimp (%);
(%o8)                             0
(%i9) {x, y, z};
                            b + a  b   a
(%o9)                      {-----, - + -}
                              c    c   c
```

To construct a set from the elements of a list, use `setify`.

```
(%i1) setify ([b, a]);
(%o1)                          {a, b}
```

Set members `x` and `y` are equal provided `is(x = y)` evaluates to `true`. Thus `rat(x)` and `x` are equal as set members; consequently,

```
(%i1) {x, rat(x)};
(%o1)                            {x}
```

Further, since `is((x - 1)*(x + 1) = x^2 - 1)` evaluates to `false`, `(x - 1)*(x + 1)` and `x^2 - 1` are distinct set members; thus

```
(%i1) {(x - 1)*(x + 1), x^2 - 1};
                                          2
(%o1)                {(x - 1) (x + 1), x  - 1}
```

To reduce this set to a singleton set, apply `rat` to each set member:

```
(%i1) {(x - 1)*(x + 1), x^2 - 1};
                                          2
(%o1)                {(x - 1) (x + 1), x  - 1}
(%i2) map (rat, %);
                                2
(%o2)/R/                       {x  - 1}
```

To remove redundancies from other sets, you may need to use other simplification functions. Here is an example that uses `trigsimp`:

```
(%i1) {1, cos(x)^2 + sin(x)^2};
                               2         2
(%o1)                  {1, sin (x) + cos (x)}
(%i2) map (trigsimp, %);
(%o2)                           {1}
```

A set is simplified when its members are non-redundant and sorted. The current version of the set functions uses the Maxima function `orderlessp` to order sets; however, *future versions of the set functions might use a different ordering function.*

Some operations on sets, such as substitution, automatically force a re-simplification; for example,

```
(%i1) s: {a, b, c}$
(%i2) subst (c=a, s);
(%o2)                          {a, b}
(%i3) subst ([a=x, b=x, c=x], s);
(%o3)                           {x}
(%i4) map (lambda ([x], x^2), set (-1, 0, 1));
(%o4)                         {0, 1}
```

Maxima treats lists and sets as distinct objects; functions such as `union` and `intersection` complain if any argument is not a set. If you need to apply a set function to a list, use the `setify` function to convert it to a set. Thus

```
(%i1) union ([1, 2], {a, b});
Function union expects a set, instead found [1,2]
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i2) union (setify ([1, 2]), {a, b});
(%o2)                      {1, 2, a, b}
```

To extract all set elements of a set `s` that satisfy a predicate `f`, use `subset(s, f)`. (A *predicate* is a boolean-valued function.) For example, to find the equations in a given set that do not depend on a variable `z`, use

```
(%i1) subset ({x + y + z, x - y + 4, x + y - 5},
                                    lambda ([e], freeof (z, e)));
(%o1)               {- y + x + 4, y + x - 5}
```

The section Section 38.2 [Functions and Variables for Sets], page 453 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:

```
(%i1) map (f, {a, b, c});
(%o1)                   {f(a), f(b), f(c)}
```

The other way is to use `for` *x* `in` *s* `do`

```
(%i1) s: {a, b, c};
(%o1)                         {a, b, c}
(%i2) for si in s do print (concat (si, 1));
a1
b1
c1
(%o2)                           done
```

The Maxima functions `first` and `rest` work correctly on sets. Applied to a set, `first` returns the first displayed element of a set; which element that is may be implementation-dependent. If `s` is a set, then `rest(s)` is equivalent to `disjoin(first(s), s)`. Currently, there are other Maxima functions that work correctly on sets. In future versions of the set functions, `first` and `rest` may function differently or not at all.

38.1.3 Bugs

The set functions use the Maxima function `orderlessp` to order set members and the (Lisp-level) function `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

```
(%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.
```

This expression causes Maxima to halt with an error (the error message depends on which version of Lisp your Maxima uses). Another example is

```
(%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.
```

These bugs are caused by bugs in `orderlessp` and `like`; they are not caused by bugs in the set functions. To illustrate, try the expressions

```
(%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.
(%i2) is ([rat(a)] = [rat(a)]);
(%o2)                          false
```

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:

```
(%i1) {x, rat (x)};
(%o1)                           {x}
```

Maxima's `orderlessp` has another bug that can cause problems with set functions, namely that the ordering predicate `orderlessp` is not transitive. The simplest known example that shows this is

```
(%i1) q: x^2$
(%i2) r: (x + 1)^2$
(%i3) s: x*(x + 2)$
(%i4) orderlessp (q, r);
(%o4)                          true
```

```
(%i5) orderlessp (r, s);
(%o5)                          true
(%i6) orderlessp (q, s);
(%o6)                          false
```

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 `ratsimp`.

Maxima's `orderless` and `ordergreat` mechanisms are incompatible with the set functions. If you need to use either `orderless` or `ordergreat`, call those functions before constructing any sets, and do not call `unorder`.

If you find something that you think might be a set function bug, please report it to the Maxima bug database. See `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

adjoin (*x*, *a*) Function

Returns the union of the set *a* with `{`*x*`}`.

`adjoin` complains if *a* is not a literal set.

`adjoin(`*x*`,` *a*`)` and `union(set(`*x*`),` *a*`)` are equivalent; however, `adjoin` may be somewhat faster than `union`.

See also `disjoin`.

Examples:

```
(%i1) adjoin (c, {a, b});
(%o1)                         {a, b, c}
(%i2) adjoin (a, {a, b});
(%o2)                          {a, b}
```

belln (*n*) Function

Represents the *n*-th Bell number. `belln(n)` is the number of partitions of a set with *n* members.

For nonnegative integers *n*, `belln(`*n*`)` simplifies to the *n*-th Bell number. `belln` does not simplify for any other arguments.

`belln` distributes over equations, lists, matrices, and sets.

Examples:

`belln` applied to nonnegative integers.

```
(%i1) makelist (belln (i), i, 0, 6);
(%o1)                [1, 1, 2, 5, 15, 52, 203]
(%i2) is (cardinality (set_partitions ({})) = belln (0));
```

```
(%o2)                          true
(%i3) is (cardinality (set_partitions ({1, 2, 3, 4, 5, 6})) =
                       belln (6));
(%o3)                          true
```

`belln` applied to arguments which are not nonnegative integers.

```
(%i1) [belln (x), belln (sqrt(3)), belln (-9)];
(%o1)        [belln(x), belln(sqrt(3)), belln(- 9)]
```

cardinality (*a*) Function

Returns the number of distinct elements of the set *a*.

`cardinality` ignores redundant elements even when simplification is disabled.

Examples:

```
(%i1) cardinality ({});
(%o1)                           0
(%i2) cardinality ({a, a, b, c});
(%o2)                           3
(%i3) simp : false;
(%o3)                          false
(%i4) cardinality ({a, a, b, c});
(%o4)                           3
```

cartesian_product (*b_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* unchanged.

`disjoin` complains if *a* is not a literal set.

`disjoin(`*x*`, `*a*`)`, `delete(`*x*`, `*a*`)`, and `setdifference(`*a*`, set(`*x*`))` are all equivalent. Of these, `disjoin` is generally faster than the others.

Examples:

```
(%i1) disjoin (a, {a, b, c, d});
(%o1)                         {b, c, d}
(%i2) disjoin (a + b, {5, z, a + b, %pi});
(%o2)                        {5, %pi, z}
(%i3) disjoin (a - b, {5, z, a + b, %pi});
(%o3)                   {5, %pi, b + a, z}
```

disjointp (*a*, *b*) Function

Returns `true` if and only if the sets *a* and *b* are disjoint.

`disjointp` complains if either *a* or *b* is not a literal set.

Examples:

```
(%i1) disjointp ({a, b, c}, {1, 2, 3});
(%o1)                           true
(%i2) disjointp ({a, b, 3}, {1, 2, 3});
(%o2)                           false
```

divisors (*n*) Function

Represents the set of divisors of *n*.

`divisors(`*n*`)` simplifies to a set of integers when *n* is a nonzero integer. The set of divisors includes the members 1 and *n*. The divisors of a negative integer are the divisors of its absolute value.

`divisors` distributes over equations, lists, matrices, and sets.

Examples:

We can verify that 28 is a perfect number: the sum of its divisors (except for itself) is 28.

```
(%i1) s: divisors(28);
(%o1)                 {1, 2, 4, 7, 14, 28}
(%i2) lreduce ("+", args(s)) - 28;
(%o2)                           28
```

`divisors` is a simplifying function. Substituting 8 for `a` in `divisors(a)` yields the divisors without reevaluating `divisors(8)`.

```
(%i1) divisors (a);
(%o1)                       divisors(a)
(%i2) subst (8, a, %);
(%o2)                      {1, 2, 4, 8}
```

`divisors` distributes over equations, lists, matrices, and sets.

```
(%i1) divisors (a = b);
(%o1)              divisors(a) = divisors(b)
(%i2) divisors ([a, b, c]);
(%o2)        [divisors(a), divisors(b), divisors(c)]
(%i3) divisors (matrix ([a, b], [c, d]));
                   [ divisors(a)  divisors(b) ]
(%o3)              [                          ]
                   [ divisors(c)  divisors(d) ]
(%i4) divisors ({a, b, c});
(%o4)        {divisors(a), divisors(b), divisors(c)}
```

elementp (*x*, *a*) Function

Returns `true` if and only if *x* is a member of the set *a*.

`elementp` complains if *a* is not a literal set.

Examples:

```
(%i1) elementp (sin(1), {sin(1), sin(2), sin(3)});
(%o1)                          true
(%i2) elementp (sin(1), {cos(1), cos(2), cos(3)});
(%o2)                          false
```

emptyp (*a*) Function

Return `true` if and only if *a* is the empty set or the empty list.

Examples:

```
(%i1) map (emptyp, [{}, []]);
(%o1)                       [true, true]
(%i2) map (emptyp, [a + b, {{}}, %pi]);
(%o2)                  [false, false, false]
```

equiv_classes (*s*, *F*) Function

Returns a set of the equivalence classes of the set *s* with respect to the equivalence relation *F*.

F is a function of two variables defined on the Cartesian product of *s* with *s*. The return value of *F* is either `true` or `false`, or an expression *expr* such that `is(`*expr*`)` is either `true` or `false`.

When *F* is not an equivalence relation, `equiv_classes` accepts it without complaint, but the result is generally incorrect in that case.

Examples:

The equivalence relation is a lambda expression which returns `true` or `false`.

```
(%i1) equiv_classes ({1, 1.0, 2, 2.0, 3, 3.0},
                         lambda ([x, y], is (equal (x, y))));
(%o1)            {{1, 1.0}, {2, 2.0}, {3, 3.0}}
```

The equivalence relation is the name of a relational function which `is` evaluates to `true` or `false`.

```
(%i1) equiv_classes ({1, 1.0, 2, 2.0, 3, 3.0}, equal);
(%o1)            {{1, 1.0}, {2, 2.0}, {3, 3.0}}
```

The equivalence classes are numbers which differ by a multiple of 3.

```
(%i1) equiv_classes ({1, 2, 3, 4, 5, 6, 7},
                      lambda ([x, y], remainder (x - y, 3) = 0));
(%o1)              {{1, 4, 7}, {2, 5}, {3, 6}}
```

every (*f*, *s*) Function

every (*f*, *L_1*, ..., *L_n*) Function

Returns `true` if the predicate *f* is `true` for all given arguments.

Given one set as the second argument, `every(`*f*`, `*s*`)` returns `true` if `is(`*f*`(`*a_i*`))` returns `true` for all *a_i* in *s*. `every` may or may not evaluate *f* for all *a_i* in *s*. Since sets are unordered, `every` may evaluate *f*`(`*a_i*`)` in any order.

Given one or more lists as arguments, `every(`*f*`, `*L_1*`, ..., `*L_n*`)` returns `true` if `is(`*f*`(`*x_1*`, ..., `*x_n*`))` returns `true` for all *x_1*, ..., *x_n* in *L_1*, ..., *L_n*, respectively. `every` may or may not evaluate *f* for every combination *x_1*, ..., *x_n*. `every` evaluates lists in the order of increasing index.

Given an empty set `{}` or empty lists `[]` as arguments, `every` returns `false`.

When the global flag `maperror` is `true`, all lists *L_1*, ..., *L_n* must have equal lengths. When `maperror` is `false`, list arguments are effectively truncated to the length of the shortest list.

Return values of the predicate *f* which evaluate (via `is`) to something other than `true` or `false` are governed by the global flag `prederror`. When `prederror` is `true`, such values are treated as `false`, and the return value from `every` is `false`. When `prederror` is `false`, such values are treated as `unknown`, and the return value from `every` is `unknown`.

Examples:

`every` applied to a single set. The predicate is a function of one argument.

```
(%i1) every (integerp, {1, 2, 3, 4, 5, 6});
(%o1)                          true
(%i2) every (atom, {1, 2, sin(3), 4, 5 + y, 6});
(%o2)                          false
```

`every` applied to two lists. The predicate is a function of two arguments.

```
(%i1) every ("=", [a, b, c], [a, b, c]);
(%o1)                          true
(%i2) every ("#", [a, b, c], [a, b, c]);
(%o2)                          false
```

Return values of the predicate *f* which evaluate to something other than `true` or `false` are governed by the global flag `prederror`.

```
(%i1) prederror : false;
(%o1)                          false
(%i2) map (lambda ([a, b], is (a < b)), [x, y, z],
                   [x^2, y^2, z^2]);
(%o2)               [unknown, unknown, unknown]
(%i3) every ("<", [x, y, z], [x^2, y^2, z^2]);
(%o3)                         unknown
(%i4) prederror : true;
(%o4)                          true
(%i5) every ("<", [x, y, z], [x^2, y^2, z^2]);
(%o5)                          false
```

extremal_subset (*s*, *f*, *max*) Function
extremal_subset (*s*, *f*, *min*) Function

Returns the subset of *s* for which the function *f* takes on maximum or minimum values.

`extremal_subset(`*s*`,` *f*`, max)` returns the subset of the set or list *s* for which the real-valued function *f* takes on its maximum value.

`extremal_subset(`*s*`,` *f*`, min)` returns the subset of the set or list *s* for which the real-valued function *f* takes on its minimum value.

Examples:

```
(%i1) extremal_subset ({-2, -1, 0, 1, 2}, abs, max);
(%o1)                        {- 2, 2}
(%i2) extremal_subset ({sqrt(2), 1.57, %pi/2}, sin, min);
(%o2)                        {sqrt(2)}
```

flatten (*expr*) Function

Collects arguments of subexpressions which have the same operator as *expr* and constructs an expression from these collected arguments.

Subexpressions in which the operator is different from the main operator of `expr` are copied without modification, even if they, in turn, contain some subexpressions in which the operator is the same as for `expr`.

It may be possible for `flatten` to construct expressions in which the number of arguments differs from the declared arguments for an operator; this may provoke an error message from the simplifier or evaluator. `flatten` does not try to detect such situations.

Expressions with special representations, for example, canonical rational expressions (CRE), cannot be flattened; in such cases, `flatten` returns its argument unchanged.

Examples:

Applied to a list, `flatten` gathers all list elements that are lists.

```
(%i1) flatten ([a, b, [c, [d, e], f], [[g, h]], i, j]);
(%o1)            [a, b, c, d, e, f, g, h, i, j]
```

Applied to a set, `flatten` gathers all members of set elements that are sets.

```
(%i1) flatten ({a, {b}, {{c}}});
(%o1)                        {a, b, c}
(%i2) flatten ({a, {[a], {a}}});
(%o2)                        {a, [a]}
```

`flatten` is similar to the effect of declaring the main operator n-ary. However, `flatten` has no effect on subexpressions which have an operator different from the main operator, while an n-ary declaration affects those.

```
(%i1) expr: flatten (f (g (f (f (x)))));
(%o1)                      f(g(f(f(x))))
(%i2) declare (f, nary);
(%o2)                         done
(%i3) ev (expr);
(%o3)                       f(g(f(x)))
```

`flatten` treats subscripted functions the same as any other operator.

```
(%i1) flatten (f[5] (f[5] (x, y), z));
(%o1)                       f (x, y, z)
                             5
```

It may be possible for `flatten` to construct expressions in which the number of arguments differs from the declared arguments for an operator;

```
(%i1) 'mod (5, 'mod (7, 4));
(%o1)                   mod(5, mod(7, 4))
(%i2) flatten (%);
(%o2)                     mod(5, 7, 4)
(%i3) ''%, nouns;
Wrong number of arguments to mod
 -- an error.  Quitting.  To debug this try debugmode(true);
```

full_listify (*a*) Function

Replaces every set operator in *a* by a list operator, and returns the result. `full_listify` replaces set operators in nested subexpressions, even if the main operator is not `set`.

`listify` replaces only the main operator.

Examples:

```
(%i1) full_listify ({a, b, {c, {d, e, f}, g}});
(%o1)                [a, b, [c, [d, e, f], g]]
(%i2) full_listify (F (G ({a, b, H({c, d, e})})));
(%o2)              F(G([a, b, H([c, d, e])]))
```

fullsetify (*a*) Function

When *a* is a list, replaces the list operator with a set operator, and applies `fullsetify` to each member which is a set. When *a* is not a list, it is returned unchanged.

`setify` replaces only the main operator.

Examples:

In line `(%o2)`, the argument of `f` isn't converted to a set because the main operator of `f([b])` isn't a list.

```
(%i1) fullsetify ([a, [a]]);
(%o1)                          {a, {a}}
(%i2) fullsetify ([a, f([b])]);
(%o2)                        {a, f([b])}
```

identity (*x*) Function

Returns *x* for any argument *x*.

Examples:

`identity` may be used as a predicate when the arguments are already Boolean values.

```
(%i1) every (identity, [true, true]);
(%o1)                           true
```

integer_partitions (*n*) Function
integer_partitions (*n*, *len*) Function

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:

```
(%i1) integer_partitions (3);
(%o1)                {[1, 1, 1], [2, 1], [3]}
(%i2) s: integer_partitions (25)$
(%i3) cardinality (s);
(%o3)                           1958
(%i4) map (lambda ([x], apply ("+", x)), s);
(%o4)                           {25}
(%i5) integer_partitions (5, 3);
(%o5) {[2, 2, 1], [3, 1, 1], [3, 2, 0], [4, 1, 0], [5, 0, 0]}
(%i6) integer_partitions (5, 2);
(%o6)                {[3, 2], [4, 1], [5, 0]}
```

To find all partitions that satisfy a condition, use the function `subset`; here is an example that finds all partitions of 10 that consist of prime numbers.

```
(%i1) s: integer_partitions (10)$
(%i2) cardinality (s);
(%o2)                            42
(%i3) xprimep(x) := integerp(x) and (x > 1) and primep(x)$
(%i4) subset (s, lambda ([x], every (xprimep, x)));
(%o4) {[2, 2, 2, 2, 2], [3, 3, 2, 2], [5, 3, 2], [5, 5], [7, 3]}
```

intersect (*a_1*, ..., *a_n*) Function

`intersect` is the same as `intersection`, which see.

intersection (*a_1*, ..., *a_n*) Function

Returns a set containing the elements that are common to the sets *a_1* through *a_n*.

`intersection` complains if any argument is not a literal set.

Examples:

```
(%i1) S_1 : {a, b, c, d};
(%o1)                        {a, b, c, d}
(%i2) S_2 : {d, e, f, g};
(%o2)                        {d, e, f, g}
(%i3) S_3 : {c, d, e, f};
(%o3)                        {c, d, e, f}
(%i4) S_4 : {u, v, w};
(%o4)                          {u, v, w}
(%i5) intersection (S_1, S_2);
```

```
(%o5)                          {d}
(%i6) intersection (S_2, S_3);
(%o6)                       {d, e, f}
(%i7) intersection (S_1, S_2, S_3);
(%o7)                          {d}
(%i8) intersection (S_1, S_2, S_3, S_4);
(%o8)                          {}
```

kron_delta (*x*, *y*) Function

Represents the Kronecker delta function.

`kron_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 `kron_delta` simplifies to a noun expression. `kron_delta` implements a cautious policy with respect to floating point expressions: if the difference *x* - *y* is a floating point number, `kron_delta` simplifies to a noun expression when *x* is apparently equivalent to *y*.

Specifically, `kron_delta(`*x*`,` *y*`)` simplifies to 1 when `is(x = y)` is `true`. `kron_delta` also simplifies to 1 when `sign(abs(`*x* - *y*`))` is `zero` and *x* - *y* is not a floating point number (neither an ordinary float nor a bigfloat). `kron_delta` simplifies to 0 when `sign(abs(`*x* - *y*`))` is `pos`.

Otherwise, `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, `kron_delta` returns a noun expression.

`kron_delta` is declared to be symmetric. That is, `kron_delta(`*x*`,` *y*`)` is equal to `kron_delta(`*y*`,` *x*`)`.

Examples:

The arguments of `kron_delta` are identical. `kron_delta` simplifies to 1.

```
(%i1) kron_delta (a, a);
(%o1)                          1
(%i2) kron_delta (x^2 - y^2, x^2 - y^2);
(%o2)                          1
(%i3) float (kron_delta (1/10, 0.1));
(%o3)                          1
```

The arguments of `kron_delta` are equivalent, and their difference is not a floating point number. `kron_delta` simplifies to 1.

```
(%i1) assume (equal (x, y));
(%o1)                     [equal(x, y)]
(%i2) kron_delta (x, y);
(%o2)                          1
```

The arguments of `kron_delta` are not equivalent. `kron_delta` simplifies to 0.

```
(%i1) kron_delta (a + 1, a);
(%o1)                          0
(%i2) assume (a > b)$
(%i3) kron_delta (a, b);
(%o3)                          0
```

```
(%i4) kron_delta (1/5, 0.7);
(%o4)                          0
```

The arguments of `kron_delta` might or might not be equivalent. `kron_delta` simplifies to a noun expression.

```
(%i1) kron_delta (a, b);
(%o1)                    kron_delta(a, b)
(%i2) assume(x >= y)$
(%i3) kron_delta (x, y);
(%o3)                    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.

```
(%i1) 1/4 - 0.25;
(%o1)                          0.0
(%i2) 1/10 - 0.1;
(%o2)                          0.0
(%i3) 0.25 - 0.25b0;
Warning:  Float to bigfloat conversion of 0.25
(%o3)                         0.0b0
(%i4) kron_delta (1/4, 0.25);
                                   1
(%o4)                  kron_delta(-, 0.25)
                                   4
(%i5) kron_delta (1/10, 0.1);
                                   1
(%o5)                  kron_delta(--, 0.1)
                                   10
(%i6) kron_delta (0.25, 0.25b0);
Warning:  Float to bigfloat conversion of 0.25
(%o6)               kron_delta(0.25, 2.5b-1)
```

`kron_delta` is symmetric.

```
(%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
```

listify (*a*) Function

Returns a list containing the members of *a* when *a* is a set. Otherwise, `listify` returns *a*.

`full_listify` replaces all set operators in *a* by list operators.

Examples:

```
(%i1) listify ({a, b, c, d});
(%o1)                        [a, b, c, d]
(%i2) listify (F ({a, b, c, d}));
(%o2)                       F({a, b, c, d})
```

lreduce (*F*, *s*) Function
lreduce (*F*, *s*, *s_0*) Function

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

`lreduce(`*F*`, `*s*`)` returns `F(... F(F(s_1, s_2), s_3), ... s_n)`. When the optional argument *s_0* is present, the result is equivalent to `lreduce(`*F*`, cons(`*s_0*`, `*s*`))`.

The function *F* is first applied to the *leftmost* list elements, thus the name "lreduce".

See also `rreduce`, `xreduce`, and `tree_reduce`.

Examples:

`lreduce` without the optional argument.

```
(%i1) lreduce (f, [1, 2, 3]);
(%o1)                        f(f(1, 2), 3)
(%i2) lreduce (f, [1, 2, 3, 4]);
(%o2)                     f(f(f(1, 2), 3), 4)
```

`lreduce` with the optional argument.

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

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

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

makeset (*expr*, *x*, *s*) Function

Returns a set with members generated from the expression *expr*, where *x* is a list of variables in *expr*, and *s* is a set or list of lists. To generate each set member, *expr* is evaluated with the variables *x* bound in parallel to a member of *s*.

Each member of *s* must have the same length as *x*. The list of variables *x* must be a list of symbols, without subscripts. Even if there is only one symbol, *x* must be a list of one element, and each member of *s* must be a list of one element.

See also `makelist`.

Examples:

```
(%i1) makeset (i/j, [i, j], [[1, a], [2, b], [3, c], [4, d]]);
                               1  2  3  4
(%o1)                         {-, -, -, -}
                               a  b  c  d
(%i2) S : {x, y, z}$
```

```
(%i3) S3 : cartesian_product (S, S, S);
(%o3) {[x, x, x], [x, x, y], [x, x, z], [x, y, x], [x, y, y],
[x, y, z], [x, z, x], [x, z, y], [x, z, z], [y, x, x],
[y, x, y], [y, x, z], [y, y, x], [y, y, y], [y, y, z],
[y, z, x], [y, z, y], [y, z, z], [z, x, x], [z, x, y],
[z, x, z], [z, y, x], [z, y, y], [z, y, z], [z, z, x],
[z, z, y], [z, z, z]}
(%i4) makeset (i + j + k, [i, j, k], S3);
(%o4) {3 x, 3 y, y + 2 x, 2 y + x, 3 z, z + 2 x, z + y + x,
                                        z + 2 y, 2 z + x, 2 z + y}
(%i5) makeset (sin(x), [x], {[1], [2], [3]});
(%o5)               {sin(1), sin(2), sin(3)}
```

moebius (*n*) Function

Represents the Moebius function.

When *n* is product of *k* distinct primes, `moebius(n)` simplifies to $(-1)^k$; when $n = 1$, it simplifies to 1; and it simplifies to 0 for all other positive integers.

`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 ]
(%o8)                       [        ]
                            [ - 1  1 ]
(%i9) moebius ({21, 22, 23, 24});
(%o9)                      {- 1, 0, 1}
```

multinomial_coeff (*a_1, ..., a_n*) Function
multinomial_coeff () 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* + ... + *a_n* distinct objects into *n* boxes with *a_k* elements in the *k*'th box. In general, `multinomial_coeff` (*a_1*, ..., *a_n*) evaluates to (*a_1* + ... + *a_n*)!/(*a_1*! ... *a_n*!).

`multinomial_coeff()` (with no arguments) evaluates to 1.

`minfactorial` may be able to simplify the value returned by `multinomial_coeff`.

Examples:

```
(%i1) multinomial_coeff (1, 2, x);
                              (x + 3)!
(%o1)                         --------
                                2 x!
(%i2) minfactorial (%);
                     (x + 1) (x + 2) (x + 3)
(%o2)                ----------------------
                                 2
(%i3) multinomial_coeff (-6, 2);
                              (- 4)!
(%o3)                         --------
                              2 (- 6)!
(%i4) minfactorial (%);
(%o4)                           10
```

num_distinct_partitions (*n*) Function
num_distinct_partitions (*n*, *list*) Function

Returns the number of distinct integer partitions of *n* when *n* is a nonnegative integer. Otherwise, `num_distinct_partitions` returns a noun expression.

`num_distinct_partitions(`*n*`, list)` returns a list of the number of distinct partitions of 1, 2, 3, ..., *n*.

A distinct partition of *n* is a list of distinct positive integers k_1, ..., k_m such that $n = k_1 + ... + k_m$.

Examples:

```
(%i1) num_distinct_partitions (12);
(%o1)                          15
(%i2) num_distinct_partitions (12, list);
(%o2)      [1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15]
(%i3) num_distinct_partitions (n);
(%o3)              num_distinct_partitions(n)
```

num_partitions (*n*) Function
num_partitions (*n*, *list*) Function

Returns the number of integer partitions of *n* when *n* is a nonnegative integer. Otherwise, `num_partitions` returns a noun expression.

`num_partitions(`*n*`, 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 `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:

```
(%i1) powerset ({a, b, c});
(%o1) {{}, {a}, {a, b}, {a, b, c}, {a, c}, {b}, {b, c}, {c}}
(%i2) powerset ({w, x, y, z}, 4);
(%o2)                    {{w, x, y, z}}
(%i3) powerset ({w, x, y, z}, 3);
(%o3)     {{w, x, y}, {w, x, z}, {w, y, z}, {x, y, z}}
(%i4) powerset ({w, x, y, z}, 2);
(%o4)   {{w, x}, {w, y}, {w, z}, {x, y}, {x, z}, {y, z}}
(%i5) powerset ({w, x, y, z}, 1);
(%o5)                 {{w}, {x}, {y}, {z}}
(%i6) powerset ({w, x, y, z}, 0);
(%o6)                         {{}}
```

random_permutation (*a*) Function

Returns a random permutation of the set or list *a*, as constructed by the Knuth shuffle algorithm.

The return value is a new list, which is distinct from the argument even if all elements happen to be the same. However, the elements of the argument are not copied.

Examples:

```
(%i1) random_permutation ([a, b, c, 1, 2, 3]);
(%o1)                   [c, 1, 2, 3, a, b]
(%i2) random_permutation ([a, b, c, 1, 2, 3]);
(%o2)                   [b, 3, 1, c, a, 2]
(%i3) random_permutation ({x + 1, y + 2, z + 3});
(%o3)                  [y + 2, z + 3, x + 1]
(%i4) random_permutation ({x + 1, y + 2, z + 3});
(%o4)                  [x + 1, y + 2, z + 3]
```

rreduce (*F*, *s*) Function

rreduce (*F*, *s*, *s_{n + 1}*) Function

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
(%o2)                          ---
                               b d
```

setdifference (*a*, *b*) Function

Returns a set containing the elements in the set *a* that are not in the set *b*.

`setdifference` complains if either *a* or *b* is not a literal set.

Examples:

```
(%i1) S_1 : {a, b, c, x, y, z};
(%o1)                  {a, b, c, x, y, z}
(%i2) S_2 : {aa, bb, c, x, y, zz};
(%o2)                 {aa, bb, c, x, y, zz}
(%i3) setdifference (S_1, S_2);
(%o3)                        {a, b, z}
(%i4) setdifference (S_2, S_1);
(%o4)                     {aa, bb, zz}
(%i5) setdifference (S_1, S_1);
(%o5)                           {}
(%i6) setdifference (S_1, {});
(%o6)                    {a, b, c, x, y, z}
(%i7) setdifference ({}, S_1);
(%o7)                           {}
```

setequalp (*a*, *b*) Function

Returns `true` if sets *a* and *b* have the same number of elements and `is(`*x* `=` *y*`)` is `true` for `x` in the elements of *a* and `y` in the elements of *b*, considered in the order determined by `listify`. Otherwise, `setequalp` returns `false`.

Examples:

```
(%i1) setequalp ({1, 2, 3}, {1, 2, 3});
(%o1)                          true
(%i2) setequalp ({a, b, c}, {1, 2, 3});
(%o2)                          false
(%i3) setequalp ({x^2 - y^2}, {(x + y) * (x - y)});
(%o3)                          false
```

setify (*a*) Function

Constructs a set from the elements of the list *a*. Duplicate elements of the list *a* are deleted and the elements are sorted according to the predicate `orderlessp`.

`setify` complains if *a* is not a literal list.

Examples:

```
(%i1) setify ([1, 2, 3, a, b, c]);
(%o1)                  {1, 2, 3, a, b, c}
(%i2) setify ([a, b, c, a, b, c]);
(%o2)                        {a, b, c}
(%i3) setify ([7, 13, 11, 1, 3, 9, 5]);
(%o3)                {1, 3, 5, 7, 9, 11, 13}
```

setp (*a*) Function

Returns `true` if and only if *a* is a Maxima set.

`setp` returns `true` for unsimplified sets (that is, sets with redundant members) as well as simplified sets.

`setp` is equivalent to the Maxima function `setp(a) := not atom(a) and op(a) = 'set`.

Examples:

```
(%i1) simp : false;
(%o1)                          false
(%i2) {a, a, a};
(%o2)                        {a, a, a}
(%i3) setp (%);
(%o3)                          true
```

set_partitions (*a*) Function
set_partitions (*a*, *n*) Function

Returns the set of all partitions of *a*, or a subset of that set.

`set_partitions`(*a*, *n*) returns a set of all decompositions of *a* into *n* nonempty disjoint subsets.

`set_partitions`(*a*) returns the set of all partitions.

`stirling2` returns the cardinality of the set of partitions of a set.

A set of sets P is a partition of a set S when

1. each member of P is a nonempty set,
2. distinct members of P are disjoint,
3. the union of the members of P equals S.

Examples:

The empty set is a partition of itself, the conditions 1 and 2 being vacuously true.

```
(%i1) set_partitions ({});
(%o1)                          {{}}
```

The cardinality of the set of partitions of a set can be found using `stirling2`.

```
(%i1) s: {0, 1, 2, 3, 4, 5}$
(%i2) p: set_partitions (s, 3)$
(%i3) cardinality(p) = stirling2 (6, 3);
(%o3)                         90 = 90
```

Each member of `p` should have $n = 3$ members; let's check.

```
(%i1) s: {0, 1, 2, 3, 4, 5}$
(%i2) p: set_partitions (s, 3)$
(%i3) map (cardinality, p);
(%o3)                          {3}
```

Finally, for each member of `p`, the union of its members should equal `s`; again let's check.

```
(%i1) s: {0, 1, 2, 3, 4, 5}$
(%i2) p: set_partitions (s, 3)$
(%i3) map (lambda ([x], apply (union, listify (x))), p);
(%o3)                 {{0, 1, 2, 3, 4, 5}}
```

some (*f*, *a*) Function
some (*f*, *L_1*, ..., *L_n*) Function

Returns `true` if the predicate *f* is `true` for one or more given arguments.

Given one set as the second argument, `some(`*f*`,` *s*`)` returns `true` if `is(`*f*`(`*a_i*`))` returns `true` for one or more *a_i* in *s*. `some` may or may not evaluate *f* for all *a_i* in *s*. Since sets are unordered, `some` may evaluate *f*`(`*a_i*`)` in any order.

Given one or more lists as arguments, `some(`*f*`,` *L_1*`, ...,` *L_n*`)` returns `true` if `is(`*f*`(`*x_1*`, ...,` *x_n*`))` returns `true` for one or more *x_1*, ..., *x_n* in *L_1*, ..., *L_n*, respectively. `some` may or may not evaluate *f* for some combinations *x_1*, ..., *x_n*. `some` evaluates lists in the order of increasing index.

Given an empty set `{}` or empty lists `[]` as arguments, `some` returns `false`.

When the global flag `maperror` is `true`, all lists *L_1*, ..., *L_n* must have equal lengths. When `maperror` is `false`, list arguments are effectively truncated to the length of the shortest list.

Return values of the predicate *f* which evaluate (via `is`) to something other than `true` or `false` are governed by the global flag `prederror`. When `prederror` is `true`, such values are treated as `false`. When `prederror` is `false`, such values are treated as `unknown`.

Examples:

`some` applied to a single set. The predicate is a function of one argument.

```
(%i1) some (integerp, {1, 2, 3, 4, 5, 6});
(%o1)                          true
(%i2) some (atom, {1, 2, sin(3), 4, 5 + y, 6});
(%o2)                          true
```

`some` applied to two lists. The predicate is a function of two arguments.

```
(%i1) some ("=", [a, b, c], [a, b, c]);
(%o1)                          true
(%i2) some ("#", [a, b, c], [a, b, c]);
(%o2)                          false
```

Return values of the predicate *f* which evaluate to something other than `true` or `false` are governed by the global flag `prederror`.

```
(%i1) prederror : false;
(%o1)                          false
```

```
(%i2) map (lambda ([a, b], is (a < b)), [x, y, z],
          [x^2, y^2, z^2]);
(%o2)              [unknown, unknown, unknown]
(%i3) some ("<", [x, y, z], [x^2, y^2, z^2]);
(%o3)                          unknown
(%i4) some ("<", [x, y, z], [x^2, y^2, z + 1]);
(%o4)                           true
(%i5) prederror : true;
(%o5)                           true
(%i6) some ("<", [x, y, z], [x^2, y^2, z^2]);
(%o6)                           false
(%i7) some ("<", [x, y, z], [x^2, y^2, z + 1]);
(%o7)                           true
```

stirling1 (*n*, *m*) Function

Represents the Stirling number of the first kind.

When *n* and *m* are nonnegative integers, the magnitude of `stirling1 (n, m)` is the number of permutations of a set with *n* members that have *m* cycles. For details, see Graham, Knuth and Patashnik *Concrete Mathematics*. Maxima uses a recursion relation to define `stirling1 (n, m)` for *m* less than 0; it is undefined for *n* less than 0 and for non-integer arguments.

`stirling1` is a simplifying function. Maxima knows the following identities.

1. $stirling1(0, n) = kron_delta(0, n)$ (Ref. [1])
2. $stirling1(n, n) = 1$ (Ref. [1])
3. $stirling1(n, n-1) = binomial(n, 2)$ (Ref. [1])
4. $stirling1(n+1, 0) = 0$ (Ref. [1])
5. $stirling1(n+1, 1) = n!$ (Ref. [1])
6. $stirling1(n+1, 2) = 2^n - 1$ (Ref. [1])

These identities are applied when the arguments are literal integers or symbols declared as integers, and the first argument is nonnegative. `stirling1` does not simplify for non-integer arguments.

References:

[1] Donald Knuth, *The Art of Computer Programming,* third edition, Volume 1, Section 1.2.6, Equations 48, 49, and 50.

Examples:

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling1 (n, n);
(%o3)                           1
```

`stirling1` does not simplify for non-integer arguments.

```
(%i1) stirling1 (sqrt(2), sqrt(2));
(%o1)              stirling1(sqrt(2), sqrt(2))
```

Maxima applies identities to `stirling1`.

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling1 (n + 1, n);
                                n (n + 1)
(%o3)                           ---------
                                    2
(%i4) stirling1 (n + 1, 1);
(%o4)                              n!
```

stirling2 (*n*, *m*) Function

Represents the Stirling number of the second kind.

When *n* and *m* are nonnegative integers, `stirling2 (n, m)` is the number of ways a set with cardinality *n* can be partitioned into *m* disjoint subsets. Maxima uses a recursion relation to define `stirling2 (n, m)` for *m* less than 0; it is undefined for *n* less than 0 and for non-integer arguments.

`stirling2` is a simplifying function. Maxima knows the following identities.

1. $stirling2(0, n) = kron_delta(0, n)$ (Ref. [1])
2. $stirling2(n, n) = 1$ (Ref. [1])
3. $stirling2(n, n-1) = binomial(n, 2)$ (Ref. [1])
4. $stirling2(n+1, 1) = 1$ (Ref. [1])
5. $stirling2(n+1, 2) = 2^n - 1$ (Ref. [1])
6. $stirling2(n, 0) = kron_delta(n, 0)$ (Ref. [2])
7. $stirling2(n, m) = 0$ when $m > n$ (Ref. [2])
8. $stirling2(n, m) = sum((-1)^(m-k) binomial(mk) k^n, i, 1, m)/m!$ 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. `stirling2` does not simplify for non-integer arguments.

References:

[1] Donald Knuth. *The Art of Computer Programming*, third edition, Volume 1, Section 1.2.6, Equations 48, 49, and 50.

[2] Graham, Knuth, and Patashnik. *Concrete Mathematics*, Table 264.

[3] Abramowitz and Stegun. *Handbook of Mathematical Functions*, Section 24.1.4.

Examples:

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling2 (n, n);
(%o3)                              1
```

`stirling2` does not simplify for non-integer arguments.

```
(%i1) stirling2 (%pi, %pi);
(%o1)                    stirling2(%pi, %pi)
```

Maxima applies identities to `stirling2`.

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling2 (n + 9, n + 8);
                          (n + 8) (n + 9)
(%o3)                     ---------------
                                 2
(%i4) stirling2 (n + 1, 2);
                               n
(%o4)                         2  - 1
```

subset (*a*, *f*) Function

Returns the subset of the set *a* that satisfies the predicate *f*.

`subset` returns a set which comprises the elements of *a* for which *f* returns anything other than `false`. `subset` does not apply `is` to the return value of *f*.

`subset` complains if *a* is not a literal set.

See also `partition_set`.

Examples:

```
(%i1) subset ({1, 2, x, x + y, z, x + y + z}, atom);
(%o1)                      {1, 2, x, z}
(%i2) subset ({1, 2, 7, 8, 9, 14}, evenp);
(%o2)                      {2, 8, 14}
```

subsetp (*a*, *b*) Function

Returns `true` if and only if the set *a* is a subset of *b*.

`subsetp` complains if either *a* or *b* is not a literal set.

Examples:

```
(%i1) subsetp ({1, 2, 3}, {a, 1, b, 2, c, 3});
(%o1)                          true
(%i2) subsetp ({a, 1, b, 2, c, 3}, {1, 2, 3});
(%o2)                          false
```

symmdifference (*a_1*, ..., *a_n*) Function

Returns the symmetric difference, that is, the set of members that occur in exactly one set *a_k*.

Given two arguments, `symmdifference` (*a*, *b*) is the same as `union` (`setdifference` (*a*, *b*), `setdifference`(*b*, *a*)).

`symmdifference` complains if any argument is not a literal set.

Examples:

```
(%i1) S_1 : {a, b, c};
(%o1)                          {a, b, c}
(%i2) S_2 : {1, b, c};
(%o2)                          {1, b, c}
(%i3) S_3 : {a, b, z};
(%o3)                          {a, b, z}
```

```
(%i4) symmdifference ();
(%o4)                           {}
(%i5) symmdifference (S_1);
(%o5)                         {a, b, c}
(%i6) symmdifference (S_1, S_2);
(%o6)                          {1, a}
(%i7) symmdifference (S_1, S_2, S_3);
(%o7)                          {1, z}
(%i8) symmdifference ({}, S_1, S_2, S_3);
(%o8)                          {1, z}
```

tree_reduce (*F*, *s*) Function
tree_reduce (*F*, *s*, *s_0*) Function

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

`tree_reduce` is equivalent to the following: Apply *F* to successive pairs of elements to form a new list `[`*F*`(`*s_1*`, `*s_2*`), `*F*`(`*s_3*`, `*s_4*`), ...]`, carrying the final element unchanged if there are an odd number of elements. Then repeat until the list is reduced to a single element, which is the return value.

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

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

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

Examples:

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

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

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

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

union (*a_1*, ..., *a_n*) Function

Returns the union of the sets *a_1* through *a_n*.

`union()` (with no arguments) returns the empty set.

`union` complains if any argument is not a literal set.

Examples:

```
(%i1) S_1 : {a, b, c + d, %e};
(%o1)                    {%e, a, b, d + c}
(%i2) S_2 : {%pi, %i, %e, c + d};
(%o2)                  {%e, %i, %pi, d + c}
(%i3) S_3 : {17, 29, 1729, %pi, %i};
(%o3)                {17, 29, 1729, %i, %pi}
```

```
(%i4) union ();
(%o4)                          {}
(%i5) union (S_1);
(%o5)                    {%e, a, b, d + c}
(%i6) union (S_1, S_2);
(%o6)              {%e, %i, %pi, a, b, d + c}
(%i7) union (S_1, S_2, S_3);
(%o7)       {17, 29, 1729, %e, %i, %pi, a, b, d + c}
(%i8) union ({}, S_1, S_2, S_3);
(%o8)       {17, 29, 1729, %e, %i, %pi, a, b, d + c}
```

xreduce (*F*, *s*) Function
xreduce (*F*, *s*, *s_0*) 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, `xreduce` is the same as `lreduce`. The argument *s* is a list.

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

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

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

Examples:

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

```
(%i1) declare (F, nary);
(%o1)                          done
(%i2) F ([L]) := L;
(%o2)                       F([L]) := L
(%i3) xreduce (F, [a, b, c, d, e]);
(%o3)         [[[[[("[", simp), a], b], c], d], e]
```

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

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

39 Function Definition

39.1 Introduction to Function Definition

39.2 Function

39.2.1 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

```
lambda ([i, j], ...)
```

can be used instead of `f` where

```
f(i,j) := block ([], ...);
map (lambda ([i], i+1), l)
```

would return a list with 1 added to each term.

You may also define a function with a variable number of arguments, by having a final argument which is assigned to a list of the extra arguments:

```
(%i1) f ([u]) := u;
(%o1)                         f([u]) := u
(%i2) f (1, 2, 3, 4);
(%o2)                        [1, 2, 3, 4]
(%i3) f (a, b, [u]) := [a, b, u];
(%o3)                f(a, b, [u]) := [a, b, u]
(%i4) f (1, 2, 3, 4, 5, 6);
(%o4)                   [1, 2, [3, 4, 5, 6]]
```

The right hand side of a function is an expression. Thus if you want a sequence of expressions, you do

```
f(x) := (expr1, expr2, ...., exprn);
```

and the value of *exprn* is what is returned by the function.

If you wish to make a `return` from some expression inside the function then you must use `block` and `return`.

```
block ([], expr1, ..., if (a > 10) then return(a), ..., exprn)
```

is itself an expression, and so could take the place of the right hand side of a function definition. Here it may happen that the return happens earlier than the last expression.

The first `[]` in the block, may contain a list of variables and variable assignments, such as `[a: 3, b, c: []]`, which would cause the three variables `a`,`b`,and `c` to not refer to their global values, but rather have these special values for as long as the code executes inside the `block`, or inside functions called from inside the `block`. This is called *dynamic* binding, since the variables last from the start of the block to the time it exits. Once you return from the `block`, or throw out of it, the old values (if any) of the variables will be restored. It is certainly a good idea to protect your variables in this way. Note that the assignments

in the block variables, are done in parallel. This means, that if you had used `c: a` in the above, the value of `c` would have been the value of `a` at the time you just entered the block, but before `a` was bound. Thus doing something like

```
block ([a: a], expr1, ... a: a+3, ..., exprn)
```

will protect the external value of `a` from being altered, but would let you access what that value was. Thus the right hand side of the assignments, is evaluated in the entering context, before any binding occurs. Using just `block ([x], ...` would cause the `x` to have itself as value, just as if it would have if you entered a fresh Maxima session.

The actual arguments to a function are treated in exactly same way as the variables in a block. Thus in

```
f(x) := (expr1, ..., exprn);
```

and

```
f(1);
```

we would have a similar context for evaluation of the expressions as if we had done

```
block ([x: 1], expr1, ..., exprn)
```

Inside functions, when the right hand side of a definition, may be computed at runtime, it is useful to use `define` and possibly `buildq`.

39.2.2 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 `arrays` (not the global list `functions`). `arrayinfo` returns the list of arguments for which there are stored values, and `listarray` returns the stored values. `dispfun` and `fundef` return the array function definition.

`arraymake` constructs an array function call, analogous to `funmake` for ordinary functions. `arrayapply` applies an array function to its arguments, analogous to `apply` for ordinary functions. There is nothing exactly analogous to `map` for array functions, although `map(lambda([`*x*`], `*a*`[`*x*`]), `*L*`)` or `makelist(`*a*`[`*x*`], `*x*`, `*L*`)`, where *L* is a list, are not too far off the mark.

`remarray` removes an array function definition (including any stored function values), analogous to `remfunction` for ordinary functions.

`kill(`*a*`[`*x*`])` removes the value of the array function *a* stored for the argument *x*; the next time *a* is called with argument *x*, the function value is recomputed. However, there is no way to remove all of the stored values at once, except for `kill(`*a*`)` or `remarray(`*a*`)`, which also remove the function definition.

39.3 Macros

buildq (*L, expr*) Function

Substitutes variables named by the list *L* into the expression *expr*, in parallel, without evaluating *expr*. The resulting expression is simplified, but not evaluated, after `buildq` carries out the substitution.

The elements of L are symbols or assignment expressions *symbol*: *value*, evaluated in parallel. That is, the binding of a variable on the right-hand side of an assignment is the binding of that variable in the context from which `buildq` was called, not the binding of that variable in the variable list L. If some variable in L is not given an explicit assignment, its binding in `buildq` is the same as in the context from which `buildq` was called.

Then the variables named by L are substituted into *expr* in parallel. That is, the substitution for every variable is determined before any substitution is made, so the substitution for one variable has no effect on any other.

If any variable x appears as `splice (x)` in *expr*, then x must be bound to a list, and the list is spliced (interpolated) into *expr* instead of substituted.

Any variables in *expr* not appearing in L are carried into the result verbatim, even if they have bindings in the context from which `buildq` was called.

Examples

`a` is explicitly bound to `x`, while `b` has the same binding (namely 29) as in the calling context, and `c` is carried through verbatim. The resulting expression is not evaluated until the explicit evaluation `''%`.

```
(%i1) (a: 17, b: 29, c: 1729)$
(%i2) buildq ([a: x, b], a + b + c);
(%o2)                       x + c + 29
(%i3) ''%;
(%o3)                        x + 1758
```

`e` is bound to a list, which appears as such in the arguments of `foo`, and interpolated into the arguments of `bar`.

```
(%i1) buildq ([e: [a, b, c]], foo (x, e, y));
(%o1)                  foo(x, [a, b, c], y)
(%i2) buildq ([e: [a, b, c]], bar (x, splice (e), y));
(%o2)                   bar(x, a, b, c, y)
```

The result is simplified after substitution. If simplification were applied before substitution, these two results would be the same.

```
(%i1) buildq ([e: [a, b, c]], splice (e) + splice (e));
(%o1)                     2 c + 2 b + 2 a
(%i2) buildq ([e: [a, b, c]], 2 * splice (e));
(%o2)                         2 a b c
```

The variables in L are bound in parallel; if bound sequentially, the first result would be `foo (b, b)`. Substitutions are carried out in parallel; compare the second result with the result of `subst`, which carries out substitutions sequentially.

```
(%i1) buildq ([a: b, b: a], foo (a, b));
(%o1)                         foo(b, a)
(%i2) buildq ([u: v, v: w, w: x, x: y, y: z, z: u],
              bar (u, v, w, x, y, z));
(%o2)                  bar(v, w, x, y, z, u)
(%i3) subst ([u=v, v=w, w=x, x=y, y=z, z=u],
             bar (u, v, w, x, y, z));
(%o3)                  bar(u, u, u, u, u, u)
```

Construct a list of equations with some variables or expressions on the left-hand side and their values on the right-hand side. `macroexpand` shows the expression returned by `show_values`.

```
(%i1) show_values ([L]) ::= buildq ([L], map ("=", 'L, L));
(%o1)   show_values([L]) ::= buildq([L], map("=", 'L, L))
(%i2) (a: 17, b: 29, c: 1729)$
(%i3) show_values (a, b, c - a - b);
(%o3)          [a = 17, b = 29, c - b - a = 1683]
(%i4) macroexpand (show_values (a, b, c - a - b));
(%o4)    map(=, '([a, b, c - b - a]), [a, b, c - b - a])
```

Given a function of several arguments, create another function for which some of the arguments are fixed.

```
(%i1) curry (f, [a]) :=
        buildq ([f, a], lambda ([[x]], apply (f, append (a, x))))$
(%i2) by3 : curry ("*", 3);
(%o2)        lambda([[x]], apply(*, append([3], x)))
(%i3) by3 (a + b);
(%o3)                        3 (b + a)
```

macroexpand (*expr*) Function

Returns the macro expansion of *expr* without evaluating it, when `expr` is a macro function call. Otherwise, `macroexpand` returns *expr*.

If the expansion of *expr* yields another macro function call, that macro function call is also expanded.

`macroexpand` quotes its argument. However, if the expansion of a macro function call has side effects, those side effects are executed.

See also `::=`, `macros`, and `macroexpand1`.

Examples

```
(%i1) g (x) ::= x / 99;
                                     x
(%o1)                       g(x) ::= --
                                     99
(%i2) h (x) ::= buildq ([x], g (x - a));
(%o2)            h(x) ::= buildq([x], g(x - a))
(%i3) a: 1234;
(%o3)                           1234
(%i4) macroexpand (h (y));
                                y - a
(%o4)                           -----
                                 99
(%i5) h (y);
                               y - 1234
(%o5)                          --------
                                  99
```

macroexpand1 (*expr*) Function

Returns the macro expansion of *expr* without evaluating it, when `expr` is a macro function call. Otherwise, `macroexpand1` returns *expr*.

`macroexpand1` quotes its argument. However, if the expansion of a macro function call has side effects, those side effects are executed.

If the expansion of *expr* yields another macro function call, that macro function call is not expanded.

See also `::=`, `macros`, and `macroexpand`.

Examples

```
(%i1) g (x) ::= x / 99;
                                      x
(%o1)                       g(x) ::= --
                                      99
(%i2) h (x) ::= buildq ([x], g (x - a));
(%o2)             h(x) ::= buildq([x], g(x - a))
(%i3) a: 1234;
(%o3)                           1234
(%i4) macroexpand1 (h (y));
(%o4)                          g(y - a)
(%i5) h (y);
                               y - 1234
(%o5)                          --------
                                  99
```

macros Global variable

Default value: `[]`

`macros` is the list of user-defined macro functions. The macro function definition operator `::=` puts a new macro function onto this list, and `kill`, `remove`, and `remfunction` remove macro functions from the list.

See also `infolists`.

splice (*a*) Function

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

```
(%i1) buildq ([x: [1, %pi, z - y]], foo (splice (x)) / length (x));
                        foo(1, %pi, z - y)
(%o1)                  ----------------------
                       length([1, %pi, z - y])
(%i2) buildq ([x: [1, %pi]], "/" (splice (x)));
                                 1
(%o2)                            ---
                                 %pi
(%i3) matchfix ("<>", "<>");
(%o3)                            <>
(%i4) buildq ([x: [1, %pi, z - y]], "<>" (splice (x)));
(%o4)                     <>1, %pi, z - y<>
```

39.4 Functions and Variables for Function Definition

apply (*F*, [*x_1*, ..., *x_n*]) Function

Constructs and evaluates an expression *F*(*arg_1*, ..., *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*`(...)` (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, `min` is applied to the value of `L`.

```
(%i1) L : [1, 5, -10.2, 4, 3];
(%o1)                  [1, 5, - 10.2, 4, 3]
(%i2) apply (min, L);
(%o2)                          - 10.2
```

`apply` evaluates arguments, even if the function *F* quotes them.

```
(%i1) F (x) := x / 1729;
                                     x
(%o1)                      F(x) := ----
                                    1729
(%i2) fname : F;
(%o2)                               F
(%i3) dispfun (F);
                                     x
(%t3)                      F(x) := ----
                                    1729

(%o3)                           [%t3]
(%i4) dispfun (fname);
fname is not the name of a user function.
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i5) apply (dispfun, [fname]);
                                     x
(%t5)                      F(x) := ----
```

```
                                1729

(%o5)                          [%t5]
```

`apply` evaluates the function name *F*. Single quote `'` defeats evaluation. `demoivre` is the name of a global variable and also a function.

```
(%i1) demoivre;
(%o1)                         false
(%i2) demoivre (exp (%i * x));
(%o2)                  %i sin(x) + cos(x)
(%i3) apply (demoivre, [exp (%i * x)]);
demoivre evaluates to false
Improper name or value in functional position.
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i4) apply ('demoivre, [exp (%i * x)]);
(%o4)                  %i sin(x) + cos(x)
```

block ([*v_1*, ..., *v_m*], *expr_1*, ..., *expr_n*) Function
block (*expr_1*, ..., *expr_n*) Function

`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.

`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".

If it is desired to save and restore other local properties besides `value`, for example `array` (except for complete arrays), `function`, `dependencies`, `atvalue`, `matchdeclare`, `atomgrad`, `constant`, and `nonscalar` then the function `local` should be used inside of the block with arguments being the names of the variables.

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*) Function

Evaluates and prints *expr_1*, ..., *expr_n* and then causes a Maxima break at which point the user can examine and change his environment. Upon typing `exit;` the computation resumes.

catch (*expr_1, ..., expr_n*) Function

Evaluates *expr_1*, ..., *expr_n* one by one; if any leads to the evaluation of an expression of the form `throw (arg)`, then the value of the `catch` is the value of `throw (arg)`, and no further expressions are evaluated. This "non-local return" thus goes through any depth of nesting to the nearest enclosing `catch`. If there is no `catch` enclosing a `throw`, an error message is printed.

If the evaluation of the arguments does not lead to the evaluation of any `throw` then the value of `catch` is the value of *expr_n*.

```
(%i1) lambda ([x], if x < 0 then throw(x) else f(x))$
(%i2) g(l) := catch (map (''%, l))$
(%i3) g ([1, 2, 3, 7]);
(%o3)                [f(1), f(2), f(3), f(7)]
(%i4) g ([1, 2, -3, 7]);
(%o4)                            - 3
```

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*) Function
compfile (*filename, functions*) Function
compfile (*filename, all*) Function

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*) Function
compile (*functions*) Function
compile (*all*) Function

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.

`compile (all)` or `compile (functions)` compiles all user-defined functions.

`compile` quotes its arguments; the quote-quote operator `''` defeats quotation.

define (*f*(*x_1*, ..., *x_n*), *expr*) Function
define (*f*[*x_1*, ..., *x_n*], *expr*) Function
define (*funmake* (*f*, [*x_1*, ..., *x_n*]), *expr*) Function
define (*arraymake* (*f*, [*x_1*, ..., *x_n*]), *expr*) Function
define (*ev* (*expr_1*), *expr_2*) Function

Defines a function named *f* with arguments *x_1*, ..., *x_n* and function body *expr*. `define` always evaluates its second argument (unless explicitly quoted). The function so defined may be an ordinary Maxima function (with arguments enclosed in parentheses) or 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 `define` accepts a variable number of arguments. Actual arguments are assigned one-to-one to formal arguments *x_1*, ..., *x_(n - 1)*, and any further actual arguments, if present, are assigned to *x_n* as a list.

When the first argument of `define` is an expression of the form *f*`(`*x_1*`, ..., `*x_n*`)` or *f*`[`*x_1*`, ..., `*x_n*`]`, the function arguments are evaluated but *f* is not evaluated, even if there is already a function or variable by that name.

When the first argument is an expression with operator `funmake`, `arraymake`, or `ev`, the first argument is evaluated; this allows for the function name to be computed, as well as the body.

All function definitions appear in the same namespace; defining a function `f` within another function `g` does not limit the scope of `f` to `g`.

If some formal argument *x_k* is a quoted symbol (after evaluation), the function defined by `define` does not evaluate the corresponding actual argument. Otherwise all actual arguments are evaluated.

See also `:=` and `::=`.

Examples:

`define` always evaluates its second argument (unless explicitly quoted).

```
(%i1) expr : cos(y) - sin(x);
(%o1)                      cos(y) - sin(x)
(%i2) define (F1 (x, y), expr);
(%o2)               F1(x, y) := cos(y) - sin(x)
(%i3) F1 (a, b);
(%o3)                      cos(b) - sin(a)
(%i4) F2 (x, y) := expr;
(%o4)                     F2(x, y) := expr
(%i5) F2 (a, b);
(%o5)                      cos(y) - sin(x)
```

The function defined by `define` may be an ordinary Maxima function or 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 `define` accepts a variable number of arguments.

```
(%i1) define (H ([L]), '(apply ("+", L)));
(%o1)                H([L]) := apply("+", L)
(%i2) H (a, b, c);
(%o2)                        c + b + a
```

When the first argument is an expression with operator `funmake`, `arraymake`, or `ev`, the first argument is evaluated.

```
(%i1) [F : I, u : x];
(%o1)                          [I, x]
(%i2) funmake (F, [u]);
(%o2)                           I(x)
(%i3) define (funmake (F, [u]), cos(u) + 1);
(%o3)                   I(x) := cos(x) + 1
(%i4) define (arraymake (F, [u]), cos(u) + 1);
(%o4)                    I  := cos(x) + 1
                          x
(%i5) define (foo (x, y), bar (y, x));
(%o5)                foo(x, y) := bar(y, x)
(%i6) define (ev (foo (x, y)), sin(x) - cos(y));
(%o6)             bar(y, x) := sin(x) - cos(y)
```

define_variable (*name, default_value, mode*) Function

Introduces a global variable into the Maxima environment. `define_variable` is useful in user-written packages, which are often translated or compiled.

`define_variable` carries out the following steps:

1. `mode_declare (`*name*`, `*mode*`)` declares the mode of *name* to the translator. See `mode_declare` for a list of the possible modes.
2. If the variable is unbound, *default_value* is assigned to *name*.
3. `declare (`*name*`, special)` declares it special.
4. Associates *name* with a test function to ensure that *name* is only assigned values of the declared mode.

The `value_check` property can be assigned to any variable which has been defined via `define_variable` with a mode other than `any`. The `value_check` property is a lambda expression or the name of a function of one variable, which is called when an attempt is made to assign a value to the variable. The argument of the `value_check` function is the would-be assigned value.

`define_variable` evaluates `default_value`, and quotes `name` and `mode`. `define_variable` returns the current value of `name`, which is `default_value` if `name` was unbound before, and otherwise it is the previous value of `name`.

Examples:

`foo` is a Boolean variable, with the initial value `true`.

```
(%i1) define_variable (foo, true, boolean);
(%o1)                          true
(%i2) foo;
(%o2)                          true
(%i3) foo: false;
(%o3)                          false
(%i4) foo: %pi;
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 # 'baz_quux then
                 error("Cannot assign to `baz_quux'."))(y=sqrt(2))
```

```
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i6) baz_quux;
(%o6)                         baz_quux
```

dispfun (*f_1*, ..., *f_n*) Function
dispfun (*all*) Function

Displays the definition of the user-defined functions *f_1*, ..., *f_n*. Each argument may be the name of a macro (defined with `::=`), an ordinary function (defined with `:=` or `define`), an array function (defined with `:=` or `define`, but enclosing arguments in square brackets `[ ]`), a subscripted function, (defined with `:=` or `define`, but enclosing some arguments in square brackets and others in parentheses `( )`) one of a family of subscripted functions selected by a particular subscript value, or a subscripted function defined with a constant subscript.

`dispfun (all)` displays all user-defined functions as given by the `functions`, `arrays`, and `macros` lists, omitting subscripted functions defined with constant subscripts.

`dispfun` creates an intermediate expression label (`%t1`, `%t2`, etc.) for each displayed function, and assigns the function definition to the label. In contrast, `fundef` returns the function definition.

`dispfun` quotes its arguments; the quote-quote operator `''` defeats quotation. `dispfun` returns the list of intermediate expression labels corresponding to the displayed functions.

Examples:

```
(%i1) m(x, y) ::= x^(-y);
                                        - y
(%o1)                      m(x, y) ::= x
(%i2) f(x, y) :=  x^(-y);
                                         - y
(%o2)                       f(x, y) := x
(%i3) g[x, y] :=  x^(-y);
                                        - y
(%o3)                        g     := x
                              x, y
(%i4) h[x](y) :=  x^(-y);
                                        - y
(%o4)                       h (y) := x
                             x
(%i5) i[8](y) :=  8^(-y);
                                        - y
(%o5)                       i (y) := 8
                             8
(%i6) dispfun (m, f, g, h, h[5], h[10], i[8]);
                                         - y
(%t6)                     m(x, y) ::= x

                                         - y
(%t7)                      f(x, y) := x
```

```
                                          - y
(%t8)                        g     := x
                              x, y

                                          - y
(%t9)                        h (y) := x
                              x

                                         1
(%t10)                        h (y) := --
                               5        y
                                        5

                                          1
(%t11)                       h  (y) := ---
                              10         y
                                        10

                                          - y
(%t12)                       i (y) := 8
                              8

(%o12)       [%t6, %t7, %t8, %t9, %t10, %t11, %t12]
(%i12) ''%;
                       - y              - y            - y
(%o12) [m(x, y) ::= x   , f(x, y) := x   , g     := x   ,
                                                 x, y
                    - y           1             1             - y
        h (y) := x   , h (y) := --, h  (y) := ---, i (y) := 8   ]
         x             5        y   10         y   8
                                5             10
```

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;
                                          x
```

```
(%o2)                    F_2(x, y) := -
                                      y
(%i3) define (F_3 (x), sqrt (x));
(%o3)                   F_3(x) := sqrt(x)
(%i4) G_1 [x] := x - 100;
(%o4)                    G_1  := x - 100
                            x
(%i5) G_2 [x, y] := x / y;
                                     x
(%o5)                     G_2     := -
                             x, y    y
(%i6) define (G_3 [x], sqrt (x));
(%o6)                    G_3  := sqrt(x)
                            x
(%i7) H_1 [x] (y) := x^y;
                                      y
(%o7)                     H_1 (y) := x
                             x
(%i8) functions;
(%o8)              [F_1(x), F_2(x, y), F_3(x)]
(%i9) arrays;
(%o9)                 [G_1, G_2, G_3, H_1]
```

fundef (*f*) Function

Returns the definition of the function *f*.

The argument may be the name of a macro (defined with `::=`), an ordinary function (defined with `:=` or `define`), an array function (defined with `:=` or `define`, but enclosing arguments in square brackets `[ ]`), a subscripted function, (defined with `:=` or `define`, but enclosing some arguments in square brackets and others in parentheses `( )`) one of a family of subscripted functions selected by a particular subscript value, or a subscripted function defined with a constant subscript.

`fundef` quotes its argument; the quote-quote operator `''` defeats quotation.

`fundef (`*f*`)` returns the definition of *f*. In contrast, `dispfun (`*f*`)` creates an intermediate expression label and assigns the definition to the label.

funmake (*F*, [*arg_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.

`funmake` does not attempt to distinguish array functions from ordinary functions; when *F* is the name of an array function, `funmake` returns *F*`(...)` (that is, a function call with parentheses instead of square brackets). `arraymake` returns a function call with square brackets in this case.

`funmake` evaluates its arguments.

Examples:

`funmake` applied to an ordinary Maxima function.

```
(%i1) F (x, y) := y^2 - x^2;
                                        2    2
(%o1)                    F(x, y) := y  - x
(%i2) funmake (F, [a + 1, b + 1]);
(%o2)                      F(a + 1, b + 1)
(%i3) ''%;
                                2           2
(%o3)                    (b + 1)  - (a + 1)
```

`funmake` applied to a macro.

```
(%i1) G (x) ::= (x - 1)/2;
                                      x - 1
(%o1)                      G(x) ::= -----
                                        2
(%i2) funmake (G, [u]);
(%o2)                            G(u)
(%i3) ''%;
                                 u - 1
(%o3)                            -----
                                   2
```

`funmake` applied to a subscripted function.

```
(%i1) H [a] (x) := (x - 1)^a;
                                                a
(%o1)                     H (x) := (x - 1)
                           a
(%i2) funmake (H [n], [%e]);
                                                n
(%o2)                lambda([x], (x - 1) )(%e)
(%i3) ''%;
                                              n
(%o3)                           (%e - 1)
(%i4) funmake ('(H [n]), [%e]);
(%o4)                            H (%e)
                                  n
(%i5) ''%;
                                              n
(%o5)                           (%e - 1)
```

`funmake` applied to a symbol which is not a defined function of any kind.

```
(%i1) funmake (A, [u]);
(%o1)                            A(u)
(%i2) ''%;
(%o2)                            A(u)
```

`funmake` evaluates its arguments, but not the return value.

```
(%i1) det(a,b,c) := b^2 -4*a*c;
                                              2
(%o1)                 det(a, b, c) := b  - 4 a c
(%i2) (x : 8, y : 10, z : 12);
(%o2)                             12
```

```
(%i3) f : det;
(%o3)                           det
(%i4) funmake (f, [x, y, z]);
(%o4)                     det(8, 10, 12)
(%i5) ''%;
(%o5)                          - 284
```

Maxima simplifies `funmake`'s return value.

```
(%i1) funmake (sin, [%pi / 2]);
(%o1)                            1
```

lambda ([*x_1*, ..., *x_m*], *expr_1*, ..., *expr_n*) Function
lambda ([[*L*]], *expr_1*, ..., *expr_n*) Function
lambda ([*x_1*, ..., *x_m*, [*L*]], *expr_1*, ..., *expr_n*) Function

Defines and returns a lambda expression (that is, an anonymous function). The function may have required arguments *x_1*, ..., *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*, ..., *x_m* are created. `lambda` may appear within `block` or another `lambda`; local variables are established each time another `block` or `lambda` is evaluated. Local variables appear to be global to any enclosed `block` or `lambda`. If a variable is not local, its value is the value most recently assigned in an enclosing `block` or `lambda`, if any, otherwise, it is the value of the variable in the global environment. This policy may coincide with the usual understanding of "dynamic scope".

After local variables are established, *expr_1* through *expr_n* are evaluated in turn. The special variable `%%`, representing the value of the preceding expression, is recognized. `throw` and `catch` may also appear in the list of expressions.

`return` cannot appear in a lambda expression unless enclosed by `block`, in which case `return` defines the return value of the block and not of the lambda expression, unless the block happens to be *expr_n*. Likewise, `go` cannot appear in a lambda expression unless enclosed by `block`.

`lambda` quotes its arguments; the quote-quote operator `''` defeats quotation.

Examples:

- A lambda expression can be assigned to a variable and evaluated like an ordinary function.

```
(%i1) f: lambda ([x], x^2);
                                        2
(%o1)                     lambda([x], x )
(%i2) f(a);
                                      2
(%o2)                                 a
```

- A lambda expression may appear in contexts in which a function evaluation is expected.

```
(%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);
                              %gamma
(%o10)                        ------
                                2
(%i11) g2: lambda ([a], a*''b);
(%o11)                 lambda([a], a %gamma)
(%i12) b: %e$
(%i13) g2(1/2);
                              %gamma
(%o13)                        ------
                                2
```

- Lambda 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.

```
(%i14) h: lambda ([a, b], h2: lambda ([a], a*b), h2(1/2));
                                                      1
(%o14)    lambda([a, b], h2 : lambda([a], a b), h2(-))
                                                      2
(%i15) h(%pi, %gamma);
                              %gamma
(%o15)                        ------
                                2
```

- 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.

```
(%i16) i: lambda ([a], lambda ([x], a*x));
(%o16)             lambda([a], lambda([x], a x))
(%i17) i(1/2);
(%o17)                    lambda([x], a x)
(%i18) i2: lambda([a], buildq([a: a], lambda([x], a*x)));
(%o18)    lambda([a], buildq([a : a], lambda([x], a x)))
```

```
(%i19) i2(1/2);
                                        x
(%o19)                    lambda([x], -)
                                        2
(%i20) i2(1/2)(%pi);
                                  %pi
(%o20)                            ---
                                   2
```

- A lambda expression may take a variable number of arguments, which are indicated by `[`*L*`]` as the sole or final argument. The arguments appear within the function body as a list.

```
(%i1) f : lambda ([aa, bb, [cc]], aa * cc + bb);
(%o1)          lambda([aa, bb, [cc]], aa cc + bb)
(%i2) f (foo, %i, 17, 29, 256);
(%o2)       [17 foo + %i, 29 foo + %i, 256 foo + %i]
(%i3) g : lambda ([[aa]], apply ("+", aa));
(%o3)             lambda([[aa]], apply(+, aa))
(%i4) g (17, 29, x, y, z, %e);
(%o4)                   z + y + x + %e + 46
```

local (*v_1*, ..., *v_n*) Function

Declares the variables *v_1*, ..., *v_n* to be local with respect to all the properties in the statement in which this function is used.

`local` quotes its arguments. `local` returns `done`.

`local` may only be used in `block`, in the body of function definitions or `lambda` expressions, or in the `ev` function, and only one occurrence is permitted in each.

`local` is independent of `context`.

macroexpansion Option variable

Default value: `false`

`macroexpansion` controls whether the expansion (that is, the return value) of a macro function is substituted for the macro function call. A substitution may speed up subsequent expression evaluations, at the cost of storing the expansion.

`false` The expansion of a macro function is not substituted for the macro function call.

`expand` The first time a macro function call is evaluated, the expansion is stored. The expansion is not recomputed on subsequent calls; any side effects (such as `print` or assignment to global variables) happen only when the macro function call is first evaluated. Expansion in an expression does not affect other expressions which have the same macro function call.

`displace` The first time a macro function call is evaluated, the expansion is substituted for the call, thus modifying the expression from which the macro function was called. The expansion is not recomputed on subsequent calls; any side effects happen only when the macro function call is first evaluated. Expansion in an expression does not affect other expressions which have the same macro function call.

Examples

When `macroexpansion` is `false`, a macro function is called every time the calling expression is evaluated, and the calling expression is not modified.

```
(%i1) f (x) := h (x) / g (x);
                                    h(x)
(%o1)                      f(x) := ----
                                    g(x)
(%i2) g (x) ::= block (print ("x + 99 is equal to", x),
                       return (x + 99));
(%o2) g(x) ::= block(print("x + 99 is equal to", x),
                                                  return(x + 99))
(%i3) h (x) ::= block (print ("x - 99 is equal to", x),
                       return (x - 99));
(%o3) h(x) ::= block(print("x - 99 is equal to", x),
                                                  return(x - 99))
(%i4) macroexpansion: false;
(%o4)                          false
(%i5) f (a * b);
x - 99 is equal to x
x + 99 is equal to x
                             a b - 99
(%o5)                        --------
                             a b + 99
(%i6) dispfun (f);
                                    h(x)
(%t6)                      f(x) := ----
                                    g(x)

(%o6)                          done
(%i7) f (a * b);
x - 99 is equal to x
x + 99 is equal to x
                             a b - 99
(%o7)                        --------
                             a b + 99
```

When `macroexpansion` is `expand`, a macro function is called once, 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: expand;
(%o4)                          expand
(%i5) f (a * b);
x - 99 is equal to x
x + 99 is equal to x
                               a b - 99
(%o5)                          --------
                               a b + 99
(%i6) dispfun (f);
                                    h(x)
(%t6)                       f(x) := ----
                                    g(x)

(%o6)                            done
(%i7) f (a * b);
                               a b - 99
(%o7)                          --------
                               a b + 99
```

When `macroexpansion` is `expand`, a macro function is called once, and the calling expression is 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: displace;
(%o4)                          displace
(%i5) f (a * b);
x - 99 is equal to x
x + 99 is equal to x
                               a b - 99
(%o5)                          --------
                               a b + 99
(%i6) dispfun (f);
                                   x - 99
(%t6)                       f(x) := ------
                                   x + 99

(%o6)                            done
(%i7) f (a * b);
                               a b - 99
```

```
(%o7)                          --------
                               a b + 99
```

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

```
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:

```
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,

```
mode_declare ([function (f_1, f_2, ...)], fixnum)
```

declares that the values returned by `f_1`, `f_2`, ... are single-word integers.

`modedeclare` is a synonym for `mode_declare`.

mode_identity (*arg_1, arg_2*) Function

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., `x: 3.3; mode_identity (fixnum, x);` yields an error. `mode_identity (flonum, x)` returns 3.3 . This has a number of uses, e.g., if you knew that `first (l)` returned a number then you might write `mode_identity (number, first (l))`. However, a more efficient way to do it would be to define a new primitive,

```
firstnumb (x) ::= buildq ([x], mode_identity (number, x));
```

and use `firstnumb` every time you take the first of a list of numbers.

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.

`compfile` sets `transcompile: true` for the duration.

translate (*f_1, ..., f_n*) Function
translate (*functions*) Function
translate (*all*) Function

Translates the user-defined functions *f_1*, ..., *f_n* from the Maxima language into Lisp and evaluates the Lisp translations. Typically the translated functions run faster than the originals.

`translate (all)` or `translate (functions)` translates all user-defined functions.

Functions to be translated should include a call to `mode_declare` at the beginning when possible in order to produce more efficient code. For example:

```
f (x_1, x_2, ...) := block ([v_1, v_2, ...],
    mode_declare (v_1, mode_1, v_2, mode_2, ...), ...)
```

where the *x_1*, *x_2*, ... are the parameters to the function and the *v_1*, *v_2*, ... 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 incompatabilities 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 `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_messagesp Option variable

Default value: `false`

When `tr_file_tty_messagesp` is `true`, messages generated by `translate_file` during translation of a file are displayed on the console and inserted into the UNLISP file. When `false`, messages about translation of the file are only inserted into the UNLISP file.

tr_float_can_branch_complex Option variable

Default value: `true`

Tells the Maxima-to-Lisp translator to assume that the functions `acos`, `asin`, `asec`, and `acsc` can return complex results.

The ostensible effect of `tr_float_can_branch_complex` is the following. However, it appears that this flag has no effect on the translator output.

When it is `true` then `acos(x)` is of mode `any` even if `x` is of mode `float` (as set by `mode_declare`). When `false` then `acos(x)` is of mode `float` if and only if `x` is of mode `float`.

tr_function_call_default Option variable

Default value: `general`

`false` means give up and call `meval`, `expr` means assume Lisp fixed arg function. `general`, the default gives code good for `mexprs` and `mlexprs` but not `macros`. `general` assures variable bindings are correct in compiled code. In `general` mode, when translating F(X), if F is a bound variable, then it assumes that `apply (f, [x])`

is meant, and translates a such, with 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 Option variable

Default value: `false`

When `tr_numer` is `true`, `numer` properties are used for atoms which have them, e.g. `%pi`.

tr_optimize_max_loop Option variable

Default value: 100

`tr_optimize_max_loop` is the maximum number of times the macro-expansion and optimization pass of the translator will loop in considering a form. This is to catch macro expansion errors, and non-terminating optimization properties.

tr_semicompile Option variable

Default value: `false`

When `tr_semicompile` is `true`, `translate_file` and `compfile` output forms which will be macroexpanded but not compiled into machine code by the Lisp compiler.

tr_state_vars System variable

Default value:

```
[transcompile, tr_semicompile, tr_warn_undeclared, tr_warn_meval,
tr_warn_fexpr, tr_warn_mode, tr_warn_undefined_variable,
tr_function_call_default, tr_array_as_ref,tr_numer]
```

The list of the switches that affect the form of the translated output. This information is useful to system people when trying to debug the translator. By comparing the translated product to what should have been produced for a given state, it is possible to track down bugs.

tr_warnings_get () Function

Prints a list of warnings which have been given by the translator during the current translation.

tr_warn_bad_function_calls Option variable

Default value: `true`

- Gives a warning when when function calls are being made which may not be correct due to improper declarations that were made at translate time.

tr_warn_fexpr Option variable

Default value: `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.

compile_file (*filename*) Function
compile_file (*filename, compiled_filename*) Function
compile_file (*filename, compiled_filename, lisp_filename*) Function

Translates the Maxima file *filename* into Lisp, executes the Lisp compiler, and, if the translation and compilation succeed, loads the compiled code into Maxima.

`compile_file` returns a list of the names of four files: the original Maxima file, the Lisp translation, notes on translation, and the compiled code. If the compilation fails, the fourth item is `false`.

Some declarations and definitions take effect as soon as the Lisp code is compiled (without loading the compiled code). These include functions defined with the `:=` operator, macros define with the `::=` operator, `alias`, `declare`, `define_variable`, `mode_declare`, and `infix`, `matchfix`, `nofix`, `postfix`, `prefix`, and `compfile`.

Assignments and function calls are not evaluated until the compiled code is loaded. In particular, within the Maxima file, assignments to the translation flags (`tr_numer`, etc.) have no effect on the translation.

filename may not contain `:lisp` statements.

`compile_file` evaluates its arguments.

declare_translated (*f_1, f_2, ...*) Function

When translating a file of Maxima code to Lisp, it is important for the translator to know which functions it sees in the file are to be called as translated or compiled functions, and which ones are just Maxima functions or undefined. Putting this declaration at the top of the file, lets it know that although a symbol does which does

not yet have a Lisp function value, will have one at call time. `(MFUNCTION-CALL fn arg1 arg2 ...)` is generated when the translator does not know `fn` is going to be a Lisp function.

40 Program Flow

40.1 Introduction to Program Flow

Maxima provides a `do` loop for iteration, as well as more primitive constructs such as `go`.

40.2 Functions and Variables for Program Flow

backtrace () Function
backtrace (*n*) Function

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 Special operator

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 `while i <= 10` is equivalent to `unless i > 10` and also `thru 10`.

```
(%i1) series: 1$
(%i2) term: exp (sin (x))$
(%i3) for p: 1 unless p > 7 do
          (term: diff (term, x)/p,
           series: series + subst (x=0, term)*x^p)$
(%i4) series;
                   7    6     5    4    2
                  x    x     x    x    x
(%o4)             -- - --- - -- - -- + -- + x + 1
                  90   240   15   8    2
```

which gives 8 terms of the Taylor series for `e^sin(x)`.

```
(%i1) poly: 0$
(%i2) for i: 1 thru 5 do
          for j: i step -1 thru 1 do
              poly: poly + i*x^j$
(%i3) poly;
                    5      4       3       2
(%o3)            5 x  + 9 x  + 12 x  + 14 x  + 15 x
(%i4) guess: -3.0$
(%i5) for i: 1 thru 10 do
          (guess: subst (guess, x, 0.5*(x + 10/x)),
           if abs (guess^2 - 10) < 0.00005 then return (guess));
(%o5)                  - 3.162280701754386
```

This example computes the negative square root of 10 using the Newton- Raphson iteration a maximum of 10 times. Had the convergence criterion not been met the value returned would have been `done`.

Instead of always adding a quantity to the control-variable one may sometimes wish to change it in some other way for each iteration. In this case one may use `next` *expression* instead of `step` *increment*. This will cause the control-variable to be set to the result of evaluating *expression* each time through the loop.

```
(%i6) for count: 2 next 3*count thru 20 do display (count)$
                              count = 2

                              count = 6
```

```
                                count = 18
```

As an alternative to `for` *variable*`:` *value* `...do...` the syntax `for` *variable* `from` *value* `...do...` may be used. This permits the `from` *value* to be placed after the `step` or `next` value or after the termination condition. If `from` *value* is omitted then 1 is used as the initial value.

Sometimes one may be interested in performing an iteration where the control-variable is never actually used. It is thus permissible to give only the termination conditions omitting the initialization and updating information as in the following example to compute the square-root of 5 using a poor initial guess.

```
(%i1) x: 1000$
(%i2) thru 20 do x: 0.5*(x + 5.0/x)$
(%i3) x;
(%o3)                    2.23606797749979
(%i4) sqrt(5), numer;
(%o4)                    2.23606797749979
```

If it is desired one may even omit the termination conditions entirely and just give `do` *body* which will continue to evaluate the *body* indefinitely. In this case the function `return` should be used to terminate execution of the `do`.

```
(%i1) newton (f, x):= ([y, df, dfx], df: diff (f ('x), 'x),
          do (y: ev(df), x: x - f(x)/y,
              if abs (f (x)) < 5e-6 then return (x)))$
(%i2) sqr (x) := x^2 - 5.0$
(%i3) newton (sqr, 1000);
(%o3)                    2.236068027062195
```

(Note that `return`, when executed, causes the current value of `x` to be returned as the value of the `do`. The `block` is exited and this value of the `do` is returned as the value of the `block` because the `do` is the last statement in the block.)

One other form of the `do` is available in Maxima. The syntax is:

`for` *variable* `in` *list* *end_tests* `do` *body*

The elements of *list* are any expressions which will successively be assigned to the `variable` on each iteration of the *body*. The optional termination tests *end_tests* can be used to terminate execution of the `do`; otherwise it will terminate when the *list* is exhausted or when a `return` is executed in the *body*. (In fact, `list` may be any non-atomic expression, and successive parts are taken.)

```
(%i1)  for f in [log, rho, atan] do ldisp(f(1))$
(%t1)                                  0
(%t2)                               rho(1)
                                    %pi
(%t3)                               ---
                                    4
(%i4) ev(%t3,numer);
(%o4)                          0.78539816
```

errcatch (*expr_1, ..., expr_n*) Function

Evaluates *expr_1*, ..., *expr_n* one by one and returns `[`*expr_n*`]` (a list) if no error occurs. If an error occurs in the evaluation of any argument, `errcatch` prevents the error from propagating and returns the empty list `[]` without evaluating any more arguments.

`errcatch` is useful in `batch` files where one suspects an error might occur which would terminate the `batch` if the error weren't caught.

error (*expr_1, ..., expr_n*) Function

error System variable

Evaluates and prints *expr_1*, ..., *expr_n*, and then causes an error return to top level Maxima or to the nearest enclosing `errcatch`.

The variable `error` is set to a list describing the error. The first element of `error` is a format string, which merges all the strings among the arguments *expr_1*, ..., *expr_n*, and the remaining elements are the values of any non-string arguments.

`errormsg()` formats and prints `error`. This is effectively reprinting the most recent error message.

errormsg () Function

Reprints the most recent error message. The variable `error` holds the message, and `errormsg` formats and prints it.

for Special operator

Used in iterations. See `do` for a description of Maxima's iteration facilities.

go (*tag*) Function

is used within a `block` to transfer control to the statement of the block which is tagged with the argument to `go`. To tag a statement, precede it by an atomic argument as another statement in the `block`. For example:

```
block ([x], x:1, loop, x+1, ..., go(loop), ...)
```

The argument to `go` must be the name of a tag appearing in the same `block`. One cannot use `go` to transfer to tag in a `block` other than the one containing the `go`.

if Special operator

Represents conditional evaluation. Various forms of `if` expressions are recognized.

`if` *cond_1* `then` *expr_1* `else` *expr_0* evaluates to *expr_1* if *cond_1* evaluates to `true`, otherwise the expression evaluates to *expr_0*.

`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.

Operation	Symbol	Type
less than	<	relational infix
less than or equal to	<=	relational infix
equality (syntactic)	=	relational infix
negation of =	#	relational infix
equality (value)	equal	relational function
negation of equal	notequal	relational function
greater than or equal to	>=	relational infix
greater than	>	relational infix
and	and	logical infix
or	or	logical infix
not	not	logical prefix

map (*f, expr_1, ..., expr_n*) Function

Returns an expression whose leading operator is the same as that of the expressions *expr_1*, ..., *expr_n* but whose subparts are the results of applying *f* to the corresponding subparts of the expressions. *f* is either the name of a function of *n* arguments or is a `lambda` form of *n* arguments.

`maperror` - if `false` will cause all of the mapping functions to (1) stop when they finish going down the shortest *expr_i* if not all of the *expr_i* are of the same length and (2) apply *f* to [*expr_1*, *expr_2*, ...] if the *expr_i* are not all the same type of object. If `maperror` is `true` then an error message will be given in the above two instances.

One of the uses of this function is to `map` a function (e.g. `partfrac`) onto each term of a very large expression where it ordinarily wouldn't be possible to use the function on the entire expression due to an exhaustion of list storage space in the course of the computation.

```
(%i1) map(f,x+a*y+b*z);
(%o1)                        f(b z) + f(a y) + f(x)
(%i2) map(lambda([u],partfrac(u,x)),x+1/(x^3+4*x^2+5*x+2));
                             1       1        1
(%o2)                      ----- - ----- + -------- + x
                           x + 2   x + 1          2
                                           (x + 1)
```

```
(%i3) map(ratsimp, x/(x^2+x)+(y^2+y)/y);
                                        1
(%o3)                             y + ----- + 1
                                      x + 1
(%i4) map("=",[a,b],[-0.5,3]);
(%o4)                          [a = - 0.5, b = 3]
```

mapatom (*expr*) Function

Returns `true` if and only if *expr* is treated by the mapping routines as an atom. "Mapatoms" are atoms, numbers (including rational numbers), and subscripted variables.

maperror Option variable

Default value: `true`

When `maperror` is `false`, causes all of the mapping functions, for example

`map (`*f*`,` *expr_1*`,` *expr_2*`, ...)`

to (1) stop when they finish going down the shortest *expr_i* if not all of the *expr_i* are of the same length and (2) apply *f* to [*expr_1*, *expr_2*, ...] if the *expr_i* are not all the same type of object.

If `maperror` is `true` then an error message is displayed in the above two instances.

mapprint Option variable

Default value: `true`

When `mapprint` is `true`, various information messages from `map`, `mapl`, and `fullmap` are produced in certain situations. These include situations where `map` would use `apply`, or `map` is truncating on the shortest list.

If `mapprint` is `false`, these messages are suppressed.

maplist (*f, expr_1, ..., expr_n*) Function

Returns a list of the applications of *f* to the parts of the expressions *expr_1*, ..., *expr_n*. *f* is the name of a function, or a lambda expression.

`maplist` differs from `map (`*f*`,` *expr_1*`, ...,` *expr_n*`)` which returns an expression with the same main operator as *expr_i* has (except for simplifications and the case where `map` does an `apply`).

prederror Option variable

Default value: `false`

When `prederror` is `true`, an error message is displayed whenever the predicate of an `if` statement or an `is` function fails to evaluate to either `true` or `false`.

If `false`, `unknown` is returned instead in this case. The `prederror: false` mode is not supported in translated code; however, `maybe` is supported in translated code.

See also `is` and `maybe`.

return (*value*) Function

May be used to exit explicitly from a block, bringing its argument. See `block` for more information.

scanmap (*f, expr*) Function
scanmap (*f, expr, bottomup*) Function

Recursively applies *f* to *expr*, in a top down manner. This is most useful when complete factorization is desired, for example:

```
(%i1) exp:(a^2+2*a+1)*y + x^2$
(%i2) scanmap(factor,exp);
                                      2      2
(%o2)                          (a + 1)  y + x
```

Note the way in which `scanmap` applies the given function `factor` to the constituent subexpressions of *expr*; if another form of *expr* is presented to `scanmap` then the result may be different. Thus, `%o2` is not recovered when `scanmap` is applied to the expanded form of `exp`:

```
(%i3) scanmap(factor,expand(exp));
                            2                 2
(%o3)                       a  y + 2 a y + y + x
```

Here is another example of the way in which `scanmap` recursively applies a given function to all subexpressions, including exponents:

```
(%i4) expr : u*v^(a*x+b) + c$
(%i5) scanmap('f, expr);
                     f(f(f(a) f(x)) + f(b))
(%o5) f(f(f(u) f(f(v                       )) + f(c))
```

`scanmap` (*f*, *expr*, `bottomup`) applies *f* to *expr* in a bottom-up manner. E.g., for undefined `f`,

```
scanmap(f,a*x+b) ->
   f(a*x+b) -> f(f(a*x)+f(b)) -> f(f(f(a)*f(x))+f(b))
scanmap(f,a*x+b,bottomup) -> f(a)*f(x)+f(b)
    -> f(f(a)*f(x))+f(b) ->
     f(f(f(a)*f(x))+f(b))
```

In this case, you get the same answer both ways.

throw (*expr*) Function

Evaluates *expr* and throws the value back to the most recent `catch`. `throw` is used with `catch` as a nonlocal return mechanism.

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 `map`, `maplist`, and `apply`.

Examples:

Elementary examples of `outermap`. To show the argument combinations more clearly, `F` is left undefined.

```
(%i1) outermap(F, [a, b, c], [1, 2, 3]);
(%o1) [[F(a, 1), F(a, 2), F(a, 3)], [F(b, 1), F(b, 2), F(b, 3)],
                                     [F(c, 1), F(c, 2), F(c, 3)]]
(%i2) outermap(F, matrix([a, b],[c, d]), matrix([1, 2],[3, 4]));
         [ [ F(a, 1)  F(a, 2) ]  [ F(b, 1)  F(b, 2) ] ]
         [ [                  ]  [                  ] ]
         [ [ F(a, 3)  F(a, 4) ]  [ F(b, 3)  F(b, 4) ] ]
(%o2)    [                                           ]
         [ [ F(c, 1)  F(c, 2) ]  [ F(d, 1)  F(d, 2) ] ]
         [ [                  ]  [                  ] ]
         [ [ F(c, 3)  F(c, 4) ]  [ F(d, 3)  F(d, 4) ] ]
(%i3) outermap (F, [a, b], x, matrix ([1, 2], [3, 4]));
       [ F(a, x, 1)  F(a, x, 2) ]  [ F(b, x, 1)  F(b, x, 2) ]
(%o3) [[                        ], [                        ]]
       [ F(a, x, 3)  F(a, x, 4) ]  [ F(b, x, 3)  F(b, x, 4) ]
(%i4) outermap (F, [a, b], matrix ([1, 2]), matrix ([x], [y]));
       [ [ F(a, 1, x) ]  [ F(a, 2, x) ] ]
(%o4) [[ [            ]  [            ] ],
       [ [ F(a, 1, y) ]  [ F(a, 2, y) ] ]
                              [ [ F(b, 1, x) ]  [ F(b, 2, x) ] ]
                              [ [            ]  [            ] ]]
                              [ [ F(b, 1, y) ]  [ F(b, 2, y) ] ]
(%i5) outermap ("+", [a, b, c], [1, 2, 3]);
(%o5) [[a + 1, a + 2, a + 3], [b + 1, b + 2, b + 3],
                                          [c + 1, c + 2, c + 3]]
```

A closer examination of the `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.

```
(%i1) arg_1 :  matrix ([a, b], [c, d]);
                              [ a  b ]
(%o1)                         [      ]
                              [ c  d ]
(%i2) arg_2 : [11, 22];
(%o2)                         [11, 22]
(%i3) arg_3 : matrix ([xx, yy]);
(%o3)                        [ xx  yy ]
(%i4) xx_0 : outermap(lambda([x, y, z], x / y + z), arg_1,
                                                  arg_2, arg_3);
                [  [      a        a  ]  [      a        a  ]  ]
                [ [[ xx + --  yy + -- ], [ xx + --  yy + -- ]] ]
                [  [      11       11 ]  [      22       22 ]  ]
(%o4)  Col 1 = [                                               ]
                [  [      c        c  ]  [      c        c  ]  ]
                [ [[ xx + --  yy + -- ], [ xx + --  yy + -- ]] ]
                [  [      11       11 ]  [      22       22 ]  ]
                  [  [      b        b  ]  [      b        b  ]  ]
                  [ [[ xx + --  yy + -- ], [ xx + --  yy + -- ]] ]
                  [  [      11       11 ]  [      22       22 ]  ]
         Col 2 = [                                               ]
                  [  [      d        d  ]  [      d        d  ]  ]
                  [ [[ xx + --  yy + -- ], [ xx + --  yy + -- ]] ]
                  [  [      11       11 ]  [      22       22 ]  ]
(%i5) xx_1 : xx_0 [1][1];
              [      a        a  ]  [      a        a  ]
(%o5)      [[ xx + --  yy + -- ], [ xx + --  yy + -- ]]
              [      11       11 ]  [      22       22 ]
(%i6) xx_2 : xx_0 [1][1] [1];
                        [      a        a  ]
(%o6)                   [ xx + --  yy + -- ]
                        [      11       11 ]
(%i7) xx_3 : xx_0 [1][1] [1] [1][1];
                                 a
(%o7)                         xx + --
                                 11
(%i8) [op (arg_1), op (arg_2), op (arg_3)];
(%o8)                    [matrix, [, matrix]
(%i9) [op (xx_0), op (xx_1), op (xx_2)];
(%o9)                    [matrix, [, matrix]
```

`outermap` 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
```

41 Debugging

41.1 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.

```
(%i1) load ("/tmp/foobar.mac");

(%o1)                           /tmp/foobar.mac

(%i2) :br foo
Turning on debugging debugmode(true)
Bkpt 0 for foo (in /tmp/foobar.mac line 1)

(%i2) bar (2,3);
Bkpt 0:(foobar.mac 1)
/tmp/foobar.mac:1::

(dbm:1) :bt                          <-- :bt typed here gives a backtrace
#0: foo(y=5)(foobar.mac line 1)
#1: bar(x=2,y=3)(foobar.mac line 9)

(dbm:1) :n                           <-- Here type :n to advance line
(foobar.mac 2)
/tmp/foobar.mac:2::

(dbm:1) :n                           <-- Here type :n to advance line
(foobar.mac 3)
/tmp/foobar.mac:3::

(dbm:1) u;                           <-- Investigate value of u
28

(dbm:1) u: 33;                       <-- Change u to be 33
33

(dbm:1) :r                           <-- Type :r to resume the computation
```

```
(%o2)                                1094
```

The file `/tmp/foobar.mac` is the following:

```
foo(y) := block ([u:y^2],
  u: u+3,
  u: u^2,
  u);

bar(x,y) := (
  x: x+2,
  y: y+2,
  x: foo(y),
  x+y);
```

USE OF THE DEBUGGER THROUGH EMACS

If the user is running the code under GNU emacs in a shell window (dbl shell), or is running the graphical interface version, Xmaxima, then if he stops at a break point, he will see his current position in the source file which will be displayed in the other half of the window, either highlighted in red, or with a little arrow pointing at the right line. He can advance single lines at a time by typing M-n (Alt-n).

Under Emacs you should run in a `dbl` shell, which requires the `dbl.el` file in the elisp directory. Make sure you install the elisp files or add the Maxima elisp directory to your path: e.g., add the following to your '`.emacs`' file or the '`site-init.el`'

```
(setq load-path (cons "/usr/share/maxima/5.9.1/emacs" load-path))
(autoload 'dbl "dbl")
```

then in emacs

```
M-x dbl
```

should start a shell window in which you can run programs, for example Maxima, gcl, gdb etc. This shell window also knows about source level debugging, and display of source code in the other window.

The user may set a break point at a certain line of the file by typing `C-x space`. This figures out which function the cursor is in, and then it sees which line of that function the cursor is on. If the cursor is on, say, line 2 of `foo`, then it will insert in the other window the command, "`:br foo 2`", to break `foo` at its second line. To have this enabled, the user must have maxima-mode.el turned on in the window in which the file `foobar.mac` is visiting. There are additional commands available in that file window, such as evaluating the function into the Maxima, by typing `Alt-Control-x`.

41.2 Keyword Commands

Keyword commands are special keywords which are not interpreted as Maxima expressions. A keyword command can be entered at the Maxima prompt or the debugger prompt, although not at the break prompt. Keyword commands start with a colon, '`:`'. For example, to evaluate a Lisp form you may type `:lisp` followed by the form to be evaluated.

```
(%i1) :lisp (+ 2 3)
5
```

The number of arguments taken depends on the particular command. Also, you need not type the whole command, just enough to be unique among the break keywords. Thus `:br` would suffice for `:break`.

The keyword commands are listed below.

`:break F n`
: Set a breakpoint in function `F` at line offset `n` from the beginning of the function. If `F` is given as a string, then it is assumed to be a file, and `n` is the offset from the beginning of the file. The offset is optional. If not given, it is assumed to be zero (first line of the function or file).

`:bt` Print a backtrace of the stack frames

`:continue`
: Continue the computation

`:delete` Delete the specified breakpoints, or all if none are specified

`:disable` Disable the specified breakpoints, or all if none are specified

`:enable` Enable the specified breakpoints, or all if none are specified

`:frame n` Print stack frame `n`, or the current frame if none is specified

`:help` Print help on a debugger command, or all commands if none is specified

`:info` Print information about item

`:lisp some-form`
: Evaluate `some-form` as a Lisp form

`:lisp-quiet some-form`
: Evaluate Lisp form `some-form` without any output

`:next` Like `:step`, except `:next` steps over function calls

`:quit` Quit the current debugger level without completing the computation

`:resume` Continue the computation

`:step` Continue the computation until it reaches a new source line

`:top` Return to the Maxima prompt (from any debugger level) without completing the computation

41.3 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.

setcheck Option variable

Default value: `false`

If `setcheck` is set to a list of variables (which can be subscripted), Maxima prints a message whenever the variables, or subscripted occurrences of them, are bound with the ordinary assignment operator `:`, the `::` assignment operator, or function argument binding, but not the function assignment `:=` nor the macro assignment `::=` operators. The message comprises the name of the variable and the value it is bound to.

`setcheck` may be set to `all` or `true` thereby including all variables.

Each new assignment of `setcheck` establishes a new list of variables to check, and any variables previously assigned to `setcheck` are forgotten.

The names assigned to `setcheck` must be quoted if they would otherwise evaluate to something other than themselves. For example, if `x`, `y`, and `z` are already bound, then enter

```
setcheck: ['x, 'y, 'z]$
```

to put them on the list of variables to check.

No printout is generated when a variable on the `setcheck` list is assigned to itself, e.g., `X: 'X`.

setcheckbreak Option variable

Default value: `false`

When `setcheckbreak` is `true`, Maxima will present a break prompt whenever a variable on the `setcheck` list is assigned a new value. The break occurs before the assignment is carried out. At this point, `setval` holds the value to which the variable is about to be assigned. Hence, one may assign a different value by assigning to `setval`.

See also `setcheck` and `setval`.

setval System variable

Holds the value to which a variable is about to be set when a `setcheckbreak` occurs. Hence, one may assign a different value by assigning to `setval`.

See also `setcheck` and `setcheckbreak`.

timer (*f_1*, ..., *f_n*) Function
timer (*all*) Function
timer () Function

Given functions *f_1*, ..., *f_n*, `timer` puts each one on the list of functions for which timing statistics are collected. `timer(f)$ timer(g)$` puts `f` and then `g` onto the list; the list accumulates from one call to the next.

`timer(all)` puts all user-defined functions (as named by the global variable `functions`) on the list of timed functions.

With no arguments, `timer` returns the list of timed functions.

Maxima records how much time is spent executing each function on the list of timed functions. `timer_info` returns the timing statistics, including the average time

elapsed per function call, the number of calls, and the total time elapsed. `untimer` removes functions from the list of timed functions.

`timer` quotes its arguments. `f(x) := x^2$ g:f$ timer(g)$` does not put `f` on the timer list.

If `trace(f)` is in effect, then `timer(f)` has no effect; `trace` and `timer` cannot both be in effect at the same time.

See also `timer_devalue`.

untimer (*f_1, ..., f_n*) Function
untimer () Function

Given functions *f_1*, ..., *f_n*, `untimer` removes each function from the timer list.

With no arguments, `untimer` removes all functions currently on the timer list.

After `untimer (f)` is executed, `timer_info (f)` still returns previously collected timing statistics, although `timer_info()` (with no arguments) does not return information about any function not currently on the timer list. `timer (f)` resets all timing statistics to zero and puts `f` on the timer list again.

timer_devalue Option variable

Default value: `false`

When `timer_devalue` is `true`, Maxima subtracts from each timed function the time spent in other timed functions. Otherwise, the time reported for each function includes the time spent in other functions. Note that time spent in untimed functions is not subtracted from the total time.

See also `timer` and `timer_info`.

timer_info (*f_1, ..., f_n*) Function
timer_info () Function

Given functions *f_1*, ..., *f_n*, `timer_info` returns a matrix containing timing information for each function. With no arguments, `timer_info` returns timing information for all functions currently on the timer list.

The matrix returned by `timer_info` contains the function name, time per function call, number of function calls, total time, and `gctime`, which meant "garbage collection time" in the original Macsyma but is now always zero.

The data from which `timer_info` constructs its return value can also be obtained by the `get` function:

```
get(f, 'calls);  get(f, 'runtime);  get(f, 'gctime);
```

See also `timer`.

trace (*f_1, ..., f_n*) Function
trace (*all*) Function
trace () Function

Given functions *f_1*, ..., *f_n*, `trace` instructs Maxima to print out debugging information whenever those functions are called. `trace(f)$ trace(g)$` puts `f` and then `g` onto the list of functions to be traced; the list accumulates from one call to the next.

`trace(all)` puts all user-defined functions (as named by the global variable `functions`) on the list of functions to be traced.

With no arguments, `trace` returns a list of all the functions currently being traced.

The `untrace` function disables tracing. See also `trace_options`.

`trace` quotes its arguments. Thus, `f(x) := x^2$ g:f$ trace(g)$` does not put `f` on the trace list.

When a function is redefined, it is removed from the timer list. Thus after `timer(f)$ f(x) := x^2$`, function `f` is no longer on the timer list.

If `timer (f)` is in effect, then `trace (f)` has no effect; `trace` and `timer` can't both be in effect for the same function.

trace_options (*f, option_1, ..., option_n*) Function
trace_options (*f*) Function

Sets the trace options for function *f*. Any previous options are superseded. `trace_options (f, ...)` has no effect unless `trace (f)` is also called (either before or after `trace_options`).

`trace_options (f)` resets all options to their default values.

The option keywords are:

- `noprint` Do not print a message at function entry and exit.
- `break` Put a breakpoint before the function is entered, and after the function is exited. See `break`.
- `lisp_print` Display arguments and return values as Lisp objects.
- `info` Print `-> true` at function entry and exit.
- `errorcatch` Catch errors, giving the option to signal an error, retry the function call, or specify a return value.

Trace options are specified in two forms. The presence of the option keyword alone puts the option into effect unconditionally. (Note that option *foo* is not put into effect by specifying *foo*`: true` or a similar form; note also that keywords need not be quoted.) Specifying the option keyword with a predicate function makes the option conditional on the predicate.

The argument list to the predicate function is always `[level, direction, function, item]` where `level` is the recursion level for the function, `direction` is either `enter` or `exit`, `function` is the name of the function, and `item` is the argument list (on entering) or the return value (on exiting).

Here is an example of unconditional trace options:

```
(%i1) ff(n) := if equal(n, 0) then 1 else n * ff(n - 1)$

(%i2) trace (ff)$

(%i3) trace_options (ff, lisp_print, break)$

(%i4) ff(3);
```

Here is the same function, with the `break` option conditional on a predicate:

```
(%i5) trace_options (ff, break(pp))$

(%i6) pp (level, direction, function, item) := block (print (item),
    return (function = 'ff and level = 3 and direction = exit))$

(%i7) ff(6);
```

untrace (*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.

42 augmented_lagrangian

42.1 Functions and Variables for augmented_lagrangian

augmented_lagrangian_method (*FOM, xx, C, yy*) Function
augmented_lagrangian_method (*FOM, xx, C, yy, optional_args*) Function

Returns an approximate minimum of the expression *FOM* with respect to the variables *xx*, holding the constraints *C* equal to zero. *yy* is a list of initial guesses for *xx*. The method employed is the augmented Lagrangian method (see Refs [1] and [2]).

`optional_args` represents additional arguments, specified as *symbol* = *value*. The optional arguments recognized are:

`niter` Number of iterations of the augmented Lagrangian algorithm

`lbfgs_tolerance`
Tolerance supplied to LBFGS

`iprint` IPRINT parameter (a list of two integers which controls verbosity) supplied to LBFGS

`%lambda` Initial value of `%lambda` to be used for calculating the augmented Lagrangian

This implementation minimizes the augmented Lagrangian by applying the limited-memory BFGS (LBFGS) algorithm, which is a quasi-Newton algorithm.

`load(augmented_lagrangian)` loads this function.

See also `lbfgs`.

References:

[1] `http://www-fp.mcs.anl.gov/otc/Guide/OptWeb/continuous/constrained/nonlinearcon/auglag.html`

[2] `http://www.cs.ubc.ca/spider/ascher/542/chap10.pdf`

Example:

```
(%i1) load (lbfgs);
(%o1)     /maxima/share/lbfgs/lbfgs.mac
(%i2) load (augmented_lagrangian);
(%o2)
   /maxima/share/contrib/augmented_lagrangian.mac
(%i3) FOM: x^2 + 2*y^2;
                                 2    2
(%o3)                         2 y  + x
(%i4) xx: [x, y];
(%o4)                          [x, y]
(%i5) C: [x + y - 1];
(%o5)                        [y + x - 1]
(%i6) yy: [1, 1];
(%o6)                          [1, 1]
(%i7) augmented_lagrangian_method(FOM, xx, C, yy, iprint=[-1,0]);
(%o7) [[x = 0.66665984108002, y = 0.33334027245545],
                                  %lambda = [- 1.333337940892525]]
```

43 bode

43.1 Functions and Variables for bode

bode_gain (*H, range, ...plot_opts...*) Function

Function to draw Bode gain plots.

Examples (1 through 7 from

`http://www.swarthmore.edu/NatSci/echeeve1/Ref/Bode/BodeHow.html`,

8 from Ron Crummett):

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

(%i2) H1 (s) := 100 * (1 + s) / ((s + 10) * (s + 100))$

(%i3) bode_gain (H1 (s), [w, 1/1000, 1000])$

(%i4) H2 (s) := 1 / (1 + s/omega0)$

(%i5) bode_gain (H2 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i6) H3 (s) := 1 / (1 + s/omega0)^2$

(%i7) bode_gain (H3 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i8) H4 (s) := 1 + s/omega0$

(%i9) bode_gain (H4 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i10) H5 (s) := 1/s$

(%i11) bode_gain (H5 (s), [w, 1/1000, 1000])$

(%i12) H6 (s) := 1/((s/omega0)^2 + 2 * zeta * (s/omega0) + 1)$

(%i13) bode_gain (H6 (s), [w, 1/1000, 1000]),
                  omega0 = 10, zeta = 1/10$

(%i14) H7 (s) := (s/omega0)^2 + 2 * zeta * (s/omega0) + 1$

(%i15) bode_gain (H7 (s), [w, 1/1000, 1000]),
                  omega0 = 10, zeta = 1/10$

(%i16) H8 (s) := 0.5 / (0.0001 * s^3 + 0.002 * s^2 + 0.01 * s)$

(%i17) bode_gain (H8 (s), [w, 1/1000, 1000])$
```

To use this function write first `load("bode")`. See also `bode_phase`

bode_phase (*H, range, ...plot_opts...*) Function

Function to draw Bode phase plots.

Examples (1 through 7 from

`http://www.swarthmore.edu/NatSci/echeeve1/Ref/Bode/BodeHow.html`,

8 from Ron Crummett):

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

(%i2) H1 (s) := 100 * (1 + s) / ((s + 10) * (s + 100))$

(%i3) bode_phase (H1 (s), [w, 1/1000, 1000])$

(%i4) H2 (s) := 1 / (1 + s/omega0)$

(%i5) bode_phase (H2 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i6) H3 (s) := 1 / (1 + s/omega0)^2$

(%i7) bode_phase (H3 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i8) H4 (s) := 1 + s/omega0$

(%i9) bode_phase (H4 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i10) H5 (s) := 1/s$

(%i11) bode_phase (H5 (s), [w, 1/1000, 1000])$

(%i12) H6 (s) := 1/((s/omega0)^2 + 2 * zeta * (s/omega0) + 1)$

(%i13) bode_phase (H6 (s), [w, 1/1000, 1000]),
                   omega0 = 10, zeta = 1/10$

(%i14) H7 (s) := (s/omega0)^2 + 2 * zeta * (s/omega0) + 1$

(%i15) bode_phase (H7 (s), [w, 1/1000, 1000]),
                   omega0 = 10, zeta = 1/10$

(%i16) H8 (s) := 0.5 / (0.0001 * s^3 + 0.002 * s^2 + 0.01 * s)$

(%i17) bode_phase (H8 (s), [w, 1/1000, 1000])$

(%i18) block ([bode_phase_unwrap : false],
              bode_phase (H8 (s), [w, 1/1000, 1000]));

(%i19) block ([bode_phase_unwrap : true],
              bode_phase (H8 (s), [w, 1/1000, 1000]));
```

To use this function write first `load("bode")`. See also `bode_gain`

44 contrib_ode

44.1 Introduction to contrib_ode

Maxima's ordinary differential equation (ODE) solver `ode2` solves elementary linear ODEs of first and second order. The function `contrib_ode` extends `ode2` with additional methods for linear and non-linear first order ODEs and linear homogeneous second order ODEs. The code is still under development and the calling sequence may change in future releases. Once the code has stabilized it may be moved from the contrib directory and integrated into Maxima.

This package must be loaded with the command `load('contrib_ode)` before use.

The calling convention for `contrib_ode` is identical to `ode2`. It takes three arguments: an ODE (only the left hand side need be given if the right hand side is 0), the dependent variable, and the independent variable. When successful, it returns a list of solutions.

The form of the solution differs from `ode2`. As non-linear equations can have multiple solutions, `contrib_ode` returns a list of solutions. Each solution can have a number of forms:

- an explicit solution for the dependent variable,
- an implicit solution for the dependent variable,
- a parametric solution in terms of variable `%t`, or
- a tranformation into another ODE in variable `%u`.

`%c` is used to represent the constant of integration for first order equations. `%k1` and `%k2` are the constants for second order equations. If `contrib_ode` cannot obtain a solution for whatever reason, it returns `false`, after perhaps printing out an error message.

It is necessary to return a list of solutions, as even first order non-linear ODEs can have multiple solutions. For example:

```
(%i1) load('contrib_ode)$

(%i2) eqn:x*'diff(y,x)^2-(1+x*y)*'diff(y,x)+y=0;

                     dy 2             dy
(%o2)             x (--)  - (x y + 1) -- + y = 0
                     dx               dx
(%i3) contrib_ode(eqn,y,x);

                                                 x
(%o3)              [y = log(x) + %c, y = %c %e ]
(%i4) method;

(%o4)                          factor
```

Nonlinear ODEs can have singular solutions without constants of integration, as in the second solution of the following example:

```
(%i1) load('contrib_ode)$
```

```
(%i2) eqn:'diff(y,x)^2+x*'diff(y,x)-y=0;

                        dy 2     dy
(%o2)                  (--)  + x -- - y = 0
                        dx       dx
(%i3) contrib_ode(eqn,y,x);

                                            2
                                 2          x
(%o3)              [y = %c x + %c , y = - --]
                                           4
(%i4) method;

(%o4)                         clairault
```

The following ODE has two parametric solutions in terms of the dummy variable `%t`. In this case the parametric solutions can be manipulated to give explicit solutions.

```
(%i1) load('contrib_ode)$

(%i2) eqn:'diff(y,x)=(x+y)^2;

                           dy          2
(%o2)                      -- = (y + x)
                           dx
(%i3) contrib_ode(eqn,y,x);

(%o3) [[x = %c - atan(sqrt(%t)), y = - x - sqrt(%t)],
                     [x = atan(sqrt(%t)) + %c, y = sqrt(%t) - x]]
(%i4) method;

(%o4)                           lagrange
```

The following example (Kamke 1.112) demonstrates an implicit solution.

```
(%i1) load('contrib_ode)$

(%i2) assume(x>0,y>0);

(%o2)                       [x > 0, y > 0]
(%i3) eqn:x*'diff(y,x)-x*sqrt(y^2+x^2)-y;

                      dy           2    2
(%o3)               x -- - x sqrt(y  + x ) - y
                      dx
(%i4) contrib_ode(eqn,y,x);

                                  y
(%o4)                   [x - asinh(-) = %c]
                                  x
(%i5) method;
```

```
(%o5)                          lie
```

The following Riccati equation is transformed into a linear second order ODE in the variable %u. Maxima is unable to solve the new ODE, so it is returned unevaluated.

```
(%i1) load('contrib_ode)$

(%i2) eqn:x^2*'diff(y,x)=a+b*x^n+c*x^2*y^2;

                    2 dy      2  2      n
(%o2)              x  -- = c x  y  + b x  + a
                       dx
(%i3) contrib_ode(eqn,y,x);

               d%u
               ---                               2
               dx        2     n - 2   a      d %u
(%o3)  [[y = - ----, %u c  (b x      + --) + ---- c = 0]]
               %u c                     2       2
                                       x      dx
(%i4) method;

(%o4)                        riccati
```

For first order ODEs `contrib_ode` calls `ode2`. It then tries the following methods: factorization, Clairault, Lagrange, Riccati, Abel and Lie symmetry methods. The Lie method is not attempted on Abel equations if the Abel method fails, but it is tried if the Riccati method returns an unsolved second order ODE.

For second order ODEs `contrib_ode` calls `ode2` then `odelin`.

Extensive debugging traces and messages are displayed if the command `put('contrib_ode,true,'verbose)` is executed.

44.2 Functions and Variables for contrib_ode

contrib_ode (*eqn, y, x*) Function

Returns a list of solutions of the ODE *eqn* with independent variable *x* and dependent variable *y*.

odelin (*eqn, y, x*) Function

`odelin` solves linear homogeneous ODEs of first and second order with independent variable *x* and dependent variable *y*. It returns a fundamental solution set of the ODE.

For second order ODEs, `odelin` uses a method, due to Bronstein and Lafaille, that searches for solutions in terms of given special functions.

```
(%i1) load('contrib_ode);

(%i2) odelin(x*(x+1)*'diff(y,x,2)+(x+5)*'diff(y,x,1)+(-4)*y,y,x);
...trying factor method
```

```
...solving 7 equations in 4 variables
...trying the Bessel solver
...solving 1 equations in 2 variables
...trying the F01 solver
...solving 1 equations in 3 variables
...trying the spherodial wave solver
...solving 1 equations in 4 variables
...trying the square root Bessel solver
...solving 1 equations in 2 variables
...trying the 2F1 solver
...solving 9 equations in 5 variables
       gauss_a(- 6, - 2, - 3, - x)  gauss_b(- 6, - 2, - 3, - x)
(%o2) {---------------------------, ---------------------------}
                     4                            4
                    x                            x
```

ode_check (*eqn, soln*) Function

Returns the value of ODE *eqn* after substituting a possible solution *soln*. The value is equivalent to zero if *soln* is a solution of *eqn*.

```
(%i1) load('contrib_ode)$

(%i2) eqn:'diff(y,x,2)+(a*x+b)*y;

                          2
                         d y
(%o2)                    --- + (a x + b) y
                           2
                         dx
(%i3) ans:[y = bessel_y(1/3,2*(a*x+b)^(3/2)/(3*a))*%k2*sqrt(a*x+b)
         +bessel_j(1/3,2*(a*x+b)^(3/2)/(3*a))*%k1*sqrt(a*x+b)];

                                    3/2
                      1  2 (a x + b)
(%o3) [y = bessel_y(-, --------------) %k2 sqrt(a x + b)
                      3       3 a
                                          3/2
                              1  2 (a x + b)
                  + bessel_j(-, --------------) %k1 sqrt(a x + b)]
                              3       3 a
(%i4) ode_check(eqn,ans[1]);

(%o4)                             0
```

method System variable

The variable `method` is set to the successful solution method.

%c Variable

`%c` is the integration constant for first order ODEs.

%k1 Variable

`%k1` is the first integration constant for second order ODEs.

%k2 Variable

`%k2` is the second integration constant for second order ODEs.

gauss_a (*a*, *b*, *c*, *x*) Function

`gauss_a(a,b,c,x)` and `gauss_b(a,b,c,x)` are 2F1 geometric functions. They represent any two independent solutions of the hypergeometric differential equation `x(1-x) diff(y,x,2) + [c-(a+b+1)x diff(y,x) - aby = 0` (A&S 15.5.1).

The only use of these functions is in solutions of ODEs returned by `odelin` and `contrib_ode`. The definition and use of these functions may change in future releases of Maxima.

See also `gauss_b`, `dgauss_a` and `gauss_b`.

gauss_b (*a*, *b*, *c*, *x*) Function

See `gauss_a`.

dgauss_a (*a*, *b*, *c*, *x*) Function

The derivative with respect to *x* of `gauss_a(`*a*`, `*b*`, `*c*`, `*x*`)`.

dgauss_b (*a*, *b*, *c*, *x*) Function

The derivative with respect to *x* of `gauss_b(`*a*`, `*b*`, `*c*`, `*x*`)`.

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

Kummer's M function, as defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Section 13.1.2.

The only use of this function is in solutions of ODEs returned by `odelin` and `contrib_ode`. The definition and use of this function may change in future releases of Maxima.

See also `kummer_u`, `dkummer_m` and `dkummer_u`.

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

Kummer's U function, as defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Section 13.1.3.

See `kummer_m`.

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

The derivative with respect to *x* of `kummer_m(`*a*`, `*b*`, `*x*`)`.

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

The derivative with respect to *x* of `kummer_u(`*a*`, `*b*`, `*x*`)`.

44.3 Possible improvements to contrib_ode

These routines are work in progress. I still need to:

- Extend the FACTOR method `ode1_factor` to work for multiple roots.
- Extend the FACTOR method `ode1_factor` to attempt to solve higher order factors. At present it only attemps to solve linear factors.
- Fix the LAGRANGE routine `ode1_lagrange` to prefer real roots over complex roots.
- Add additional methods for Riccati equations.
- Improve the detection of Abel equations of second kind. The exisiting pattern matching is weak.
- Work on the Lie symmetry group routine `ode1_lie`. There are quite a few problems with it: some parts are unimplemented; some test cases seem to run forever; other test cases crash; yet others return very complex "solutions". I wonder if it really ready for release yet.
- Add more test cases.

44.4 Test cases for contrib_ode

The routines have been tested on a approximately one thousand test cases from Murphy, Kamke, Zwillinger and elsewhere. These are included in the tests subdirectory.

- The Clairault routine `ode1_clairault` finds all known solutions, including singular solutions, of the Clairault equations in Murphy and Kamke.
- The other routines often return a single solution when multiple solutions exist.
- Some of the "solutions" from `ode1_lie` are overly complex and impossible to check.
- There are some crashes.

44.5 References for contrib_ode

1. E. Kamke, Differentialgleichungen Losungsmethoden und Losungen, Vol 1, Geest & Portig, Leipzig, 1961
2. G. M. Murphy, Ordinary Differential Equations and Their Solutions, Van Nostrand, New York, 1960
3. D. Zwillinger, Handbook of Differential Equations, 3rd edition, Academic Press, 1998
4. F. Schwarz, Symmetry Analysis of Abel's Equation, Studies in Applied Mathematics, 100:269-294 (1998)
5. F. Schwarz, Algorithmic Solution of Abel's Equation, Computing 61, 39-49 (1998)
6. E. S. Cheb-Terrab, A. D. Roche, Symmetries and First Order ODE Patterns, Computer Physics Communications 113 (1998), p 239. (`http://lie.uwaterloo.ca/papers/ode_vii.pdf`)
7. E. S. Cheb-Terrab, T. Koloknikov, First Order ODEs, Symmetries and Linear Transformations, European Journal of Applied Mathematics, Vol. 14, No. 2, pp. 231-246 (2003). (`http://arxiv.org/abs/math-ph/0007023`, `http://lie.uwaterloo.ca/papers/ode_iv.pdf`)

8. G. W. Bluman, S. C. Anco, Symmetry and Integration Methods for Differential Equations, Springer, (2002)
9. M. Bronstein, S. Lafaille, Solutions of linear ordinary differential equations in terms of special functions, Proceedings of ISSAC 2002, Lille, ACM Press, 23-28. (`http://www-sop.inria.fr/cafe/Manuel.Bronstein/publications/issac2002.pdf`)

45 descriptive

45.1 Introduction to descriptive

Package `descriptive` contains a set of functions for making descriptive statistical computations and graphing. Together with the source code there are three data sets in your Maxima tree: `pidigits.data`, `wind.data` and `biomed.data`.

Any statistics manual can be used as a reference to the functions in package `descriptive`.

For comments, bugs or suggestions, please contact me at *'mario AT edu DOT xunta DOT es'*.

Here is a simple example on how the descriptive functions in `descriptive` do they work, depending on the nature of their arguments, lists or matrices,

```
(%i1) load (descriptive)$
(%i2) /* univariate sample */   mean ([a, b, c]);
                               c + b + a
(%o2)                          ---------
                                   3
(%i3) matrix ([a, b], [c, d], [e, f]);
                               [ a  b ]
                               [      ]
(%o3)                          [ c  d ]
                               [      ]
                               [ e  f ]
(%i4) /* multivariate sample */ mean (%);
                         e + c + a  f + d + b
(%o4)                   [---------, ---------]
                             3          3
```

Note that in multivariate samples the mean is calculated for each column.

In case of several samples with possible different sizes, the Maxima function `map` can be used to get the desired results for each sample,

```
(%i1) load (descriptive)$
(%i2) map (mean, [[a, b, c], [d, e]]);
                           c + b + a  e + d
(%o2)                     [---------, -----]
                               3        2
```

In this case, two samples of sizes 3 and 2 were stored into a list.

Univariate samples must be stored in lists like

```
(%i1) s1 : [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5];
(%o1)           [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
```

and multivariate samples in matrices as in

```
(%i1) s2 : matrix ([13.17, 9.29], [14.71, 16.88], [18.50, 16.88],
             [10.58, 6.63], [13.33, 13.25], [13.21,  8.12]);
                           [ 13.17  9.29  ]
                           [              ]
```

```
                          [ 14.71  16.88 ]
                          [              ]
                          [ 18.5   16.88 ]
(%o1)                     [              ]
                          [ 10.58  6.63  ]
                          [              ]
                          [ 13.33  13.25 ]
                          [              ]
                          [ 13.21  8.12  ]
```

In this case, the number of columns equals the random variable dimension and the number of rows is the sample size.

Data can be introduced by hand, but big samples are usually stored in plain text files. For example, file `pidigits.data` contains the first 100 digits of number `%pi`:

```
3
1
4
1
5
9
2
6
5
3 ...
```

In order to load these digits in Maxima,

```
(%i1) s1 : read_list (file_search ("pidigits.data"))$
(%i2) length (s1);
(%o2)                          100
```

On the other hand, file `wind.data` contains daily average wind speeds at 5 meteorological stations in the Republic of Ireland (This is part of a data set taken at 12 meteorological stations. The original file is freely downloadable from the StatLib Data Repository and its analysis is discused in Haslett, J., Raftery, A. E. (1989) *Space-time Modelling with Long-memory Dependence: Assessing Ireland's Wind Power Resource, with Discussion*. Applied Statistics 38, 1-50). This loads the data:

```
(%i1) s2 : read_matrix (file_search ("wind.data"))$
(%i2) length (s2);
(%o2)                          100
(%i3) s2 [%]; /* last record */
(%o3)            [3.58, 6.0, 4.58, 7.62, 11.25]
```

Some samples contain non numeric data. As an example, file `biomed.data` (which is part of another bigger one downloaded from the StatLib Data Repository) contains four blood measures taken from two groups of patients, `A` and `B`, of different ages,

```
(%i1) s3 : read_matrix (file_search ("biomed.data"))$
(%i2) length (s3);
(%o2)                          100
(%i3) s3 [1]; /* first record */
(%o3)            [A, 30, 167.0, 89.0, 25.6, 364]
```

The first individual belongs to group `A`, is 30 years old and his/her blood measures were 167.0, 89.0, 25.6 and 364.

One must take care when working with categorical data. In the next example, symbol `a` is asigned a value in some previous moment and then a sample with categorical value `a` is taken,

```
(%i1) a : 1$
(%i2) matrix ([a, 3], [b, 5]);
                              [ 1  3 ]
(%o2)                         [      ]
                              [ b  5 ]
```

45.2 Functions and Variables for data manipulation

continuous_freq (*list*) Function
continuous_freq (*list*, *m*) Function

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*) Function

Counts absolute frequencies in discrete samples, both numeric and categorical. Its unique argument is a list,

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) discrete_freq (s1);
(%o3) [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
                                [8, 8, 12, 12, 10, 8, 9, 8, 12, 13]]
```

The first list gives the sample values and the second their absolute frequencies. Commands `? col` and `? transpose` should help you to understand the last input.

subsample (*data_matrix*, *predicate_function*) Function
subsample (*data_matrix*, *predicate_function*, *col_num1*, *col_num2*, ...) Function

This is a sort of variant of the Maxima `submatrix` function. The first argument is the data matrix, the second is a predicate function and optional additional arguments are the numbers of the columns to be taken. Its behaviour is better understood with examples.

These are multivariate records in which the wind speed in the first meteorological station were greater than 18. See that in the lambda expression the *i*-th component is refered to as `v[i]`.

```
(%i1) load (descriptive)$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) subsample (s2, lambda([v], v[1] > 18));
                [ 19.38  15.37  15.12  23.09  25.25 ]
                [                                    ]
                [ 18.29  18.66  19.08  26.08  27.63 ]
(%o3)           [                                    ]
                [ 20.25  21.46  19.95  27.71  23.38 ]
                [                                    ]
                [ 18.79  18.96  14.46  26.38  21.84 ]
```

In the following example, we request only the first, second and fifth components of those records with wind speeds greater or equal than 16 in station number 1 and less than 25 knots in station number 4. The sample contains only data from stations 1, 2 and 5. In this case, the predicate function is defined as an ordinary Maxima function.

```
(%i1) load (descriptive)$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) g(x):= x[1] >= 16 and x[4] < 25$
(%i4) subsample (s2, g, 1, 2, 5);
                     [ 19.38  15.37  25.25 ]
                     [                     ]
                     [ 17.33  14.67  19.58 ]
(%o4)                [                     ]
                     [ 16.92  13.21  21.21 ]
                     [                     ]
                     [ 17.25  18.46  23.87 ]
```

Here is an example with the categorical variables of `biomed.data`. We want the records corresponding to those patients in group `B` who are older than 38 years.

```
(%i1) load (descriptive)$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i3) h(u):= u[1] = B and u[2] > 38 $
(%i4) subsample (s3, h);
                 [ B  39  28.0  102.3  17.1  146 ]
                 [                               ]
                 [ B  39  21.0  92.4   10.3  197 ]
                 [                               ]
                 [ B  39  23.0  111.5  10.0  133 ]
                 [                               ]
                 [ B  39  26.0  92.6   12.3  196 ]
(%o4)            [                               ]
                 [ B  39  25.0  98.7   10.0  174 ]
                 [                               ]
                 [ B  39  21.0  93.2   5.9   181 ]
                 [                               ]
                 [ B  39  18.0  95.0   11.3  66  ]
                 [                               ]
```

```
                    [ B  39  39.0  88.5   7.6   168 ]
```

Probably, the statistical analysis will involve only the blood measures,

```
(%i1) load (descriptive)$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i3) subsample (s3, lambda([v], v[1] = B and v[2] > 38),
                 3, 4, 5, 6);
                   [ 28.0  102.3  17.1  146 ]
                   [                       ]
                   [ 21.0  92.4   10.3  197 ]
                   [                       ]
                   [ 23.0  111.5  10.0  133 ]
                   [                       ]
                   [ 26.0  92.6   12.3  196 ]
(%o3)              [                       ]
                   [ 25.0  98.7   10.0  174 ]
                   [                       ]
                   [ 21.0  93.2   5.9   181 ]
                   [                       ]
                   [ 18.0  95.0   11.3  66  ]
                   [                       ]
                   [ 39.0  88.5   7.6   168 ]
```

This is the multivariate mean of `s3`,

```
(%i1) load (descriptive)$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i3) mean (s3);
       65 B + 35 A  317          6 NA + 8145.0
(%o3) [-----------, ---, 87.178, -------------, 18.123,
           100      10                100
                                                  3 NA + 19587
                                                  ------------]
                                                      100
```

Here, the first component is meaningless, since `A` and `B` are categorical, the second component is the mean age of individuals in rational form, and the fourth and last values exhibit some strange behaviour. This is because symbol `NA` is used here to indicate *non available* data, and the two means are nonsense. A possible solution would be to take out from the matrix those rows with `NA` symbols, although this deserves some loss of information.

```
(%i1) load (descriptive)$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i3) g(v):= v[4] # NA and v[6] # NA $
(%i4) mean(subsample(s3, g, 3,4,5,6));
(%o4) [79.4923076923077, 86.2032967032967, 16.93186813186813,
                                                        2514
                                                        ----]
                                                         13
```

45.3 Functions and Variables for descriptive statistics

mean (*list*) Function
mean (*matrix*) Function

This is the sample mean, defined as

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) mean (s1);
                                 471
(%o3)                            ---
                                 100
(%i4) %, numer;
(%o4)                           4.71
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) mean (s2);
(%o6)     [9.9485, 10.1607, 10.8685, 15.7166, 14.8441]
```

var (*list*) Function
var (*matrix*) Function

This is the sample variance, defined as

$$\frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^2$$

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) var (s1), numer;
(%o3)                  8.425899999999999
```

See also function `var1`.

var1 (*list*) Function
var1 (*matrix*) Function

This is the sample variance, defined as

$$\frac{1}{n-1}\sum_{i=1}^{n} (x_i - \bar{x})^2$$

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) var1 (s1), numer;
(%o3)                   8.5110101010101
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) var1 (s2);
(%o5) [17.39586540404041, 15.13912778787879, 15.63204924242424,
                            32.50152569696971, 24.66977392929294]
```

See also function `var`.

std (*list*) Function
std (*matrix*) Function

This is the the square root of function `var`, the variance with denominator n.

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) std (s1), numer;
(%o3)                   2.902740084816414
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) std (s2);
(%o5) [4.149928523480858, 3.871399812729241, 3.933920277534866,
                            5.672434260526957, 4.941970881136392]
```

See also functions `var` and `std1`.

std1 (*list*) Function
std1 (*matrix*) Function

This is the the square root of function `var1`, the variance with denominator $n - 1$.

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) std1 (s1), numer;
(%o3)                   2.917363553109228
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) std1 (s2);
(%o5) [4.17083509672109, 3.89090320978032, 3.953738641137555,
                            5.701010936401517, 4.966867617451963]
```

See also functions `var1` and `std`.

noncentral_moment (*list*, *k*) Function
noncentral_moment (*matrix*, *k*) Function

The non central moment of order k, defined as

$$\frac{1}{n}\sum_{i=1}^{n} x_i^k$$

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) noncentral_moment (s1, 1), numer; /* the mean */
(%o3)                          4.71
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) noncentral_moment (s2, 5);
(%o5) [319793.8724761506, 320532.1923892463, 391249.5621381556,
                            2502278.205988911, 1691881.797742255]
```

See also function `central_moment`.

central_moment (*list*, *k*) Function
central_moment (*matrix*, *k*) Function

The central moment of order k, defined as

$$\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^k$$

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) central_moment (s1, 2), numer; /* the variance */
(%o3)                    8.425899999999999
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) central_moment (s2, 3);
(%o5) [11.29584771375004, 16.97988248298583, 5.626661952750102,
                              37.5986572057918, 25.85981904394192]
```

See also functions `central_moment` and `mean`.

cv (*list*) Function
cv (*matrix*) Function

The variation coefficient is the quotient between the sample standard deviation (`std`) and the `mean`,

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) cv (s1), numer;
(%o3)                    .6193977819764815
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) cv (s2);
(%o5) [.4192426091090204, .3829365309260502, 0.363779605385983,
                             .3627381836021478, .3346021393989506]
```

See also functions `std` and `mean`.

mini (*list*) Function
mini (*matrix*) Function

This is the minimum value of the sample *list*,

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) mini (s1);
(%o3)                             0
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) mini (s2);
(%o5)             [0.58, 0.5, 2.67, 5.25, 5.17]
```

See also function `maxi`.

maxi (*list*) Function
maxi (*matrix*) Function

This is the maximum value of the sample *list*,

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) maxi (s1);
(%o3)                          9
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) maxi (s2);
(%o5)          [20.25, 21.46, 20.04, 29.63, 27.63]
```

See also function `mini`.

range (*list*) Function
range (*matrix*) Function

The range is the difference between the extreme values.

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) range (s1);
(%o3)                          9
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) range (s2);
(%o5)          [19.67, 20.96, 17.37, 24.38, 22.46]
```

quantile (*list*, *p*) Function
quantile (*matrix*, *p*) Function

This is the p-quantile, with p a number in $[0, 1]$, of the sample *list*. Although there are several definitions for the sample quantile (Hyndman, R. J., Fan, Y. (1996) *Sample quantiles in statistical packages*. American Statistician, 50, 361-365), the one based on linear interpolation is implemented in package `descriptive`.

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) /* 1st and 3rd quartiles */
         [quantile (s1, 1/4), quantile (s1, 3/4)], numer;
(%o3)                      [2.0, 7.25]
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) quantile (s2, 1/4);
(%o5)    [7.2575, 7.477500000000001, 7.82, 11.28, 11.48]
```

median (*list*) Function
median (*matrix*) Function

Once the sample is ordered, if the sample size is odd the median is the central value, otherwise it is the mean of the two central values.

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) median (s1);
                               9
```

```
(%o3)                                -
                                     2
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) median (s2);
(%o5)          [10.06, 9.855, 10.73, 15.48, 14.105]
```

The median is the 1/2-quantile.

See also function `quantile`.

qrange (*list*) Function
qrange (*matrix*) Function

The interquartilic range is the difference between the third and first quartiles, `quantile(`*list*`,3/4) - quantile(`*list*`,1/4)`,

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) qrange (s1);
                                  21
(%o3)                             --
                                  4
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) qrange (s2);
(%o5) [5.385, 5.572499999999998, 6.0225, 8.729999999999999,
                                                  6.650000000000002]
```

See also function `quantile`.

mean_deviation (*list*) Function
mean_deviation (*matrix*) Function

The mean deviation, defined as

$$\frac{1}{n}\sum_{i=1}^{n}|x_i - \bar{x}|$$

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) mean_deviation (s1);
                                  51
(%o3)                             --
                                  20
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) mean_deviation (s2);
(%o5) [3.287959999999999, 3.075342, 3.23907, 4.715664000000001,
                                                  4.028546000000002]
```

See also function `mean`.

median_deviation (*list*) Function
median_deviation (*matrix*) Function

The median deviation, defined as

$$\frac{1}{n}\sum_{i=1}^{n}|x_i - med|$$

where `med` is the median of *list*.

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) median_deviation (s1);
                                5
(%o3)                           -
                                2
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) median_deviation (s2);
(%o5)           [2.75, 2.755, 3.08, 4.315, 3.31]
```

See also function `mean`.

harmonic_mean (*list*) Function
harmonic_mean (*matrix*) Function

The harmonic mean, defined as

$$\frac{n}{\sum_{i=1}^{n}\frac{1}{x_i}}$$

Example:

```
(%i1) load (descriptive)$
(%i2) y : [5, 7, 2, 5, 9, 5, 6, 4, 9, 2, 4, 2, 5]$
(%i3) harmonic_mean (y), numer;
(%o3)                   3.901858027632205
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) harmonic_mean (s2);
(%o5) [6.948015590052786, 7.391967752360356, 9.055658197151745,
                            13.44199028193692, 13.01439145898509]
```

See also functions `mean` and `geometric_mean`.

geometric_mean (*list*) Function
geometric_mean (*matrix*) Function

The geometric mean, defined as

$$\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}}$$

Example:

```
(%i1) load (descriptive)$
(%i2) y : [5, 7, 2, 5, 9, 5, 6, 4, 9, 2, 4, 2, 5]$
(%i3) geometric_mean (y), numer;
(%o3)                   4.454845412337012
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) geometric_mean (s2);
(%o5) [8.82476274347979, 9.22652604739361, 10.0442675714889,
                            14.61274126349021, 13.96184163444275]
```

See also functions `mean` and `harmonic_mean`.

kurtosis (*list*) Function
kurtosis (*matrix*) Function

The kurtosis coefficient, defined as

$$\frac{1}{ns^4}\sum_{i=1}^{n}(x_i-\bar{x})^4-3$$

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) kurtosis (s1), numer;
(%o3)                  - 1.273247946514421
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) kurtosis (s2);
(%o5) [- .2715445622195385, 0.119998784429451,
   - .4275233490482866, - .6405361979019522, - .4952382132352935]
```

See also functions `mean`, `var` and `skewness`.

skewness (*list*) Function
skewness (*matrix*) Function

The skewness coefficient, defined as

$$\frac{1}{ns^3}\sum_{i=1}^{n}(x_i-\bar{x})^3$$

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) skewness (s1), numer;
(%o3)                  .009196180476450306
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) skewness (s2);
(%o5) [.1580509020000979, .2926379232061854, .09242174416107717,
                            .2059984348148687, .2142520248890832]
```

See also functions `mean`, `var` and `kurtosis`.

pearson_skewness (*list*) Function
pearson_skewness (*matrix*) Function

Pearson's skewness coefficient, defined as

$$\frac{3\ (\bar{x} - med)}{s}$$

where *med* is the median of *list*.

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) pearson_skewness (s1), numer;
(%o3)                    .2159484029093895
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) pearson_skewness (s2);
(%o5) [- .08019976629211892, .2357036272952649,
          .1050904062491204, .1245042340592368, .4464181795804519]
```

See also functions `mean`, `var` and `median`.

quartile_skewness (*list*) Function
quartile_skewness (*matrix*) Function

The quartile skewness coefficient, defined as

$$\frac{c_{\frac{3}{4}} - 2\,c_{\frac{1}{2}} + c_{\frac{1}{4}}}{c_{\frac{3}{4}} - c_{\frac{1}{4}}}$$

where c_p is the p-quantile of sample *list*.

Example:

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) quartile_skewness (s1), numer;
(%o3)                   .04761904761904762
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) quartile_skewness (s2);
(%o5) [- 0.0408542246982353, .1467025572005382,
       0.0336239103362392, .03780068728522298, 0.210526315789474]
```

See also function `quantile`.

45.4 Functions and Variables for specific multivariate descriptive statistics

cov (*matrix*) Function

The covariance matrix of the multivariate sample, defined as

$$S = \frac{1}{n}\sum_{j=1}^{n} (X_j - \bar{X})\ (X_j - \bar{X})'$$

where X_j is the j-th row of the sample matrix.

Example:

```
(%i1) load (descriptive)$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) fpprintprec : 7$  /* change precision for pretty output */
(%i4) cov (s2);
      [ 17.22191  13.61811  14.37217  19.39624  15.42162 ]
      [                                                  ]
      [ 13.61811  14.98774  13.30448  15.15834  14.9711  ]
      [                                                  ]
(%o4) [ 14.37217  13.30448  15.47573  17.32544  16.18171 ]
      [                                                  ]
      [ 19.39624  15.15834  17.32544  32.17651  20.44685 ]
      [                                                  ]
      [ 15.42162  14.9711   16.18171  20.44685  24.42308 ]
```

See also function `cov1`.

cov1 (*matrix*) Function

The covariance matrix of the multivariate sample, defined as

$$\frac{1}{n-1}\sum_{j=1}^{n}\left(X_j-\bar{X}\right)\,\left(X_j-\bar{X}\right)'$$

where X_j is the j-th row of the sample matrix.

Example:

```
(%i1) load (descriptive)$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) fpprintprec : 7$ /* change precision for pretty output */
(%i4) cov1 (s2);
      [ 17.39587  13.75567  14.51734  19.59216  15.5774  ]
      [                                                  ]
      [ 13.75567  15.13913  13.43887  15.31145  15.12232 ]
      [                                                  ]
(%o4) [ 14.51734  13.43887  15.63205  17.50044  16.34516 ]
      [                                                  ]
      [ 19.59216  15.31145  17.50044  32.50153  20.65338 ]
      [                                                  ]
      [ 15.5774   15.12232  16.34516  20.65338  24.66977 ]
```

See also function `cov`.

global_variances (*matrix*) Function
global_variances (*matrix, logical_value*) Function

Function `global_variances` returns a list of global variance measures:

- *total variance*: `trace(S_1)`,
- *mean variance*: `trace(S_1)/p`,
- *generalized variance*: `determinant(S_1)`,
- *generalized standard deviation*: `sqrt(determinant(S_1))`,
- *efective variance* `determinant(S_1)^(1/p)`, (defined in: Peña, D. (2002) *Análisis de datos multivariantes*; McGraw-Hill, Madrid.)

- *efective standard deviation*: `determinant(S_1)^(1/(2*p))`.

where p is the dimension of the multivariate random variable and S_1 the covariance matrix returned by `cov1`.

Example:

```
(%i1) load (descriptive)$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) global_variances (s2);
(%o3) [105.338342060606, 21.06766841212119, 12874.34690469686,
         113.4651792608502, 6.636590811800794, 2.576158149609762]
```

Function `global_variances` has an optional logical argument: `global_variances(x,true)` tells Maxima that `x` is the data matrix, making the same as `global_variances(x)`. On the other hand, `global_variances(x,false)` means that `x` is not the data matrix, but the covariance matrix, avoiding its recalculation,

```
(%i1) load (descriptive)$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) s : cov1 (s2)$
(%i4) global_variances (s, false);
(%o4) [105.338342060606, 21.06766841212119, 12874.34690469686,
         113.4651792608502, 6.636590811800794, 2.576158149609762]
```

See also `cov` and `cov1`.

cor (*matrix*) Function
cor (*matrix*, *logical_value*) Function

The correlation matrix of the multivariate sample.

Example:

```
(%i1) load (descriptive)$
(%i2) fpprintprec:7$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) cor (s2);
      [   1.0     .8476339  .8803515  .8239624  .7519506 ]
      [                                                   ]
      [ .8476339    1.0     .8735834  .6902622  0.782502 ]
      [                                                   ]
(%o4) [ .8803515  .8735834    1.0     .7764065  .8323358 ]
      [                                                   ]
      [ .8239624  .6902622  .7764065    1.0     .7293848 ]
      [                                                   ]
      [ .7519506  0.782502  .8323358  .7293848    1.0    ]
```

Function `cor` has an optional logical argument: `cor(x,true)` tells Maxima that `x` is the data matrix, making the same as `cor(x)`. On the other hand, `cor(x,false)` means that `x` is not the data matrix, but the covariance matrix, avoiding its recalculation,

```
(%i1) load (descriptive)$
(%i2) fpprintprec:7$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) s : cov1 (s2)$
```

```
(%i5) cor (s, false); /* this is faster */
      [   1.0     .8476339  .8803515  .8239624  .7519506 ]
      [                                                   ]
      [ .8476339    1.0     .8735834  .6902622  0.782502 ]
      [                                                   ]
(%o5) [ .8803515  .8735834    1.0     .7764065  .8323358 ]
      [                                                   ]
      [ .8239624  .6902622  .7764065    1.0     .7293848 ]
      [                                                   ]
      [ .7519506  0.782502  .8323358  .7293848    1.0    ]
```

See also `cov` and `cov1`.

list_correlations (*matrix*) Function
list_correlations (*matrix, logical_value*) Function

Function `list_correlations` returns a list of correlation measures:

- *precision matrix*: the inverse of the covariance matrix S_1,

$$S_1^{-1} = \left(s^{ij}\right)_{i,j=1,2,...,p}$$

- *multiple correlation vector*: $(R_1^2, R_2^2, ..., R_p^2)$, with

$$R_i^2 = 1 - \frac{1}{s^{ii} s_{ii}}$$

  being an indicator of the goodness of fit of the linear multivariate regression model on X_i when the rest of variables are used as regressors.

- *partial correlation matrix*: with element (i, j) being

$$r_{ij.rest} = -\frac{s^{ij}}{\sqrt{s^{ii} s^{jj}}}$$

Example:

```
(%i1) load (descriptive)$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) z : list_correlations (s2)$
(%i4) fpprintprec : 5$ /* for pretty output */
(%i5) z[1];  /* precision matrix */
      [  .38486   - .13856   - .15626   - .10239    .031179  ]
      [                                                     ]
      [ - .13856   .34107    - .15233    .038447   - .052842 ]
      [                                                     ]
(%o5) [ - .15626  - .15233    .47296    - .024816  - .10054  ]
      [                                                     ]
      [ - .10239   .038447   - .024816   .10937    - .034033 ]
      [                                                     ]
      [ .031179   - .052842  - .10054   - .034033   .14834   ]
(%i6) z[2];  /* multiple correlation vector */
(%o6)       [.85063, .80634, .86474, .71867, .72675]
```

```
(%i7) z[3];  /* partial correlation matrix */
       [  - 1.0     .38244   .36627   .49908   - .13049 ]
       [                                                 ]
       [  .38244    - 1.0    .37927  - .19907   .23492  ]
       [                                                 ]
(%o7)  [  .36627    .37927   - 1.0    .10911    .37956  ]
       [                                                 ]
       [  .49908   - .19907  .10911   - 1.0     .26719  ]
       [                                                 ]
       [ - .13049   .23492   .37956   .26719    - 1.0   ]
```

Function `list_correlations` also has an optional logical argument: `list_correlations(x,true)` tells Maxima that `x` is the data matrix, making the same as `list_correlations(x)`. On the other hand, `list_correlations(x,false)` means that `x` is not the data matrix, but the covariance matrix, avoiding its recalculation.

See also `cov` and `cov1`.

45.5 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 (*list*) Function
scatterplot (*list*, *option_1*, *option_2*, ...) Function
scatterplot (*matrix*) Function
scatterplot (*matrix*, *option_1*, *option_2*, ...) 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.

```
(%i1) load (descriptive)$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) scatterplot(submatrix (s2, 1,2))$
```

Five dimensional scatter plot, with five classes histograms.

```
(%i1) load (descriptive)$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) scatterplot(
        s2,
        nclasses     = 5,
        fill_color   = blue,
        fill_density = 0.3,
        xtics        = 5)$
```

For plotting isolated or line-joined points in two and three dimensions, see `points`. For histogram related options, see `bars`.

See also `histogram`.

barsplot (*list*) Function
barsplot (*list*, *option_1*, *option_2*, ...) Function
barsplot (*one_column_matrix*) Function
barsplot (*one_column_matrix*, *option_1*, *option_2*, ...) Function
barsplot (*one_row_matrix*) Function
barsplot (*one_row_matrix*, *option_1*, *option_2*, ...) Function

Similar to `histogram` but for discrete, numeric or categorical, statistical variables.

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]`.

Example:

```
(%i1) load (descriptive)$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i3) barsplot(col(s3,2),
         title        = "Ages",
         xlabel       = "years",
         box_width    = 1/2,
         fill_density = 0.3)$
```

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-whishker diagrams. Argument *data* can be a list, which is not of great interest, since these diagrams are mainly used for comparing different samples, or a matrix, so it is possible to compare two or more components of a multivariate statistical variable. But it is also allowed *data* to be a list of samples with possible different sample sizes, in fact this is the only function in package `descriptive` that admits this type of data structure.

Available options are:

- 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-whishker diagram from a multivariate sample.

```
(%i1) load (descriptive)$
(%i2) s2 : read_matrix(file_search("wind.data"))$
(%i3) boxplot(s2,
        box_width  = 0.2,
        title      = "Windspeed in knots",
        xlabel     = "Stations",
        color      = red,
        line_width = 2) $
```

Box-and-whishker diagram from three samples of different sizes.

```
(%i1) load (descriptive)$
(%i2) A :
       [[6, 4, 6, 2, 4, 8, 6, 4, 6, 4, 3, 2],
        [8, 10, 7, 9, 12, 8, 10],
        [16, 13, 17, 12, 11, 18, 13, 18, 14, 12]]$
(%i3) boxplot (A)$
```

46 diag

46.1 Functions and Variables for diag

diag (*lm*) Function

Constructs a square matrix with the matrices of *lm* in the diagonal. *lm* is a list of matrices or scalars.

Example:

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

(%i2) a1:matrix([1,2,3],[0,4,5],[0,0,6])$

(%i3) a2:matrix([1,1],[1,0])$

(%i4) diag([a1,x,a2]);
                    [ 1  2  3  0  0  0 ]
                    [                  ]
                    [ 0  4  5  0  0  0 ]
                    [                  ]
                    [ 0  0  6  0  0  0 ]
(%o4)               [                  ]
                    [ 0  0  0  x  0  0 ]
                    [                  ]
                    [ 0  0  0  0  1  1 ]
                    [                  ]
                    [ 0  0  0  0  1  0 ]
```

To use this function write first `load("diag")`.

JF (*lambda*,*n*) Function

Returns the Jordan cell of order *n* with eigenvalue *lambda*.

Example:

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

(%i2) JF(2,5);
                     [ 2  1  0  0  0 ]
                     [               ]
                     [ 0  2  1  0  0 ]
                     [               ]
(%o2)                [ 0  0  2  1  0 ]
                     [               ]
                     [ 0  0  0  2  1 ]
                     [               ]
                     [ 0  0  0  0  2 ]
(%i3) JF(3,2);
                          [ 3  1 ]
```

```
(%o3)                          [      ]
                               [ 0  3 ]
```

To use this function write first `load("diag")`.

jordan (*mat*) Function

Returns the Jordan form of matrix *mat*, but codified in a Maxima list. To get the corresponding matrix, call function `dispJordan` using as argument the output of `jornan`.

Example:

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

(%i3) a:matrix([2,0,0,0,0,0,0,0],
               [1,2,0,0,0,0,0,0],
               [-4,1,2,0,0,0,0,0],
               [2,0,0,2,0,0,0,0],
               [-7,2,0,0,2,0,0,0],
               [9,0,-2,0,1,2,0,0],
               [-34,7,1,-2,-1,1,2,0],
               [145,-17,-16,3,9,-2,0,3])$

(%i34) jordan(a);
(%o4)             [[2, 3, 3, 1], [3, 1]]
(%i5) dispJordan(%);
                [ 2  1  0  0  0  0  0  0 ]
                [                        ]
                [ 0  2  1  0  0  0  0  0 ]
                [                        ]
                [ 0  0  2  0  0  0  0  0 ]
                [                        ]
                [ 0  0  0  2  1  0  0  0 ]
(%o5)           [                        ]
                [ 0  0  0  0  2  1  0  0 ]
                [                        ]
                [ 0  0  0  0  0  2  0  0 ]
                [                        ]
                [ 0  0  0  0  0  0  2  0 ]
                [                        ]
                [ 0  0  0  0  0  0  0  3 ]
```

To use this function write first `load("diag")`. See also `dispJordan` and `minimalPoly`.

dispJordan (*l*) Function

Returns the Jordan matrix associated to the codification given by the Maxima list *l*, which is the output given by function `jordan`.

Example:

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

(%i2) b1:matrix([0,0,1,1,1],
```

```
                    [0,0,0,1,1],
                    [0,0,0,0,1],
                    [0,0,0,0,0],
                    [0,0,0,0,0])$

(%i3) jordan(b1);
(%o3)                    [[0, 3, 2]]
(%i4) dispJordan(%);
                        [ 0  1  0  0  0 ]
                        [               ]
                        [ 0  0  1  0  0 ]
                        [               ]
(%o4)                   [ 0  0  0  0  0 ]
                        [               ]
                        [ 0  0  0  0  1 ]
                        [               ]
                        [ 0  0  0  0  0 ]
```

To use this function write first `load("diag")`. See also `jordan` and `minimalPoly`.

minimalPoly (*l*) Function

Returns the minimal polynomial associated to the codification given by the Maxima list *l*, which is the output given by function `jordan`.

Example:

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

(%i2) a:matrix([2,1,2,0],
               [-2,2,1,2],
               [-2,-1,-1,1],
               [3,1,2,-1])$

(%i3) jordan(a);
(%o3)                [[- 1, 1], [1, 3]]
(%i4) minimalPoly(%);
                                3
(%o4)                (x - 1)  (x + 1)
```

To use this function write first `load("diag")`. See also `jordan` and `dispJordan`.

ModeMatrix (*A*,*l*) Function

Returns the matrix M such that $(Mm1).A.M = J$, where J is the Jordan form of A. The Maxima list *l* is the codified form of the Jordan form as returned by function `jordan`.

Example:

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

(%i2) a:matrix([2,1,2,0],
          [-2,2,1,2],
          [-2,-1,-1,1],
```

```
          [3,1,2,-1])$

(%i3) jordan(a);
(%o3)               [[- 1, 1], [1, 3]]
(%i4) M: ModeMatrix(a,%);
                  [  1    - 1   1   1 ]
                  [                   ]
                  [   1               ]
                  [ - -   - 1   0   0 ]
                  [   9               ]
                  [                   ]
(%o4)             [   13              ]
                  [ - --   1   - 1  0 ]
                  [   9               ]
                  [                   ]
                  [  17               ]
                  [  --   - 1   1   1 ]
                  [  9                ]
(%i5) is(  (M^^-1).a.M = dispJordan(%o3)  );
(%o5)                        true
```

Note that `dispJordan(%o3)` is the Jordan form of matrix `a`.

To use this function write first `load("diag")`. See also `jordan` and `dispJordan`.

mat_function (*f*,*mat*) Function

Returns $f(mat)$, where f is an analytic function and *mat* a matrix. This computation is based on Cauchy's integral formula, which states that if `f(x)` is analytic and

```
mat = diag([JF(m1,n1),...,JF(mk,nk)]),
```

then

```
f(mat) = ModeMatrix*diag([f(JF(m1,n1)), ..., f(JF(mk,nk))])
                                                 *ModeMatrix^^(-1)
```

Note that there are about 6 or 8 other methods for this calculation.

Some examples follow.

Example 1:

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

(%i2) b2:matrix([0,1,0], [0,0,1], [-1,-3,-3])$

(%i3) mat_function(exp,t*b2);
                2   - t
               t  %e          - t     - t
(%o3) matrix([-------- + t %e    + %e   ,
                  2
        - t     - t                              - t
 2    %e      %e         - t            - t   %e
t  (- ----- - ----- + %e   ) + t (2 %e    - -----)
       t       2                              t
              t
```

```
                                  - t          - t     - t
       - t       - t   %e         2  %e       %e
 + 2 %e   , t (%e    - -----) + t  (----- - -----)
                           t            2       t
              2   - t            - t     - t
     - t      t  %e        2    %e      %e        - t
 + %e   ], [- --------, - t  (- ----- - ----- + %e   ),
                 2                t       2
                                           t
        - t     - t      2   - t
   2  %e      %e        t  %e          - t
- t  (----- - -----)], [-------- - t %e   ,
        2       t          2
        - t     - t                              - t
 2    %e      %e        - t           - t   %e
t  (- ----- - ----- + %e   ) - t (2 %e    - -----),
        t       2                                t
               t
      - t     - t                 - t
 2  %e      %e            - t   %e
t  (----- - -----) - t (%e    - -----)])
      2       t                   t
(%i4) ratsimp(%);
                [   2              - t ]
                [ (t  + 2 t + 2) %e    ]
                [ -------------------- ]
                [          2           ]
                [                      ]
                [         2   - t      ]
(%o4)  Col 1 = [        t  %e         ]
                [      - --------      ]
                [           2          ]
                [                      ]
                [     2          - t   ]
                [   (t  - 2 t) %e      ]
                [   ----------------   ]
                [          2           ]
          [      2        - t    ]
          [    (t  + t) %e       ]
          [                      ]
 Col 2 = [     2            - t ]
          [ - (t  - t - 1) %e    ]
          [                      ]
          [     2          - t   ]
          [   (t  - 3 t) %e      ]
          [        2   - t       ]
          [       t  %e          ]
          [       --------       ]
          [          2           ]
```

```
          [                      ]
          [      2          - t  ]
Col 3 =   [    (t  - 2 t) %e     ]
          [  - ----------------  ]
          [           2          ]
          [                      ]
          [   2              - t ]
          [ (t  - 4 t + 2) %e    ]
          [ -------------------- ]
          [          2           ]
```

Example 2:

```
(%i5) b1:matrix([0,0,1,1,1],
                [0,0,0,1,1],
                [0,0,0,0,1],
                [0,0,0,0,0],
                [0,0,0,0,0])$

(%i6) mat_function(exp,t*b1);
                  [              2     ]
                  [             t      ]
                  [ 1  0  t  t  -- + t ]
                  [             2      ]
                  [                    ]
(%o6)             [ 0  1  0  t    t    ]
                  [                    ]
                  [ 0  0  1  0    t    ]
                  [                    ]
                  [ 0  0  0  1    0    ]
                  [                    ]
                  [ 0  0  0  0    1    ]
(%i7) minimalPoly(jordan(b1));
                             3
(%o7)                       x
(%i8) ident(5)+t*b1+1/2*(t^2)*b1^^2;
                  [              2     ]
                  [             t      ]
                  [ 1  0  t  t  -- + t ]
                  [             2      ]
                  [                    ]
(%o8)             [ 0  1  0  t    t    ]
                  [                    ]
                  [ 0  0  1  0    t    ]
                  [                    ]
                  [ 0  0  0  1    0    ]
                  [                    ]
                  [ 0  0  0  0    1    ]
(%i9) mat_function(exp,%i*t*b1);
             [                          2 ]
```

```
                [                                t  ]
                [ 1  0  %i t  %i t  %i t - %i t/2 ]
                [                                   ]
                [                                   ]
(%o9)           [ 0  1   0    %i t    %i t         ]
                [                                   ]
                [ 0  0   1     0      %i t         ]
                [                                   ]
                [ 0  0   0     1        0          ]
                [                                   ]
                [ 0  0   0     0        1          ]
(%i10) mat_function(cos,t*b1)+%i*mat_function(sin,t*b1);
                 [                              2 ]
                 [                             t  ]
                 [ 1  0  %i t  %i t  %i t - %i t/2 ]
                 [                              2  ]
                 [                                 ]
(%o10)           [ 0  1   0    %i t    %i t       ]
                 [                                 ]
                 [ 0  0   1     0      %i t       ]
                 [                                 ]
                 [ 0  0   0     1        0        ]
                 [                                 ]
                 [ 0  0   0     0        1        ]
```

Example 3:

```
(%i11) a1:matrix([2,1,0,0,0,0],
                 [-1,4,0,0,0,0],
                 [-1,1,2,1,0,0],
                 [-1,1,-1,4,0,0],
                 [-1,1,-1,1,3,0],
                 [-1,1,-1,1,1,2])$

(%i12) fpow(x):=block([k],declare(k,integer),x^k)$

(%i13) mat_function(fpow,a1);
                [  k      k - 1 ]         [      k - 1    ]
                [ 3  - k 3      ]         [   k 3         ]
                [               ]         [               ]
                [       k - 1   ]         [  k      k - 1 ]
                [  - k 3        ]         [ 3  + k 3      ]
                [               ]         [               ]
                [       k - 1   ]         [      k - 1    ]
                [  - k 3        ]         [   k 3         ]
(%o13)  Col 1 = [               ] Col 2 = [               ]
                [       k - 1   ]         [      k - 1    ]
                [  - k 3        ]         [   k 3         ]
                [               ]         [               ]
                [       k - 1   ]         [      k - 1    ]
                [  - k 3        ]         [   k 3         ]
```

```
                                [               ]          [               ]
                                [       k - 1   ]          [      k - 1    ]
                                [  - k 3        ]          [   k 3         ]
                       [       0       ]           [       0       ]
                       [               ]           [               ]
                       [       0       ]           [       0       ]
                       [               ]           [               ]
                       [  k      k - 1 ]           [      k - 1    ]
                       [ 3  - k 3      ]           [   k 3         ]
                       [               ]           [               ]
Col 3 =                [       k - 1   ] Col 4 =   [  k      k - 1 ]
                       [  - k 3        ]           [ 3  + k 3      ]
                       [               ]           [               ]
                       [       k - 1   ]           [      k - 1    ]
                       [  - k 3        ]           [   k 3         ]
                       [               ]           [               ]
                       [       k - 1   ]           [      k - 1    ]
                       [  - k 3        ]           [   k 3         ]
                       [    0    ]
                       [         ]           [ 0  ]
                       [    0    ]           [    ]
                       [         ]           [ 0  ]
                       [    0    ]           [    ]
                       [         ]           [ 0  ]
Col 5 =                [    0    ] Col 6 =   [    ]
                       [         ]           [ 0  ]
                       [    k    ]           [    ]
                       [   3     ]           [ 0  ]
                       [         ]           [    ]
                       [  k    k ]           [  k ]
                       [ 3  - 2  ]           [ 2  ]
```

To use this function write first `load("diag")`.

47 distrib

47.1 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 <= 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

$$F(x) = \sum_{x_i \le x} f(x_i)$$

The mean, variance, standard deviation, skewness coefficient and kurtosis coefficient take the form

$$E[X] = \sum_{x_i} x_i f(x_i),$$

$$V[X] = \sum_{x_i} f(x_i)\,(x_i - E[X])^2,$$

$$D[X] = \sqrt{V[X]},$$

$$SK[X] = \frac{\sum_{x_i} f(x)\ (x - E[X])^3\ dx}{D[X]^3}$$

and

$$KU[X] = \frac{\sum_{x_i} f(x)\ (x - E[X])^4\ dx}{D[X]^4} - 3,$$

respectively.

Package `distrib` includes functions for simulating random variates. Some of these functions make use of optional variables indicating the algorithm to be used. The general inverse method (based on the fact that if u is an uniform random number in $(0,1)$, then *F^(-1)(u)* is a random variate with distribution F) is implemented in most cases; this is a suboptimal method in terms of timing, but useful for comparing with other algorithms. In this example, the performance of algorithms `ahrens_cheng` and `inverse` for simulating chi-square variates are compared by means of their histograms:

```
(%i1) load(distrib)$
(%i2) load(descriptive)$
(%i3) showtime: true$
Evaluation took 0.00 seconds (0.00 elapsed) using 32 bytes.
(%i4) random_chi2_algorithm: 'ahrens_cheng$
                                        histogram(random_chi2(10,500))$
Evaluation took 0.00 seconds (0.00 elapsed) using 40 bytes.
Evaluation took 0.69 seconds (0.71 elapsed) using 5.694 MB.
(%i6) random_chi2_algorithm: 'inverse$ histogram(random_chi2(10,500))$
Evaluation took 0.00 seconds (0.00 elapsed) using 32 bytes.
Evaluation took 10.15 seconds (10.17 elapsed) using 322.098 MB.
```

In order to make visual comparisons among algorithms for a discrete variate, function `barsplot` of the `descriptive` package should be used.

Note that some work remains to be done, since these simulating functions are not yet checked by more rigurous goodness of fit tests.

Please, consult an introductory manual on probability and statistics for more information about all this mathematical stuff.

There is a naming convention in package `distrib`. Every function name has two parts, the first one makes reference to the function or parameter we want to calculate,

```
Functions:
   Density function            (pdf_*)
   Distribution function       (cdf_*)
   Quantile                    (quantile_*)
   Mean                        (mean_*)
   Variance                    (var_*)
   Standard deviation          (std_*)
```

```
   Skewness coefficient        (skewness_*)
   Kurtosis coefficient        (kurtosis_*)
   Random variate              (random_*)
```

The second part is an explicit reference to the probabilistic model,

```
Continuous distributions:
   Normal              (*normal)
   Student             (*student_t)
   Chi^2               (*chi2)
   F                   (*f)
   Exponential         (*exp)
   Lognormal           (*lognormal)
   Gamma               (*gamma)
   Beta                (*beta)
   Continuous uniform  (*continuous_uniform)
   Logistic            (*logistic)
   Pareto              (*pareto)
   Weibull             (*weibull)
   Rayleigh            (*rayleigh)
   Laplace             (*laplace)
   Cauchy              (*cauchy)
   Gumbel              (*gumbel)

Discrete distributions:
   Binomial             (*binomial)
   Poisson              (*poisson)
   Bernoulli            (*bernoulli)
   Geometric            (*geometric)
   Discrete uniform     (*discrete_uniform)
   hypergeometric       (*hypergeometric)
   Negative binomial    (*negative_binomial)
```

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'*.

47.2 Functions and Variables for continuous distributions

pdf_normal (*x,m,s*) Function

Returns the value at x of the density function of a $Normal(m, s)$ random variable, with $s > 0$. To make use of this function, write first `load(distrib)`.

cdf_normal (*x,m,s*) Function

Returns the value at x of the distribution function of a $Normal(m, s)$ random variable, with $s > 0$. This function is defined in terms of Maxima's built-in error function `erf`.

```
(%i1) load (distrib)$
(%i2) assume(s>0)$ cdf_normal(x,m,s);
                                 x - m
                           erf(---------)
                               sqrt(2) s    1
(%o3)                      -------------- + -
                                 2          2
```

See also `erf`.

quantile_normal (*q,m,s*) Function

Returns the q-quantile of a $Normal(m, s)$ random variable, with $s > 0$; in other words, this is the inverse of `cdf_normal`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

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_algorithm Option variable

Default value: `box_mueller`

This is the selected algorithm for simulating random normal variates. Implemented algorithms are `box_mueller` and `inverse`:

- `box_mueller`, based on algorithm described in Knuth, D.E. (1981) *Seminumerical Algorithms. The Art of Computer Programming.* Addison-Wesley.
- `inverse`, based on the general inverse method.

See also `random_normal`.

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.

There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `random_normal_algorithm`, which defaults to `box_mueller`.

See also `random_normal_algorithm`. 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 $n > 0$. 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 $n > 0$. This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) load (distrib)$
(%i2) cdf_student_t(1/2, 7/3);
                                      1  7
(%o2)                  cdf_student_t(-, -)
                                      2  3
(%i3) %,numer;
(%o3)                   .6698450596140417
```

quantile_student_t (*q*,*n*) Function

Returns the q-quantile of a Student random variable $t(n)$, with $n > 0$; in other words, this is the inverse of `cdf_student_t`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_student_t (*n*) Function

Returns the mean of a Student random variable $t(n)$, with $n > 0$, which is always equal to 0. To make use of this function, write first `load(distrib)`.

var_student_t (*n*) Function

Returns the variance of a Student random variable $t(n)$, with $n > 2$.

```
(%i1) load (distrib)$
(%i2) assume(n>2)$  var_student_t(n);
                                 n
(%o3)                          -----
                               n - 2
```

std_student_t (*n*) Function

Returns the standard deviation of a Student random variable $t(n)$, with $n > 2$. To make use of this function, write first `load(distrib)`.

skewness_student_t (*n*) Function

Returns the skewness coefficient of a Student random variable $t(n)$, with $n > 3$, which is always equal to 0. To make use of this function, write first `load(distrib)`.

kurtosis_student_t (*n*) Function

Returns the kurtosis coefficient of a Student random variable $t(n)$, with $n > 4$. To make use of this function, write first `load(distrib)`.

random_student_t_algorithm Option variable

Default value: `ratio`

This is the selected algorithm for simulating random Student variates. Implemented algorithms are `inverse` and `ratio`:

- `inverse`, based on the general inverse method.
- `ratio`, based on the fact that if Z is a normal random variable $N(0,1)$ and S^2 is chi square random variable with n degrees of freedom, $Chi^2(n)$, then

$$X = \frac{Z}{\sqrt{\frac{S^2}{n}}}$$

  is a Student random variable with n degrees of freedom, $t(n)$.

See also `random_student_t`.

random_student_t (*n*) Function
random_student_t (*n,m*) Function

Returns a Student random variate $t(n)$, with $n > 0$. Calling `random_student_t` with a second argument m, a random sample of size m will be simulated.

There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `random_student_t_algorithm`, which defaults to `ratio`.

See also `random_student_t_algorithm`. To make use of this function, write first `load(distrib)`.

pdf_chi2 (*x,n*) Function

Returns the value at x of the density function of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a noun form based on the gamma density is returned.

```
(%i1) load (distrib)$
(%i2) pdf_chi2(x,n);
                                    n
(%o2)                  pdf_gamma(x, -, 2)
                                    2
(%i3) assume(x>0, n>0)$  pdf_chi2(x,n);
                         n/2 - 1   - x/2
```

```
                              x        %e
(%o4)                         ----------------
                                n/2       n
                               2    gamma(-)
                                         2
```

cdf_chi2 (*x*,*n*) Function

Returns the value at x of the distribution function of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

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 distribution, since the $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$.

```
(%i1) load (distrib)$
(%i2) cdf_chi2(3,4);
(%o2)                   cdf_gamma(3, 2, 2)
(%i3) cdf_chi2(3,4),numer;
(%o3)                    .4421745996289249
```

quantile_chi2 (*q*,*n*) Function

Returns the q-quantile of a Chi-square random variable $Chi^2(n)$, with $n > 0$; in other words, this is the inverse of `cdf_chi2`. Argument q must be an element of $[0, 1]$.

This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression based on the gamma quantile function, since the $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$.

```
(%i1) load (distrib)$
(%i2) quantile_chi2(0.99,9);
(%o2)                    21.66599433346194
(%i3) quantile_chi2(0.99,n);
                                          n
(%o3)              quantile_gamma(0.99, -, 2)
                                          2
```

mean_chi2 (*n*) Function

Returns the mean of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a noun form based on the gamma mean is returned.

```
(%i1) load (distrib)$
(%i2) mean_chi2(n);
                                    n
(%o2)                    mean_gamma(-, 2)
                                    2
(%i3) assume(n>0)$ mean_chi2(n);
(%o4)                             n
```

var_chi2 (n) Function

Returns the variance of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a noun form based on the gamma variance is returned.

```
(%i1) load (distrib)$
(%i2) var_chi2(n);
                                    n
(%o2)                    var_gamma(-, 2)
                                    2
(%i3) assume(n>0)$ var_chi2(n);
(%o4)                           2 n
```

std_chi2 (n) Function

Returns the standard deviation of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a noun form based on the gamma standard deviation is returned.

```
(%i1) load (distrib)$
(%i2) std_chi2(n);
                                    n
(%o2)                    std_gamma(-, 2)
                                    2
(%i3) assume(n>0)$ std_chi2(n);
(%o4)                    sqrt(2) sqrt(n)
```

skewness_chi2 (n) Function

Returns the skewness coefficient of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a noun form based on the gamma skewness coefficient is returned.

```
(%i1) load (distrib)$
(%i2) skewness_chi2(n);
                                      n
(%o2)                 skewness_gamma(-, 2)
                                      2
(%i3) assume(n>0)$ skewness_chi2(n);
                             2 sqrt(2)
(%o4)                        ---------
                              sqrt(n)
```

kurtosis_chi2 (n) Function

Returns the kurtosis coefficient of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a noun form based on the gamma kurtosis coefficient is returned.

```
(%i1) load (distrib)$
(%i2) kurtosis_chi2(n);
                                      n
(%o2)                 kurtosis_gamma(-, 2)
                                      2
(%i3) assume(n>0)$ kurtosis_chi2(n);
                                 12
(%o4)                            --
                                 n
```

random_chi2_algorithm Option variable

Default value: `ahrens_cheng`

This is the selected algorithm for simulating random Chi-square variates. Implemented algorithms are `ahrens_cheng` and `inverse`:

- `ahrens_cheng`, based on the random simulation of gamma variates. See `random_gamma_algorithm` for details.
- `inverse`, based on the general inverse method.

See also `random_chi2`.

random_chi2 (*n*) Function
random_chi2 (*n*,*m*) Function

Returns a Chi-square random variate $Chi^2(n)$, with $n > 0$. Calling `random_chi2` with a second argument m, a random sample of size m will be simulated.

There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `random_chi2_algorithm`, which defaults to `ahrens_cheng`.

See also `random_chi2_algorithm`. To make use of this function, write first `load(distrib)`.

pdf_f (*x*,*m*,*n*) Function

Returns the value at x of the density function of a F random variable $F(m, n)$, with $m, n > 0$. To make use of this function, write first `load(distrib)`.

cdf_f (*x*,*m*,*n*) Function

Returns the value at x of the distribution function of a F random variable $F(m, n)$, with $m, n > 0$. This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) load (distrib)$
(%i2) cdf_f(2,3,9/4);
                                      9
(%o2)                    cdf_f(2, 3, -)
                                      4
(%i3) %,numer;
(%o3)                   0.66756728179008
```

quantile_f (*q*,*m*,*n*) Function

Returns the *q*-quantile of a F random variable $F(m,n)$, with $m,n>0$; in other words, this is the inverse of `cdf_f`. Argument q must be an element of $[0,1]$.

This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) load (distrib)$
(%i2) quantile_f(2/5,sqrt(3),5);
                                 2
(%o2)                quantile_f(-, sqrt(3), 5)
                                 5
(%i3) %,numer;
(%o3)                     0.518947838573693
```

mean_f (*m*,*n*) Function

Returns the mean of a F random variable $F(m,n)$, with $m>0, n>2$. To make use of this function, write first `load(distrib)`.

var_f (*m*,*n*) Function

Returns the variance of a F random variable $F(m,n)$, with $m>0, n>4$. To make use of this function, write first `load(distrib)`.

std_f (*m*,*n*) Function

Returns the standard deviation of a F random variable $F(m,n)$, with $m>0, n>4$. To make use of this function, write first `load(distrib)`.

skewness_f (*m*,*n*) Function

Returns the skewness coefficient of a F random variable $F(m,n)$, with $m>0, n>6$. To make use of this function, write first `load(distrib)`.

kurtosis_f (*m*,*n*) Function

Returns the kurtosis coefficient of a F random variable $F(m,n)$, with $m>0, n>8$. To make use of this function, write first `load(distrib)`.

random_f_algorithm Option variable

Default value: `inverse`

This is the selected algorithm for simulating random F variates. Implemented algorithms are `ratio` and `inverse`:

- `ratio`, 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, $F(m,n)$.
- `inverse`, based on the general inverse method.

See also `random_f`.

random_f (*m*,*n*) Function
random_f (*m*,*n*,*k*) Function

Returns a F random variate $F(m, n)$, with $m, n > 0$. Calling `random_f` with a third argument k, a random sample of size k will be simulated.

There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `random_f_algorithm`, which defaults to `inverse`.

See also `random_f_algorithm`. To make use of this function, write first `load(distrib)`.

pdf_exp (*x*,*m*) Function

Returns the value at x of the density function of an $Exponential(m)$ random variable, with $m > 0$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull density is returned.

```
(%i1) load (distrib)$
(%i2) pdf_exp(x,m);
                                          1
(%o2)                pdf_weibull(x, 1, -)
                                          m
(%i3) assume(x>0,m>0)$  pdf_exp(x,m);
                                   - m x
(%o4)                         m %e
```

cdf_exp (*x*,*m*) Function

Returns the value at x of the distribution function of an $Exponential(m)$ random variable, with $m > 0$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull distribution is returned.

```
(%i1) load (distrib)$
(%i2) cdf_exp(x,m);
                                          1
(%o2)                cdf_weibull(x, 1, -)
                                          m
(%i3) assume(x>0,m>0)$  cdf_exp(x,m);
                                     - m x
(%o4)                         1 - %e
```

quantile_exp (*q*,*m*) Function

Returns the q-quantile of an $Exponential(m)$ random variable, with $m > 0$; in other words, this is the inverse of `cdf_exp`. Argument q must be an element of $[0, 1]$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull quantile is returned.

```
(%i1) load (distrib)$
(%i2) quantile_exp(0.56,5);
(%o2)                    .1641961104139661
(%i3) quantile_exp(0.56,m);
                                              1
(%o3)             quantile_weibull(0.56, 1, -)
                                              m
```

mean_exp (m) Function

Returns the mean of an $Exponential(m)$ random variable, with $m > 0$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull mean is returned.

```
(%i1) load (distrib)$
(%i2) mean_exp(m);
                                        1
(%o2)                  mean_weibull(1, -)
                                        m
(%i3) assume(m>0)$  mean_exp(m);
                                 1
(%o4)                            -
                                 m
```

var_exp (m) Function

Returns the variance of an $Exponential(m)$ random variable, with $m > 0$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull variance is returned.

```
(%i1) load (distrib)$
(%i2) var_exp(m);
                                        1
(%o2)                   var_weibull(1, -)
                                        m
(%i3) assume(m>0)$  var_exp(m);
                                  1
(%o4)                             --
                                   2
                                  m
```

std_exp (m) Function

Returns the standard deviation of an $Exponential(m)$ random variable, with $m > 0$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull standard deviation is returned.

```
(%i1) load (distrib)$
(%i2) std_exp(m);
                                              1
```

```
(%o2)                    std_weibull(1, -)
                                        m
(%i3) assume(m>0)$  std_exp(m);
                                  1
(%o4)                             -
                                  m
```

skewness_exp (m) Function

Returns the skewness coefficient of an $Exponential(m)$ random variable, with $m > 0$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull skewness coefficient is returned.

```
(%i1) load (distrib)$
(%i2) skewness_exp(m);
                                          1
(%o2)                skewness_weibull(1, -)
                                          m
(%i3) assume(m>0)$  skewness_exp(m);
(%o4)                             2
```

kurtosis_exp (m) Function

Returns the kurtosis coefficient of an $Exponential(m)$ random variable, with $m > 0$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull kurtosis coefficient is returned.

```
(%i1) load (distrib)$
(%i2) kurtosis_exp(m);
                                          1
(%o2)                kurtosis_weibull(1, -)
                                          m
(%i3) assume(m>0)$  kurtosis_exp(m);
(%o4)                             6
```

random_exp_algorithm Option variable

Default value: `inverse`

This is the selected algorithm for simulating random exponential variates. Implemented algorithms are `inverse`, `ahrens_cheng` and `ahrens_dieter`

- `inverse`, based on the general inverse method.
- `ahrens_cheng`, based on the fact that the $Exp(m)$ random variable is equivalent to the $Gamma(1, 1/m)$. See `random_gamma_algorithm` for details.
- `ahrens_dieter`, based on algorithm described in Ahrens, J.H. and Dieter, U. (1972) *Computer methods for sampling from the exponential and normal distributions.* Comm, ACM, 15, Oct., 873-882.

See also `random_exp`.

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.

There are three algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `random_exp_algorithm`, which defaults to `inverse`.

See also `random_exp_algorithm`. To make use of this function, write first `load(distrib)`.

pdf_lognormal (*x*,*m*,*s*) Function

Returns the value at *x* of the density function of a *Lognormal*(m, s) random variable, with $s > 0$. To make use of this function, write first `load(distrib)`.

cdf_lognormal (*x*,*m*,*s*) Function

Returns the value at *x* of the distribution function of a *Lognormal*(m, s) random variable, with $s > 0$. This function is defined in terms of Maxima's built-in error function `erf`.

```
(%i1) load (distrib)$
(%i2) assume(x>0, s>0)$  cdf_lognormal(x,m,s);
                             log(x) - m
                          erf(----------)
                             sqrt(2) s     1
(%o3)                     -------------- + -
                                 2         2
```

See also `erf`.

quantile_lognormal (*q*,*m*,*s*) Function

Returns the *q*-quantile of a *Lognormal*(m, s) random variable, with $s > 0$; in other words, this is the inverse of `cdf_lognormal`. Argument *q* must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_lognormal (*m*,*s*) Function

Returns the mean of a *Lognormal*(m, s) random variable, with $s > 0$. To make use of this function, write first `load(distrib)`.

var_lognormal (*m*,*s*) Function

Returns the variance of a *Lognormal*(m, s) random variable, with $s > 0$. To make use of this function, write first `load(distrib)`.

std_lognormal (*m*,*s*) Function

Returns the standard deviation of a *Lognormal*(m, s) random variable, with $s > 0$. To make use of this function, write first `load(distrib)`.

skewness_lognormal (*m*,*s*) Function

Returns the skewness coefficient of a *Lognormal*(m, s) random variable, with $s > 0$. To make use of this function, write first `load(distrib)`.

kurtosis_lognormal (*m*,*s*) Function

Returns the kurtosis coefficient of a $Lognormal(m, s)$ random variable, with $s > 0$. To make use of this function, write first `load(distrib)`.

random_lognormal (*m*,*s*) Function
random_lognormal (*m*,*s*,*n*) Function

Returns a $Lognormal(m, s)$ random variate, with $s > 0$. Calling `random_lognormal` with a third argument n, a random sample of size n will be simulated.

Log-normal variates are simulated by means of random normal variates. There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `random_normal_algorithm`, which defaults to `box_mueller`.

See also `random_normal_algorithm`. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the density function of a $Gamma(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the distribution function of a $Gamma(a, b)$ random variable, with $a, b > 0$.

This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) load (distrib)$
(%i2) cdf_gamma(3,5,21);
(%o2)                    cdf_gamma(3, 5, 21)
(%i3) %,numer;
(%o3)                  4.402663157135039E-7
```

quantile_gamma (*q*,*a*,*b*) Function

Returns the q-quantile of a $Gamma(a, b)$ random variable, with $a, b > 0$; in other words, this is the inverse of `cdf_gamma`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_gamma (*a*,*b*) Function

Returns the mean of a $Gamma(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

var_gamma (*a*,*b*) Function

Returns the variance of a $Gamma(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

std_gamma (*a*,*b*) Function

Returns the standard deviation of a $Gamma(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

skewness_gamma (*a*,*b*) Function

Returns the skewness coefficient of a $Gamma(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

kurtosis_gamma (*a*,*b*) Function

Returns the kurtosis coefficient of a $Gamma(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

random_gamma_algorithm Option variable

Default value: `ahrens_cheng`

This is the selected algorithm for simulating random gamma variates. Implemented algorithms are `ahrens_cheng` and `inverse`

- `ahrens_cheng`, this is a combinantion of two procedures, depending on the value of parameter *a*:

  For $a >= 1$, Cheng, R.C.H. and Feast, G.M. (1979). *Some simple gamma variate generators*. Appl. Stat., 28, 3, 290-295.

  For $0 < a < 1$, Ahrens, J.H. and Dieter, U. (1974). *Computer methods for sampling from gamma, beta, poisson and binomial cdf_tributions*. Computing, 12, 223-246.

- `inverse`, based on the general inverse method.

See also `random_gamma`.

random_gamma (*a*,*b*) Function
random_gamma (*a*,*b*,*n*) Function

Returns a $Gamma(a, b)$ random variate, with $a, b > 0$. Calling `random_gamma` with a third argument n, a random sample of size n will be simulated.

There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `random_gamma_algorithm`, which defaults to `ahrens_cheng`.

See also `random_gamma_algorithm`. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the density function of a $Beta(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the distribution function of a $Beta(a, b)$ random variable, with $a, b > 0$.

This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) load (distrib)$
(%i2) cdf_beta(1/3,15,2);
```

```
(%o2)                    cdf_beta(-, 15, 2)
                                    3
(%i3) %,numer;
(%o3)                  7.666089131388224E-7
```

quantile_beta (*q*,*a*,*b*) Function

Returns the *q*-quantile of a $Beta(a, b)$ random variable, with $a, b > 0$; in other words, this is the inverse of `cdf_beta`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_beta (*a*,*b*) Function

Returns the mean of a $Beta(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

var_beta (*a*,*b*) Function

Returns the variance of a $Beta(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

std_beta (*a*,*b*) Function

Returns the standard deviation of a $Beta(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

skewness_beta (*a*,*b*) Function

Returns the skewness coefficient of a $Beta(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

kurtosis_beta (*a*,*b*) Function

Returns the kurtosis coefficient of a $Beta(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

random_beta_algorithm Option variable

Default value: `cheng`

This is the selected algorithm for simulating random beta variates. Implemented algorithms are `cheng`, `inverse` and `ratio`

- `cheng`, this is the algorithm defined in Cheng, R.C.H. (1978). *Generating Beta Variates with Nonintegral Shape Parameters*. Communications of the ACM, 21:317-322
- `inverse`, based on the general inverse method.
- `ratio`, based on the fact that if X is a random variable $Gamma(a, 1)$ and Y is $Gamma(b, 1)$, then the ratio $X/(X + Y)$ is distributed as $Beta(a, b)$.

See also `random_beta`.

random_beta (*a*,*b*) Function
random_beta (*a*,*b*,*n*) Function

Returns a $Beta(a, b)$ random variate, with $a, b > 0$. Calling `random_beta` with a third argument n, a random sample of size n will be simulated.

There are three algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `random_beta_algorithm`, which defaults to `cheng`.

See also `random_beta_algorithm`. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the density function of a $ContinuousUniform(a, b)$ random variable, with $a < b$. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the distribution function of a $ContinuousUniform(a, b)$ random variable, with $a < b$. To make use of this function, write first `load(distrib)`.

quantile_continuous_uniform (*q*,*a*,*b*) Function

Returns the q-quantile of a $ContinuousUniform(a, b)$ random variable, with $a < b$; in other words, this is the inverse of `cdf_continuous_uniform`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_continuous_uniform (*a*,*b*) Function

Returns the mean of a $ContinuousUniform(a, b)$ random variable, with $a < b$. To make use of this function, write first `load(distrib)`.

var_continuous_uniform (*a*,*b*) Function

Returns the variance of a $ContinuousUniform(a, b)$ random variable, with $a < b$. To make use of this function, write first `load(distrib)`.

std_continuous_uniform (*a*,*b*) Function

Returns the standard deviation of a $ContinuousUniform(a, b)$ random variable, with $a < b$. To make use of this function, write first `load(distrib)`.

skewness_continuous_uniform (*a*,*b*) Function

Returns the skewness coefficient of a $ContinuousUniform(a, b)$ random variable, with $a < b$. To make use of this function, write first `load(distrib)`.

kurtosis_continuous_uniform (*a*,*b*) Function

Returns the kurtosis coefficient of a $ContinuousUniform(a, b)$ random variable, with $a < b$. To make use of this function, write first `load(distrib)`.

random_continuous_uniform (*a*,*b*) Function
random_continuous_uniform (*a*,*b*,*n*) Function

Returns a $ContinuousUniform(a, b)$ random variate, with $a < b$. Calling `random_continuous_uniform` with a third argument n, a random sample of size n will be simulated.

This is a direct application of the `random` built-in Maxima function.

See also `random`. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the density function of a $Logistic(a, b)$ random variable , with $b > 0$. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the distribution function of a $Logistic(a, b)$ random variable , with $b > 0$. To make use of this function, write first `load(distrib)`.

quantile_logistic (*q*,*a*,*b*) Function

Returns the q-quantile of a $Logistic(a, b)$ random variable , with $b > 0$; in other words, this is the inverse of `cdf_logistic`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_logistic (*a*,*b*) Function

Returns the mean of a $Logistic(a, b)$ random variable , with $b > 0$. To make use of this function, write first `load(distrib)`.

var_logistic (*a*,*b*) Function

Returns the variance of a $Logistic(a, b)$ random variable , with $b > 0$. To make use of this function, write first `load(distrib)`.

std_logistic (*a*,*b*) Function

Returns the standard deviation of a $Logistic(a, b)$ random variable , with $b > 0$. To make use of this function, write first `load(distrib)`.

skewness_logistic (*a*,*b*) Function

Returns the skewness coefficient of a $Logistic(a, b)$ random variable , with $b > 0$. To make use of this function, write first `load(distrib)`.

kurtosis_logistic (*a*,*b*) Function

Returns the kurtosis coefficient of a $Logistic(a, b)$ random variable , with $b > 0$. To make use of this function, write first `load(distrib)`.

random_logistic (*a*,*b*) Function
random_logistic (*a*,*b*,*n*) Function

Returns a $Logistic(a, b)$ random variate, with $b > 0$. Calling `random_logistic` with a third argument n, a random sample of size n will be simulated.

Only the inverse method is implemented. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the density function of a $Pareto(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the distribution function of a $Pareto(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

quantile_pareto (*q*,*a*,*b*) Function

Returns the q-quantile of a $Pareto(a, b)$ random variable, with $a, b > 0$; in other words, this is the inverse of `cdf_pareto`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_pareto (*a*,*b*) Function

Returns the mean of a $Pareto(a, b)$ random variable, with $a > 1, b > 0$. To make use of this function, write first `load(distrib)`.

var_pareto (*a*,*b*) Function

Returns the variance of a $Pareto(a, b)$ random variable, with $a > 2, b > 0$. To make use of this function, write first `load(distrib)`.

std_pareto (*a*,*b*) Function

Returns the standard deviation of a $Pareto(a, b)$ random variable, with $a > 2, b > 0$. To make use of this function, write first `load(distrib)`.

skewness_pareto (*a*,*b*) Function

Returns the skewness coefficient of a $Pareto(a, b)$ random variable, with $a > 3, b > 0$. To make use of this function, write first `load(distrib)`.

kurtosis_pareto (*a*,*b*) Function

Returns the kurtosis coefficient of a $Pareto(a, b)$ random variable, with $a > 4, b > 0$. To make use of this function, write first `load(distrib)`.

random_pareto (*a*,*b*) Function
random_pareto (*a*,*b*,*n*) Function

Returns a $Pareto(a, b)$ random variate, with $a > 0, b > 0$. Calling `random_pareto` with a third argument n, a random sample of size n will be simulated.

Only the inverse method is implemented. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the density function of a $Weibull(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the distribution function of a $Weibull(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

quantile_weibull (*q*,*a*,*b*) Function

Returns the *q*-quantile of a $Weibull(a, b)$ random variable, with $a, b > 0$; in other words, this is the inverse of `cdf_weibull`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_weibull (*a*,*b*) Function

Returns the mean of a $Weibull(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

var_weibull (*a*,*b*) Function

Returns the variance of a $Weibull(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

std_weibull (*a*,*b*) Function

Returns the standard deviation of a $Weibull(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

skewness_weibull (*a*,*b*) Function

Returns the skewness coefficient of a $Weibull(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

kurtosis_weibull (*a*,*b*) Function

Returns the kurtosis coefficient of a $Weibull(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load(distrib)`.

random_weibull (*a*,*b*) Function
random_weibull (*a*,*b*,*n*) Function

Returns a $Weibull(a, b)$ random variate, with $a, b > 0$. Calling `random_weibull` with a third argument n, a random sample of size n will be simulated.

Only the inverse method is implemented. To make use of this function, write first `load(distrib)`.

pdf_rayleigh (*x*,*b*) Function

Returns the value at x of the density function of a $Rayleigh(b)$ random variable, with $b > 0$.

The $Rayleigh(b)$ random variable is equivalent to the $Weibull(2, 1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull density is returned.

```
(%i1) load (distrib)$
(%i2) pdf_rayleigh(x,b);
                                          1
(%o2)                 pdf_weibull(x, 2, -)
                                          b
(%i3) assume(x>0,b>0)$ pdf_rayleigh(x,b);
                                       2  2
                           2       - b  x
(%o4)                    2 b  x %e
```

cdf_rayleigh (*x*,*b*) Function

Returns the value at x of the distribution function of a $Rayleigh(b)$ random variable, with $b > 0$.

The $Rayleigh(b)$ random variable is equivalent to the $Weibull(2, 1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull distribution is returned.

```
(%i1) load (distrib)$
(%i2) cdf_rayleigh(x,b);
                                        1
(%o2)                cdf_weibull(x, 2, -)
                                        b
(%i3) assume(x>0,b>0)$ cdf_rayleigh(x,b);
                                    2  2
                                - b  x
(%o4)                    1 - %e
```

quantile_rayleigh (*q*,*b*) Function

Returns the q-quantile of a $Rayleigh(b)$ random variable, with $b > 0$; in other words, this is the inverse of `cdf_rayleigh`. Argument q must be an element of $[0, 1]$.

The $Rayleigh(b)$ random variable is equivalent to the $Weibull(2, 1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull quantile is returned.

```
(%i1) load (distrib)$
(%i2) quantile_rayleigh(0.99,b);
                                          1
(%o2)            quantile_weibull(0.99, 2, -)
                                          b
(%i3) assume(x>0,b>0)$ quantile_rayleigh(0.99,b);
                       2.145966026289347
(%o4)                  -----------------
                               b
```

mean_rayleigh (*b*) Function

Returns the mean of a $Rayleigh(b)$ random variable, with $b > 0$.

The $Rayleigh(b)$ random variable is equivalent to the $Weibull(2, 1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull mean is returned.

```
(%i1) load (distrib)$
(%i2) mean_rayleigh(b);
                                      1
(%o2)                mean_weibull(2, -)
                                      b
(%i3) assume(b>0)$ mean_rayleigh(b);
                           sqrt(%pi)
(%o4)                      ---------
                              2 b
```

var_rayleigh (b) Function

Returns the variance of a $Rayleigh(b)$ random variable, with $b > 0$.

The $Rayleigh(b)$ random variable is equivalent to the $Weibull(2, 1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull variance is returned.

```
(%i1) load (distrib)$
(%i2) var_rayleigh(b);
                                        1
(%o2)                  var_weibull(2, -)
                                        b
(%i3) assume(b>0)$ var_rayleigh(b);
                                  %pi
                              1 - ---
                                   4
(%o4)                         -------
                                 2
                                b
```

std_rayleigh (b) Function

Returns the standard deviation of a $Rayleigh(b)$ random variable, with $b > 0$.

The $Rayleigh(b)$ random variable is equivalent to the $Weibull(2, 1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull standard deviation is returned.

```
(%i1) load (distrib)$
(%i2) std_rayleigh(b);
                                        1
(%o2)                  std_weibull(2, -)
                                        b
(%i3) assume(b>0)$ std_rayleigh(b);
                                     %pi
                           sqrt(1 - ---)
                                      4
(%o4)                      -------------
                                 b
```

skewness_rayleigh (b) Function

Returns the skewness coefficient of a $Rayleigh(b)$ random variable, with $b > 0$.

The $Rayleigh(b)$ random variable is equivalent to the $Weibull(2, 1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull skewness coefficient is returned.

```
(%i1) load (distrib)$
(%i2) skewness_rayleigh(b);
                                          1
(%o2)                skewness_weibull(2, -)
                                          b
(%i3) assume(b>0)$ skewness_rayleigh(b);
                           3/2
```

```
                              %pi      3 sqrt(%pi)
                              ------ - -----------
                                4           4
(%o4)                         --------------------
                                       %pi 3/2
                                  (1 - ---)
                                        4
```

kurtosis_rayleigh (*b*) Function

Returns the kurtosis coefficient of a *Rayleigh*(*b*) random variable, with $b > 0$.

The *Rayleigh*(*b*) random variable is equivalent to the $Weibull(2, 1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull kurtosis coefficient is returned.

```
(%i1) load (distrib)$
(%i2) kurtosis_rayleigh(b);
                                              1
(%o2)                 kurtosis_weibull(2, -)
                                              b
(%i3) assume(b>0)$ kurtosis_rayleigh(b);
                                    2
                                3 %pi
                           2 - ------
                                 16
(%o4)                      ---------- - 3
                                %pi 2
                           (1 - ---)
                                 4
```

random_rayleigh (*b*) Function

random_rayleigh (*b*,*n*) Function

Returns a *Rayleigh*(*b*) random variate, with $b > 0$. Calling `random_rayleigh` with a second argument *n*, a random sample of size *n* will be simulated.

Only the inverse method is implemented. To make use of this function, write first `load(distrib)`.

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

Returns the value at *x* of the density function of a $Laplace(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load(distrib)`.

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

Returns the value at *x* of the distribution function of a $Laplace(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load(distrib)`.

quantile_laplace (*q*,*a*,*b*) Function

Returns the *q*-quantile of a $Laplace(a, b)$ random variable, with $b > 0$; in other words, this is the inverse of `cdf_laplace`. Argument *q* must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_laplace (*a*,*b*) Function

Returns the mean of a $Laplace(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load(distrib)`.

var_laplace (*a*,*b*) Function

Returns the variance of a $Laplace(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load(distrib)`.

std_laplace (*a*,*b*) Function

Returns the standard deviation of a $Laplace(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load(distrib)`.

skewness_laplace (*a*,*b*) Function

Returns the skewness coefficient of a $Laplace(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load(distrib)`.

kurtosis_laplace (*a*,*b*) Function

Returns the kurtosis coefficient of a $Laplace(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load(distrib)`.

random_laplace (*a*,*b*) Function
random_laplace (*a*,*b*,*n*) Function

Returns a $Laplace(a, b)$ random variate, with $b > 0$. Calling `random_laplace` with a third argument n, a random sample of size n will be simulated.

Only the inverse method is implemented. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the density function of a $Cauchy(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the distribution function of a $Cauchy(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load(distrib)`.

quantile_cauchy (*q*,*a*,*b*) Function

Returns the q-quantile of a $Cauchy(a, b)$ random variable, with $b > 0$; in other words, this is the inverse of `cdf_cauchy`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

random_cauchy (*a*,*b*) Function
random_cauchy (*a*,*b*,*n*) Function

Returns a $Cauchy(a, b)$ random variate, with $b > 0$. Calling `random_cauchy` with a third argument n, a random sample of size n will be simulated.

Only the inverse method is implemented. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the density function of a $Gumbel(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the distribution function of a $Gumbel(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load(distrib)`.

quantile_gumbel (*q*,*a*,*b*) Function

Returns the q-quantile of a $Gumbel(a, b)$ random variable, with $b > 0$; in other words, this is the inverse of `cdf_gumbel`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_gumbel (*a*,*b*) Function

Returns the mean of a $Gumbel(a, b)$ random variable, with $b > 0$.

```
(%i1) load (distrib)$
(%i2) assume(b>0)$  mean_gumbel(a,b);
(%o3)                       %gamma b + a
```

where symbol `%gamma` stands for the Euler-Mascheroni constant. See also `%gamma`.

var_gumbel (*a*,*b*) Function

Returns the variance of a $Gumbel(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load(distrib)`.

std_gumbel (*a*,*b*) Function

Returns the standard deviation of a $Gumbel(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load(distrib)`.

skewness_gumbel (*a*,*b*) Function

Returns the skewness coefficient of a $Gumbel(a, b)$ random variable, with $b > 0$.

```
(%i1) load (distrib)$
(%i2) assume(b>0)$ skewness_gumbel(a,b);
                          12 sqrt(6) zeta(3)
(%o3)                     ------------------
                                    3
                                  %pi
(%i4) numer:true$ skewness_gumbel(a,b);
(%o5)                     1.139547099404649
```

where `zeta` stands for the Riemann's zeta function.

kurtosis_gumbel (*a*,*b*) Function

Returns the kurtosis coefficient of a $Gumbel(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load(distrib)`.

random_gumbel (*a*,*b*) Function
random_gumbel (*a*,*b*,*n*) Function

Returns a $Gumbel(a, b)$ random variate, with $b > 0$. Calling `random_gumbel` with a third argument n, a random sample of size n will be simulated.

Only the inverse method is implemented. To make use of this function, write first `load(distrib)`.

47.3 Functions and Variables for discrete distributions

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

Returns the value at x of the probability function of a $Binomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load(distrib)`.

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

Returns the value at x of the distribution function of a $Binomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer.

This function is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) load (distrib)$
(%i2) cdf_binomial(5,7,1/6);
                                          1
(%o2)                cdf_binomial(5, 7, -)
                                          6
(%i3) cdf_binomial(5,7,1/6), numer;
(%o3)                  .9998713991769548
```

quantile_binomial (*q*,*n*,*p*) Function

Returns the q-quantile of a $Binomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer; in other words, this is the inverse of `cdf_binomial`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_binomial (*n*,*p*) Function

Returns the mean of a $Binomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load(distrib)`.

var_binomial (*n*,*p*) Function

Returns the variance of a $Binomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load(distrib)`.

std_binomial (*n*,*p*) Function

Returns the standard deviation of a $Binomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load(distrib)`.

skewness_binomial (*n*,*p*) Function

Returns the skewness coefficient of a $Binomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load(distrib)`.

kurtosis_binomial (*n*,*p*) Function

Returns the kurtosis coefficient of a $Binomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load(distrib)`.

random_binomial_algorithm Option variable

Default value: `kachit`

This is the selected algorithm for simulating random binomial variates. Implemented algorithms are `kachit`, `bernoulli` and `inverse`:

- `kachit`, based on algorithm described in Kachitvichyanukul, V. and Schmeiser, B.W. (1988) *Binomial Random Variate Generation*. Communications of the ACM, 31, Feb., 216.
- `bernoulli`, based on simulation of Bernoulli trials.
- `inverse`, based on the general inverse method.

See also `random_binomial`.

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.

There are three algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `random_binomial_algorithm`, which defaults to `kachit`.

See also `random_binomial_algorithm`. To make use of this function, write first `load(distrib)`.

pdf_poisson (*x*,*m*) Function

Returns the value at x of the probability function of a $Poisson(m)$ random variable, with $m > 0$. To make use of this function, write first `load(distrib)`.

cdf_poisson (*x*,*m*) Function

Returns the value at x of the distribution function of a $Poisson(m)$ random variable, with $m > 0$.

This function is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) load (distrib)$
(%i2) cdf_poisson(3,5);
(%o2)                    cdf_poisson(3, 5)
(%i3) cdf_poisson(3,5), numer;
(%o3)                    .2650259152973617
```

quantile_poisson (*q*,*m*) Function

Returns the *q*-quantile of a *Poisson*(*m*) random variable, with $m > 0$; in other words, this is the inverse of `cdf_poisson`. Argument *q* must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_poisson (*m*) Function

Returns the mean of a *Poisson*(*m*) random variable, with $m > 0$. To make use of this function, write first `load(distrib)`.

var_poisson (*m*) Function

Returns the variance of a *Poisson*(*m*) random variable, with $m > 0$. To make use of this function, write first `load(distrib)`.

std_poisson (*m*) Function

Returns the standard deviation of a *Poisson*(*m*) random variable, with $m > 0$. To make use of this function, write first `load(distrib)`.

skewness_poisson (*m*) Function

Returns the skewness coefficient of a *Poisson*(*m*) random variable, with $m > 0$. To make use of this function, write first `load(distrib)`.

kurtosis_poisson (*m*) Function

Returns the kurtosis coefficient of a Poisson random variable *Poi*(*m*), with $m > 0$. To make use of this function, write first `load(distrib)`.

random_poisson_algorithm Option variable

Default value: `ahrens_dieter`

This is the selected algorithm for simulating random Poisson variates. Implemented algorithms are `ahrens_dieter` and `inverse`:

- `ahrens_dieter`, based on algorithm described in Ahrens, J.H. and Dieter, U. (1982) *Computer Generation of Poisson Deviates From Modified Normal Distributions*. ACM Trans. Math. Software, 8, 2, June,163-179.
- `inverse`, based on the general inverse method.

See also `random_poisson`.

random_poisson (*m*) Function
random_poisson (*m*,*n*) Function

Returns a *Poisson*(*m*) random variate, with $m > 0$. Calling `random_poisson` with a second argument *n*, a random sample of size *n* will be simulated.

There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `random_poisson_algorithm`, which defaults to `ahrens_dieter`.

See also `random_poisson_algorithm`. To make use of this function, write first `load(distrib)`.

pdf_bernoulli (*x*,*p*) Function

Returns the value at x of the probability function of a $Bernoulli(p)$ random variable, with $0 < p < 1$.

The $Bernoulli(p)$ random variable is equivalent to the $Binomial(1, p)$, therefore when Maxima has not enough information to get the result, a noun form based on the binomial probability function is returned.

```
(%i1) load (distrib)$
(%i2) pdf_bernoulli(1,p);
(%o2)                  pdf_binomial(1, 1, p)
(%i3) assume(0<p,p<1)$ pdf_bernoulli(1,p);
(%o4)                             p
```

cdf_bernoulli (*x*,*p*) Function

Returns the value at x of the distribution function of a $Bernoulli(p)$ random variable, with $0 < p < 1$. To make use of this function, write first `load(distrib)`.

quantile_bernoulli (*q*,*p*) Function

Returns the q-quantile of a $Bernoulli(p)$ random variable, with $0 < p < 1$; in other words, this is the inverse of `cdf_bernoulli`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_bernoulli (*p*) Function

Returns the mean of a $Bernoulli(p)$ random variable, with $0 < p < 1$.

The $Bernoulli(p)$ random variable is equivalent to the $Binomial(1, p)$, therefore when Maxima has not enough information to get the result, a noun form based on the binomial mean is returned.

```
(%i1) load (distrib)$
(%i2) mean_bernoulli(p);
(%o2)                   mean_binomial(1, p)
(%i3) assume(0<p,p<1)$ mean_bernoulli(p);
(%o4)                             p
```

var_bernoulli (*p*) Function

Returns the variance of a $Bernoulli(p)$ random variable, with $0 < p < 1$.

The $Bernoulli(p)$ random variable is equivalent to the $Binomial(1, p)$, therefore when Maxima has not enough information to get the result, a noun form based on the binomial variance is returned.

```
(%i1) load (distrib)$
(%i2) var_bernoulli(p);
(%o2)                   var_binomial(1, p)
(%i3) assume(0<p,p<1)$ var_bernoulli(p);
(%o4)                         (1 - p) p
```

std_bernoulli (*p*) Function

Returns the standard deviation of a $Bernoulli(p)$ random variable, with $0 < p < 1$.

The *Bernoulli*(p) random variable is equivalent to the *Binomial*($1, p$), therefore when Maxima has not enough information to get the result, a noun form based on the binomial standard deviation is returned.

```
(%i1) load (distrib)$
(%i2) std_bernoulli(p);
(%o2)                  std_binomial(1, p)
(%i3) assume(0<p,p<1)$ std_bernoulli(p);
(%o4)                  sqrt(1 - p) sqrt(p)
```

skewness_bernoulli (p) Function

Returns the skewness coefficient of a *Bernoulli*(p) random variable, with $0 < p < 1$.

The *Bernoulli*(p) random variable is equivalent to the *Binomial*($1, p$), therefore when Maxima has not enough information to get the result, a noun form based on the binomial skewness coefficient is returned.

```
(%i1) load (distrib)$
(%i2) skewness_bernoulli(p);
(%o2)                skewness_binomial(1, p)
(%i3) assume(0<p,p<1)$ skewness_bernoulli(p);
                              1 - 2 p
(%o4)                  ------------------
                       sqrt(1 - p) sqrt(p)
```

kurtosis_bernoulli (p) Function

Returns the kurtosis coefficient of a *Bernoulli*(p) random variable, with $0 < p < 1$.

The *Bernoulli*(p) random variable is equivalent to the *Binomial*($1, p$), therefore when Maxima has not enough information to get the result, a noun form based on the binomial kurtosis coefficient is returned.

```
(%i1) load (distrib)$
(%i2) kurtosis_bernoulli(p);
(%o2)                kurtosis_binomial(1, p)
(%i3) assume(0<p,p<1)$ kurtosis_bernoulli(p);
                         1 - 6 (1 - p) p
(%o4)                    ---------------
                            (1 - p) p
```

random_bernoulli (p) Function
random_bernoulli (p,n) Function

Returns a *Bernoulli*(p) random variate, with $0 < p < 1$. Calling `random_bernoulli` with a second argument n, a random sample of size n will be simulated.

This is a direct application of the `random` built-in Maxima function.

See also `random`. To make use of this function, write first `load(distrib)`.

pdf_geometric (x,p) Function

Returns the value at x of the probability function of a *Geometric*(p) random variable, with $0 < p < 1$. To make use of this function, write first `load(distrib)`.

cdf_geometric (*x*,*p*) Function

Returns the value at x of the distribution function of a *Geometric*(p) random variable, with $0 < p < 1$. To make use of this function, write first `load(distrib)`.

quantile_geometric (*q*,*p*) Function

Returns the q-quantile of a *Geometric*(p) random variable, with $0 < p < 1$; in other words, this is the inverse of `cdf_geometric`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_geometric (*p*) Function

Returns the mean of a *Geometric*(p) random variable, with $0 < p < 1$. To make use of this function, write first `load(distrib)`.

var_geometric (*p*) Function

Returns the variance of a *Geometric*(p) random variable, with $0 < p < 1$. To make use of this function, write first `load(distrib)`.

std_geometric (*p*) Function

Returns the standard deviation of a *Geometric*(p) random variable, with $0 < p < 1$. To make use of this function, write first `load(distrib)`.

skewness_geometric (*p*) Function

Returns the skewness coefficient of a *Geometric*(p) random variable, with $0 < p < 1$. To make use of this function, write first `load(distrib)`.

kurtosis_geometric (*p*) Function

Returns the kurtosis coefficient of a geometric random variable *Geo*(p), with $0 < p < 1$. To make use of this function, write first `load(distrib)`.

random_geometric_algorithm Option variable

Default value: `bernoulli`

This is the selected algorithm for simulating random geometric variates. Implemented algorithms are `bernoulli`, `devroye` and `inverse`:

- `bernoulli`, based on simulation of Bernoulli trials.
- `devroye`, based on algorithm described in Devroye, L. (1986) *Non-Uniform Random Variate Generation.* Springer Verlag, p. 480.
- `inverse`, based on the general inverse method.

See also `random_geometric`.

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.

There are three algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `random_geometric_algorithm`, which defaults to `bernoulli`.

See also `random_geometric_algorithm`. To make use of this function, write first `load(distrib)`.

pdf_discrete_uniform (*x*,*n*) Function

Returns the value at x of the probability function of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer. To make use of this function, write first `load(distrib)`.

cdf_discrete_uniform (*x*,*n*) Function

Returns the value at x of the distribution function of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer. To make use of this function, write first `load(distrib)`.

quantile_discrete_uniform (*q*,*n*) Function

Returns the q-quantile of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer; in other words, this is the inverse of `cdf_discrete_uniform`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_discrete_uniform (*n*) Function

Returns the mean of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer. To make use of this function, write first `load(distrib)`.

var_discrete_uniform (*n*) Function

Returns the variance of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer. To make use of this function, write first `load(distrib)`.

std_discrete_uniform (*n*) Function

Returns the standard deviation of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer. To make use of this function, write first `load(distrib)`.

skewness_discrete_uniform (*n*) Function

Returns the skewness coefficient of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer. To make use of this function, write first `load(distrib)`.

kurtosis_discrete_uniform (*n*) Function

Returns the kurtosis coefficient of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer. To make use of this function, write first `load(distrib)`.

random_discrete_uniform (*n*) Function
random_discrete_uniform (*n*,*m*) Function

Returns a $DiscreteUniform(n)$ random variate, with n a strictly positive integer. Calling `random_discrete_uniform` with a second argument m, a random sample of size m will be simulated.

This is a direct application of the `random` built-in Maxima function.

See also `random`. To make use of this function, write first `load(distrib)`.

pdf_hypergeometric (*x*,*n1*,*n2*,*n*) Function

Returns the value at x of the probability function of a $Hypergeometric(n1, n2, n)$ random variable, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$. To make use of this function, write first `load(distrib)`.

cdf_hypergeometric (*x*,*n1*,*n2*,*n*) Function

Returns the value at x of the distribution function of a $Hypergeometric(n1, n2, n)$ random variable, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$. To make use of this function, write first `load(distrib)`.

quantile_hypergeometric (*q*,*n1*,*n2*,*n*) Function

Returns the q-quantile of a $Hypergeometric(n1, n2, n)$ random variable, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$; in other words, this is the inverse of `cdf_hypergeometric`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_hypergeometric (*n1*,*n2*,*n*) Function

Returns the mean of a discrete uniform random variable $Hyp(n1, n2, n)$, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$. To make use of this function, write first `load(distrib)`.

var_hypergeometric (*n1*,*n2*,*n*) Function

Returns the variance of a hypergeometric random variable $Hyp(n1, n2, n)$, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$. To make use of this function, write first `load(distrib)`.

std_hypergeometric (*n1*,*n2*,*n*) Function

Returns the standard deviation of a $Hypergeometric(n1, n2, n)$ random variable, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$. To make use of this function, write first `load(distrib)`.

skewness_hypergeometric (*n1*,*n2*,*n*) Function

Returns the skewness coefficient of a $Hypergeometric(n1, n2, n)$ random variable, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$. To make use of this function, write first `load(distrib)`.

kurtosis_hypergeometric (*n1*,*n2*,*n*) Function

Returns the kurtosis coefficient of a $Hypergeometric(n1, n2, n)$ random variable, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$. To make use of this function, write first `load(distrib)`.

random_hypergeometric_algorithm Option variable

Default value: `kachit`

This is the selected algorithm for simulating random hypergeometric variates. Implemented algorithms are `kachit` and `inverse`:

- `kachit`, based on algorithm described in Kachitvichyanukul, V., Schmeiser, B.W. (1985) *Computer generation of hypergeometric random variates.* Journal of Statistical Computation and Simulation 22, 127-145.
- `inverse`, based on the general inverse method.

See also `random_hypergeometric`.

random_hypergeometric (*n1,n2,n*) Function
random_hypergeometric (*n1,n2,n,m*) Function

Returns a $Hypergeometric(n1, n2, n)$ random variate, with *n1*, *n2* and *n* non negative integers and $n <= n1+n2$. Calling `random_hypergeometric` with a fourth argument *m*, a random sample of size *m* will be simulated.

There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `random_hypergeometric_algorithm`, which defaults to `kachit`.

See also `random_hypergeometric_algorithm`. To make use of this function, write first `load(distrib)`.

pdf_negative_binomial (*x,n,p*) Function

Returns the value at x of the probability function of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load(distrib)`.

cdf_negative_binomial (*x,n,p*) Function

Returns the value at x of the distribution function of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer.

This function is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) load (distrib)$
(%i2) cdf_negative_binomial(3,4,1/8);
                                                 1
(%o2)            cdf_negative_binomial(3, 4, -)
                                                 8
(%i3) cdf_negative_binomial(3,4,1/8), numer;
(%o3)                  .006238937377929698
```

quantile_negative_binomial (*q,n,p*) Function

Returns the q-quantile of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer; in other words, this is the inverse of `cdf_negative_binomial`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load(distrib)`.

mean_negative_binomial (*n*,*p*) Function

Returns the mean of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load(distrib)`.

var_negative_binomial (*n*,*p*) Function

Returns the variance of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load(distrib)`.

std_negative_binomial (*n*,*p*) Function

Returns the standard deviation of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load(distrib)`.

skewness_negative_binomial (*n*,*p*) Function

Returns the skewness coefficient of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load(distrib)`.

kurtosis_negative_binomial (*n*,*p*) Function

Returns the kurtosis coefficient of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load(distrib)`.

random_negative_binomial_algorithm Option variable

Default value: `bernoulli`

This is the selected algorithm for simulating random negative binomial variates. Implemented algorithms are `devroye`, `bernoulli` and `inverse`:

- `devroye`, based on algorithm described in Devroye, L. (1986) *Non-Uniform Random Variate Generation*. Springer Verlag, p. 480.
- `bernoulli`, based on simulation of Bernoulli trials.
- `inverse`, based on the general inverse method.

See also `random_negative_binomial`.

random_negative_binomial (*n*,*p*) Function
random_negative_binomial (*n*,*p*,*m*) Function

Returns a $NegativeBinomial(n, p)$ random variate, with $0 < p < 1$ and n a positive integer. Calling `random_negative_binomial` with a third argument m, a random sample of size m will be simulated.

There are three algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `random_negative_binomial_algorithm`, which defaults to `bernoulli`.

See also `random_negative_binomial_algorithm`. To make use of this function, write first `load(distrib)`.

48 draw

48.1 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://www.telefonica.net/web2/biomates/maxima/gpdraw`

You need Gnuplot 4.2 to run this program.

48.2 Functions and Variables for draw

xrange Graphic option

Default value: `auto`

If `xrange` is `auto`, the range for the x coordinate is computed automatically.

If the user wants a specific interval for x, it must be given as a Maxima list, as in `xrange=[-2, 3]`.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(xrange = [-3,5],
             explicit(x^2,x,-1,1))$
```

See also `yrange` and `zrange`.

yrange Graphic option

Default value: `auto`

If `yrange` is `auto`, the range for the y coordinate is computed automatically.

If the user wants a specific interval for y, it must be given as a Maxima list, as in `yrange=[-2, 3]`.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(yrange = [-2,3],
             explicit(x^2,x,-1,1),
             xrange = [-3,3])$
```

See also `xrange` and `zrange`.

zrange Graphic option

Default value: `auto`

If `zrange` is `auto`, the range for the z coordinate is computed automatically.

If the user wants a specific interval for z, it must be given as a Maxima list, as in `zrange=[-2, 3]`.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(yrange = [-3,3],
             zrange = [-2,5],
             explicit(x^2+y^2,x,-1,1,y,-1,1),
             xrange = [-3,3])$
```

See also `xrange` and `yrange`.

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`.

terminal Graphic option

Default value: `screen`

Selects the terminal to be used by Gnuplot; possible values are: `screen` (default), `png`, `jpg`, `eps`, `eps_color`, `gif`, `animated_gif`, `wxt` and `aquaterm`.

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:

```
(%i1) load(draw)$
(%i2) /* screen terminal (default) */
      draw2d(explicit(x^2,x,-1,1))$
(%i3) /* png file */
      draw2d(terminal  = 'png,
             pic_width = 300,
             explicit(x^2,x,-1,1))$
(%i4) /* jpg file */
      draw2d(terminal   = 'jpg,
             pic_width  = 300,
             pic_height = 300,
             explicit(x^2,x,-1,1))$
(%i5) /* eps file */
      draw2d(file_name = "myfile",
             explicit(x^2,x,-1,1),
             terminal  = 'eps)$
(%i6) /* wxwidgets window */
      draw2d(explicit(x^2,x,-1,1),
             terminal  = 'wxt)$
```

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`, `pic_width`, `pic_height` and `delay`.

font Graphic option

Default value: `""` (empty string)

This option can be used to set the font face to be used by the terminal. Only one font face and size can be used throughout the plot.

Since this is a global graphics option, its position in the scene description does not matter.

See also `font_size`.

Gnuplot doesn't handle fonts by itself, it leaves this task to the support libraries of the different terminals, each one with its own philosophy about it. A brief summary follows:

- *x11*: Uses the normal x11 font server mechanism.

  Example:

  ```
  (%i1) load(draw)$
  (%i2) draw2d(font      = "Arial",
               font_size = 20,
               label(["Arial font, size 20",1,1]))$
  ```

- *windows*: The windows terminal doesn't support changing of fonts from inside the plot. Once the plot has been generated, the font can be changed right-clicking on the menu of the graph window.

- *png, jpeg, gif*: The *libgd* library uses the font path stored in the environment variable `GDFONTPATH`; in this case, it is only necessary to set option `font` to the font's name. It is also possible to give the complete path to the font file.

  Examples:

  Option `font` can be given the complete path to the font file:

  ```
  (%i1) load(draw)$
  (%i2) path: "/usr/share/fonts/truetype/freefont/" $
  (%i3) file: "FreeSerifBoldItalic.ttf" $
  (%i4) draw2d(
          font      = concat(path, file),
          font_size = 20,
          color     = red,
          label(["FreeSerifBoldItalic font, size 20",1,1]),
          terminal  = png)$
  ```

  If environment variable `GDFONTPATH` is set to the path where font files are allocated, it is possible to set graphic option `font` to the name of the font.

  ```
  (%i1) load(draw)$
  (%i2) draw2d(
          font      = "FreeSerifBoldItalic",
          font_size = 20,
          color     = red,
          label(["FreeSerifBoldItalic font, size 20",1,1]),
          terminal  = png)$
  ```

- *Postscript*: Standard Postscript fonts are: `"Times-Roman"`, `"Times-Italic"`, `"Times-Bold"`, `"Times-BoldItalic"`, `"Helvetica"`, `"Helvetica-Oblique"`, `"Helvetica-Bold"`, `"Helvetic-BoldOblique"`, `"Courier"`, `"Courier-Oblique"`, `"Courier-Bold"`, and `"Courier-BoldOblique"`.

  Example:

```
(%i1) load(draw)$
(%i2) draw2d(
        font      = "Courier-Oblique",
        font_size = 15,
        label(["Courier-Oblique font, size 15",1,1]),
        terminal = eps)$
```

- *wxt*: The *pango* library finds fonts via the `fontconfig` utility.
- *aqua*: Default is `"Times-Roman"`.

The gnuplot documentation is an important source of information about terminals and fonts.

font_size Graphic option

Default value: 12

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`.

grid Graphic option

Default value: `false`

If `grid` is `true`, a grid will be drawn on the *xy* plane.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(grid = true,
             explicit(exp(u),u,-2,2))$
```

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")$
```

xlabel Graphic option

Default value: `""` (empty string)

Option `xlabel`, a string, is the label for the *x* axis. By default, no label is written.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(xlabel = "Time",
             explicit(exp(u),u,-2,2),
             ylabel = "Population")$
```

See also `ylabel`, and `zlabel`.

ylabel Graphic option

Default value: `""` (empty string)

Option `ylabel`, a string, is the label for the *y* axis. By default, no label is written.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(xlabel = "Time",
             ylabel = "Population",
             explicit(exp(u),u,-2,2) )$
```

See also `xlabel`, and `zlabel`.

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`.

xtics Graphic option

Default value: `auto`

This graphic option controls the way tic marks are drawn on the *x* axis.

- When option `xtics` is bounded to symbol *auto*, tic marks are drawn automatically.
- When option `xtics` is bounded to symbol *none*, tic marks are not drawn.
- When option `xtics` is bounded to a positive number, this is the distance between two consecutive tic marks.

- When option `xtics` is bounded to a list of length three of the form `[start,incr,end]`, tic marks are plotted from `start` to `end` at intervals of length `incr`.
- When option `xtics` is bounded to a set of numbers of the form `{n1, n2, ...}`, tic marks are plotted at values `n1`, `n2`, ...
- When option `xtics` is bounded to a set of pairs of the form `{["label1", n1], ["label2", n2], ...}`, tic marks corresponding to values `n1`, `n2`, ... are labeled with `"label1"`, `"label2"`, ..., respectively.

Since this is a global graphics option, its position in the scene description does not matter.

Examples:

Disable tics.

```
(%i1) load(draw)$
(%i2) draw2d(xtics = 'none,
             explicit(x^3,x,-1,1)  )$
```

Tics every 1/4 units.

```
(%i1) load(draw)$
(%i2) draw2d(xtics = 1/4,
             explicit(x^3,x,-1,1)  )$
```

Tics from -3/4 to 3/4 in steps of 1/8.

```
(%i1) load(draw)$
(%i2) draw2d(xtics = [-3/4,1/8,3/4],
             explicit(x^3,x,-1,1)  )$
```

Tics at points -1/2, -1/4 and 3/4.

```
(%i1) load(draw)$
(%i2) draw2d(xtics = {-1/2,-1/4,3/4},
             explicit(x^3,x,-1,1)  )$
```

Labeled tics.

```
(%i1) load(draw)$
(%i2) draw2d(xtics = {["High",0.75],["Medium",0],["Low",-0.75]},
             explicit(x^3,x,-1,1)  )$
```

See also `ytics`, and `ztics`.

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.

ztics Graphic option

Default value: `auto`

This graphic option controls the way tic marks are drawn on the *z* axis.

See `xtics` for a complete description.

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.

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.

ztics_rotate Graphic option

Default value: `false`

If `ztics_rotate` is `true`, tic marks on the z axis are rotated 90 degrees.

Since this is a global graphics option, its position in the scene description does not matter.

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.

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.

ztics_axis Graphic option

Default value: `false`

If `ztics_axis` is `true`, tic marks and their labels are plotted just along the z axis, if it is `false` tics are plotted on the border.

Since this is a global graphics option, its position in the scene description does not matter.

xaxis Graphic option

Default value: `false`

If `xaxis` is `true`, the x axis is drawn.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(explicit(x^3,x,-1,1),
             xaxis       = true,
             xaxis_color = blue)$
```

See also `xaxis_width`, `xaxis_type` and `xaxis_color`.

xaxis_width Graphic option

Default value: 1

`xaxis_width` is the width of the *x* axis. Its value must be a positive number.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(explicit(x^3,x,-1,1),
             xaxis       = true,
             xaxis_width = 3)$
```

See also `xaxis`, `xaxis_type` and `xaxis_color`.

xaxis_type Graphic option

Default value: `dots`

`xaxis_type` indicates how the *x* axis is displayed; possible values are `solid` and `dots`.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(explicit(x^3,x,-1,1),
             xaxis       = true,
             xaxis_type  = solid)$
```

See also `xaxis`, `xaxis_width` and `xaxis_color`.

xaxis_color Graphic option

Default value: `"black"`

`xaxis_color` specifies the color for the *x* axis. See `color` to know how colors are defined.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(explicit(x^3,x,-1,1),
             xaxis       = true,
             xaxis_color = red)$
```

See also `xaxis`, `xaxis_width` and `xaxis_type`.

yaxis Graphic option

Default value: `false`

If `yaxis` is `true`, the y axis is drawn.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(explicit(x^3,x,-1,1),
             yaxis       = true,
             yaxis_color = blue)$
```

See also `yaxis_width`, `yaxis_type` and `yaxis_color`.

yaxis_width Graphic option

Default value: 1

`yaxis_width` is the width of the y axis. Its value must be a positive number.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(explicit(x^3,x,-1,1),
             yaxis       = true,
             yaxis_width = 3)$
```

See also `yaxis`, `yaxis_type` and `yaxis_color`.

yaxis_type Graphic option

Default value: `dots`

`yaxis_type` indicates how the y axis is displayed; possible values are `solid` and `dots`.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(explicit(x^3,x,-1,1),
             yaxis       = true,
             yaxis_type  = solid)$
```

See also `yaxis`, `yaxis_width` and `yaxis_color`.

yaxis_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`.

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_width Graphic option

Default value: 1

`zaxis_width` is the width of the z axis. Its value must be a positive number. This option has no effect in 2D scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
             zaxis       = true,
             zaxis_type  = solid,
             zaxis_width = 3)$
```

See also `zaxis`, `zaxis_type` and `zaxis_color`.

zaxis_type Graphic option

Default value: `dots`

`zaxis_type` indicates how the z axis is displayed; possible values are `solid` and `dots`. This option has no effect in 2D scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
             zaxis       = true,
             zaxis_type  = solid)$
```

See also `zaxis`, `zaxis_width` and `zaxis_color`.

zaxis_color Graphic option

Default value: `"black"`

`zaxis_color` specifies the color for the z axis. See `color` to know how colors are defined. This option has no effect in 2D scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
             zaxis       = true,
             zaxis_type  = solid,
             zaxis_color = red)$
```

See also `zaxis`, `zaxis_width` and `zaxis_type`.

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))$
```

rot_vertical Graphic option

Default value: 60

`rot_vertical` is the angle (in degrees) of vertical rotation (around the x axis) to set the view point in 3d scenes.

The angle is bounded to the $[0, 180]$ interval.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(rot_vertical = 170,
             explicit(sin(x^2+y^2),x,-2,2,y,-2,2) )$
```

See also `rot_horizontal`.

rot_horizontal Graphic option

Default value: 30

`rot_horizontal` is the angle (in degrees) of horizontal rotation (around the z axis) to set the view point in 3d scenes.

The angle is bounded to the $[0, 360]$ interval.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(rot_vertical = 170,
             rot_horizontal = 360,
             explicit(sin(x^2+y^2),x,-2,2,y,-2,2) )$
```

See also `rot_vertical`.

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.

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")$
```

file_name Graphic option

Default value: `"maxima_out"`

This is the name of the file where terminals `png`, `jpg`, `eps` and `eps_color` will save the graphic.

Since this is a global graphics option, its position in the scene description does not matter. It can be also used as an argument of function `draw`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(file_name = "myfile",
             explicit(x^2,x,-1,1),
             terminal  = 'png)$
```

See also `terminal`, `pic_width`, and `pic_height`.

delay Graphic option

Default value: 5

This is the delay in 1/100 seconds of frames in animated gif files.

Since this is a global graphics option, its position in the scene description does not matter. It can be also used as an argument of function `draw`.

Example:

```
(%i1) load(draw)$
(%i2) draw(
        delay     = 100,
        file_name = "zzz",
        terminal  = 'animated_gif,
        gr2d(explicit(x^2,x,-1,1)),
        gr2d(explicit(x^3,x,-1,1)),
        gr2d(explicit(x^4,x,-1,1)));
End of animation sequence
(%o2)          [gr2d(explicit), gr2d(explicit), gr2d(explicit)]
```

Option `delay` is only active in animated gif's; it is ignored in any other case.

See also `terminal`, `pic_width`, and `pic_height`.

pic_width Graphic option

Default value: 640

This is the width of the bitmap file generated by terminals `png` and `jpg`.

Since this is a global graphics option, its position in the scene description does not matter. It can be also used as an argument of function `draw`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(terminal   = 'png,
             pic_width  = 300,
             pic_height = 300,
             explicit(x^2,x,-1,1))$
```

See also `terminal`, `file_name`, and `pic_height`.

pic_height Graphic option

Default value: 640

This is the height of the bitmap file generated by terminals `png` and `jpg`.

Since this is a global graphics option, its position in the scene description does not matter. It can be also used as an argument of function `draw`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(terminal   = 'png,
             pic_width  = 300,
             pic_height = 300,
             explicit(x^2,x,-1,1))$
```

See also `terminal`, `file_name`, and `pic_width`.

eps_width Graphic option

Default value: 12

This is the width (measured in cm) of the Postscipt file generated by terminals `eps` and `eps_color`.

Since this is a global graphics option, its position in the scene description does not matter. It can be also used as an argument of function `draw`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(terminal   = 'eps,
             eps_width  = 3,
             eps_height = 3,
             explicit(x^2,x,-1,1))$
```

See also `terminal`, `file_name`, and `eps_height`.

eps_height Graphic option

Default value: 8

This is the height (measured in cm) of the Postscipt file generated by terminals `eps` and `eps_color`.

Since this is a global graphics option, its position in the scene description does not matter. It can be also used as an argument of function `draw`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(terminal   = 'eps,
             eps_width  = 3,
             eps_height = 3,
             explicit(x^2,x,-1,1))$
```

See also `terminal`, `file_name`, and `eps_width`.

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_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`.

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_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.

palette Graphic option

Default value: `color`

`palette` indicates how to map the real values of a matrix passed to object `image` onto color components.

`palette` is a vector of length three with components ranging from -36 to +36; each value is an index for a formula mapping the levels onto red, green and blue colors, respectively:

```
 0: 0               1: 0.5           2: 1
 3: x               4: x^2           5: x^3
 6: x^4             7: sqrt(x)       8: sqrt(sqrt(x))
 9: sin(90x)       10: cos(90x)     11: |x-0.5|
12: (2x-1)^2       13: sin(180x)    14: |cos(180x)|
15: sin(360x)      16: cos(360x)    17: |sin(360x)|
18: |cos(360x)|    19: |sin(720x)|  20: |cos(720x)|
21: 3x             22: 3x-1         23: 3x-2
24: |3x-1|         25: |3x-2|       26: (3x-1)/2
27: (3x-2)/2       28: |(3x-1)/2|   29: |(3x-2)/2|
30: x/0.32-0.78125 31: 2*x-0.84     32: 4x;1;-2x+1.84;x/0.08-11.5
33: |2*x - 0.5|    34: 2*x          35: 2*x - 0.5
36: 2*x - 1
```

negative numbers mean negative colour component.

`palette = gray` and `palette = color` are short cuts for `palette = [3,3,3]` and `palette = [7,5,15]`, respectively.

Since this is a global graphics option, its position in the scene description does not matter.

Examples:

```
(%i1) load(draw)$
(%i2) im: apply(
           'matrix,
            makelist(makelist(random(200),i,1,30),i,1,30))$
(%i3) /* palette = color, default */
      draw2d(image(im,0,0,30,30))$
(%i4) draw2d(palette = gray, image(im,0,0,30,30))$
(%i5) draw2d(palette = [15,20,-4],
             colorbox=false,
             image(im,0,0,30,30))$
```

See also `colorbox`.

colorbox Graphic option

Default value: `true`

If `colorbox` is `true`, a color scale is drawn together with `image` objects.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%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))$
```

See also `palette`.

enhanced3d Graphic option

Default value: `false`

If `enhanced3d` is `true`, surfaces are colored in 3d plots; in other words, it sets Gnuplot's pm3d mode.

See option `palette` to learn how palettes are specified.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(surface_hide = true,
             enhanced3d   = true,
             palette      = gray,
             explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3))$
```

point_size Graphic option

Default value: 1

`point_size` sets the size for plotted points. It must be a non negative number.

This option has no effect when graphic option `point_type` is set to `dot`.

This option affects the following graphic objects:

- `gr2d`: `points`.
- `gr3d`: `points`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(points(makelist([random(20),random(50)],k,1,10)),
        point_size = 5,
        points(makelist(k,k,1,20),makelist(random(30),k,1,20)))$
```

point_type Graphic option

Default value: 1

`point_type` indicates how isolated points are displayed; the value of this option can be any integer index greater or equal than -1, or the name of a point style: `$none` (-1), `dot` (0), `plus` (1), `multiply` (2), `asterisk` (3), `square` (4), `filled_square` (5), `circle` (6), `filled_circle` (7), `up_triangle` (8), `filled_up_triangle` (9), `down_triangle` (10), `filled_down_triangle` (11), `diamant` (12) and `filled_diamant` (13).

This option affects the following graphic objects:

- `gr2d`: `points`.
- `gr3d`: `points`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(xrange = [0,10],
             yrange = [0,10],
             point_size = 3,
             point_type = diamant,
             points([[1,1],[5,1],[9,1]]),
             point_type = filled_down_triangle,
             points([[1,2],[5,2],[9,2]]),
             point_type = asterisk,
             points([[1,3],[5,3],[9,3]]),
```

```
                    point_type = filled_diamant,
                    points([[1,4],[5,4],[9,4]]),
                    point_type = 5,
                    points([[1,5],[5,5],[9,5]]),
                    point_type = 6,
                    points([[1,6],[5,6],[9,6]]),
                    point_type = filled_circle,
                    points([[1,7],[5,7],[9,7]]),
                    point_type = 8,
                    points([[1,8],[5,8],[9,8]]),
                    point_type = filled_diamant,
                    points([[1,9],[5,9],[9,9]]) )$
```

points_joined Graphic option

Default value: `false`

When `points_joined` is `true`, points are joined by lines; when `false`, isolated points are drawn. A third possible value for this graphic option is `impulses`; in such case, vertical segments are drawn from points to the x-axis (2D) or to the xy-plane (3D).

This option affects the following graphic objects:

- `gr2d`: `points`.
- `gr3d`: `points`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(xrange        = [0,10],
             yrange        = [0,4],
             point_size    = 3,
             point_type    = up_triangle,
             color         = blue,
             points([[1,1],[5,1],[9,1]]),
             points_joined = true,
             point_type    = square,
             line_type     = dots,
             points([[1,2],[5,2],[9,2]]),
             point_type    = circle,
             color         = red,
             line_width    = 7,
             points([[1,3],[5,3],[9,3]]) )$
```

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`.

transparent

Graphic option

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]]) )$
```

border

Graphic option

Default value: `true`

If `border` is `true`, borders of polygons are painted according to `line_type` and `line_width`.

This option affects the following graphic objects:

- `gr2d`: `polygon`, `rectangle`, and `ellipse`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(color       = brown,
             line_width  = 8,
             polygon([[3,2],[7,2],[5,5]]),
             border      = false,
             fill_color  = blue,
             polygon([[5,2],[9,2],[7,5]]) )$
```

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_angle Graphic option

Default value: 45

`head_angle` indicates the angle, in degrees, between the arrow heads and the segment.

This option is relevant only for `vector` objects.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(xrange      = [0,10],
             yrange      = [0,9],
             head_length = 0.7,
             head_angle  = 10,
             vector([1,1],[0,6]),
             head_angle  = 20,
             vector([2,1],[0,6]),
```

```
                head_angle  = 30,
                vector([3,1],[0,6]),
                head_angle  = 40,
                vector([4,1],[0,6]),
                head_angle  = 60,
                vector([5,1],[0,6]),
                head_angle  = 90,
                vector([6,1],[0,6]),
                head_angle  = 120,
                vector([7,1],[0,6]),
                head_angle  = 160,
                vector([8,1],[0,6]),
                head_angle  = 180,
                vector([9,1],[0,6]) )$
```

See also `head_both`, `head_length`, and `head_type`.

head_type Graphic option

Default value: `filled`

`head_type` is used to specify how arrow heads are plotted. Possible values are: `filled` (closed and filled arrow heads), `empty` (closed but not filled arrow heads), and `nofilled` (open arrow heads).

This option is relevant only for `vector` objects.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(xrange      = [0,12],
             yrange      = [0,10],
             head_length = 1,
             vector([0,1],[5,5]), /* default type */
             head_type = 'empty,
             vector([3,1],[5,5]),
             head_type = 'nofilled,
             vector([6,1],[5,5]))$
```

See also `head_both`, `head_angle`, and `head_length`.

unit_vectors Graphic option

Default value: `false`

If `unit_vectors` is `true`, vectors are plotted with module 1. This is useful for plotting vector fields. If `unit_vectors` is `false`, vectors are plotted with its original length.

This option is relevant only for `vector` objects.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(xrange      = [-1,6],
             yrange      = [-1,6],
             head_length = 0.1,
             vector([0,0],[5,2]),
             unit_vectors = true,
```

```
                color        = red,
                vector([0,3],[5,2]))$
```

label_alignment Graphic option

Default value: `center`

`label_alignment` is used to specify where to write labels with respect to the given coordinates. Possible values are: `center`, `left`, and `right`.

This option is relevant only for `label` objects.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(xrange          = [0,10],
             yrange          = [0,10],
             points_joined   = true,
             points([[5,0],[5,10]]),
             color           = blue,
             label(["Centered alignment (default)",5,2]),
             label_alignment = 'left,
             label(["Left alignment",5,5]),
             label_alignment = 'right,
             label(["Right alignment",5,8]))$
```

See also `label_orientation`, and `color`.

label_orientation Graphic option

Default value: `horizontal`

`label_orientation` is used to specify orientation of labels. Possible values are: `horizontal`, and `vertical`.

This option is relevant only for `label` objects.

Example:

In this example, a dummy point is added to get an image. Package `draw` needs always data to draw an scene.

```
(%i1) load(draw)$
(%i2) draw2d(xrange     = [0,10],
             yrange     = [0,10],
             point_size = 0,
             points([[5,5]]),
             color      = navy,
             label(["Horizontal orientation (default)",5,2]),
             label_orientation = 'vertical,
             color             = "#654321",
             label(["Vertical orientation",1,5]))$
```

See also `label_alignment` and `color`.

color Graphic option

Default value: `"black"`

`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 componentes in hexadecimal code are introduced in the form `"#rrggbb"`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(explicit(x^2,x,-1,1), /* default is black */
             color = "red",
             explicit(0.5 + x^2,x,-1,1),
             color = blue,
             explicit(1 + x^2,x,-1,1),
             color = "light-blue", /* double quotes if - is used */
             explicit(1.5 + x^2,x,-1,1),
             color = "#23ab0f",
             label(["This is a label",0,1.2])  )$
```

See also `fill_color`.

fill_color Graphic option

Default value: `"red"`

`fill_color` specifies the color for filling polygons and 2d `explicit` functions.

See `color` to learn how colors are specified.

fill_density Graphic option

Default value: 0

`fill_density` is a number between 0 and 1 that specifies the intensity of the `fill_color` in `bars` objects.

See `bars` for examples.

line_width Graphic option

Default value: 1

`line_width` is the width of plotted lines. Its value must be a positive number.

This option affects the following graphic objects:

- gr2d: points, polygon, rectangle, ellipse, vector, explicit, implicit, parametric and polar.
- gr3d: points and parametric.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(explicit(x^2,x,-1,1), /* default width */
             line_width = 5.5,
             explicit(1 + x^2,x,-1,1),
             line_width = 10,
             explicit(2 + x^2,x,-1,1))$
```

See also line_type.

line_type Graphic option

Default value: solid

line_type indicates how lines are displayed; possible values are solid and dots.

This option affects the following graphic objects:

- gr2d: points, polygon, rectangle, ellipse, vector, explicit, implicit, parametric and polar.
- gr3d: points, explicit, parametric and parametric_surface.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(line_type = dots,
             explicit(1 + x^2,x,-1,1),
             line_type = solid, /* default */
             explicit(2 + x^2,x,-1,1))$
```

See also line_width.

nticks Graphic option

Default value: 30

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) )$
```

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.

key Graphic option

Default value: `""` (empty string)

`key` is the name of a function in the legend. If `key` is an empty string, no key is assigned to the function.

This option affects the following graphic objects:

- `gr2d`: `points`, `polygon`, `rectangle`, `ellipse`, `vector`, `explicit`, `implicit`, `parametric`, and `polar`.
- `gr3d`: `points`, `explicit`, `parametric`, and `parametric_surface`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(key   = "Sinus",
             explicit(sin(x),x,0,10),
             key   = "Cosinus",
             color = red,
             explicit(cos(x),x,0,10) )$
```

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`.

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`.

surface_hide Graphic option

Default value: `false`

If `surface_hide` is `true`, hidden parts are not plotted in 3d surfaces.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw(columns=2,
           gr3d(explicit(exp(sin(x)+cos(x^2)),x,-3,3,y,-3,3)),
           gr3d(surface_hide = true,
                explicit(exp(sin(x)+cos(x^2)),x,-3,3,y,-3,3)) )$
```

contour Graphic option

Default value: `none`

Option `contour` enables the user to select where to plot contour lines. Possible values are:

- `none`: no contour lines are plotted.
- `base`: contour lines are projected on the xy plane.
- `surface`: contour lines are plotted on the surface.
- `both`: two contour lines are plotted: on the xy plane and on the surface.
- `map`: contour lines are projected on the xy plane, and the view point is set just in the vertical.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
             contour_levels = 15,
             contour        = both,
             surface_hide   = true) $
```

contour_levels Graphic option

Default value: 5

This graphic option controls the way contours are drawn. `contour_levels` can be set to a positive integer number, a list of three numbers or an arbitrary set of numbers:

- When option `contour_levels` is bounded to positive integer n, n contour lines will be drawn at equal intervals. By default, five equally spaced contours are plotted.

- When option `contour_levels` is bounded to a list of length three of the form `[lowest,s,highest]`, contour lines are plotted from `lowest` to `highest` in steps of `s`.
- When option `contour_levels` is bounded to a set of numbers of the form `{n1, n2, ...}`, contour lines are plotted at values `n1`, `n2`, ...

Since this is a global graphics option, its position in the scene description does not matter.

Examples:

Ten equally spaced contour lines. The actual number of levels can be adjusted to give simple labels.

```
(%i1) load(draw)$
(%i2) draw3d(color = green,
             explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
             contour_levels = 10,
             contour        = both,
             surface_hide   = true) $
```

From -8 to 8 in steps of 4.

```
(%i1) load(draw)$
(%i2) draw3d(color = green,
             explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
             contour_levels = [-8,4,8],
             contour        = both,
             surface_hide   = true) $
```

Isolines at levels -7, -6, 0.8 and 5.

```
(%i1) load(draw)$
(%i2) draw3d(color = green,
             explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
             contour_levels = {-7, -6, 0.8, 5},
             contour        = both,
             surface_hide   = true) $
```

See also `contour`.

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)$
```

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.

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.

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.

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.

gr2d (*graphic option, ..., graphic object, ...*) Scene constructor

Function `gr2d` builds an object describing a 2D scene. Arguments are *graphic options* and *graphic objects*. This scene is interpreted sequentially: *graphic options* affect those *graphic objects* placed on its right. Some *graphic options* affect the global appearence of the scene.

This is the list of *graphic objects* available for scenes in two dimensions: `points`, `polygon`, `rectangle`, `bars`, `ellipse`, `label`, `vector`, `explicit`, `implicit`, `polar`, `parametric`, `image` and `geomap`.

See also the following global *graphic options*: `xrange`, `yrange`, `logx`, `logy`, `terminal`, `grid`, `title`, `xlabel`, `ylabel`, `xtics`, `ytics`, `xtics_rotate`, `ytics_rotate`, `xtics_axis`, `ytics_axis`, `xaxis`, `yaxis`, `xaxis_width`, `yaxis_width`, `xaxis_type`, `yaxis_type`, `xaxis_color`, `yaxis_color`, `xy_file`, `file_name`, `pic_width`, `pic_height`, `eps_width`, `eps_height`, `user_preamble`, `axis_bottom`, `axis_left`, `axis_top` and `axis_right`.

To make use of this function, write first `load(draw)`.

gr3d (*graphic option, ..., graphic object, ...*) Scene constructor

Function `gr3d` builds an object describing a 3d scene. Arguments are *graphic options* and *graphic objects*. This scene is interpreted sequentially: *graphic options* affect those

graphic objects placed on its right. Some *graphic options* affect the global appearence of the scene.

This is the list of *graphic objects* available for scenes in three dimensions: `points`, `label`, `vector`, `explicit`, `implicit`, `parametric`, `parametric_surface` and `geomap`.

See also the following global *graphic options*: `xrange`, `yrange`, `zrange`, `logx`, `logy`, `logz`, `terminal`, `grid`, `title`, `xlabel`, `ylabel`, `zlabel`, `xtics`, `ytics`, `ztics`, `xtics_rotate`, `ytics_rotate`, `ztics_rotate`, `xtics_axis`, `ytics_axis`, `ztics_axis`, `xaxis`, `yaxis`, `zaxis`, `xaxis_width`, `yaxis_width`, `zaxis_width`, `xaxis_type`, `yaxis_type`, `zaxis_type`, `xaxis_color`, `yaxis_color`, `zaxis_color`, `xy_file`, `user_preamble`, `axis_bottom`, `axis_left`, `axis_top`, `file_name`, `pic_width`, `pic_height`, `eps_width`, `eps_height`, `axis_right`, `rot_vertical`, `rot_horizontal`, `axis_3d`, `xu_grid`, `yv_grid`, `surface_hide`, `contour`, `contour_levels`, `palette`, `colorbox` and `enhanced3d`.

To make use of this function, write first `load(draw)`.

points ([[*x1*,*y1*], [*x2*,*y2*],...]) Graphic object
points ([*x1*,*x2*,...], [*y1*,*y2*,...]) Graphic object
points ([*y1*,*y2*,...]) Graphic object
points ([[*x1*,*y1*,*z1*], [*x2*,*y2*,*z2*],...]) Graphic object
points ([*x1*,*x2*,...], [*y1*,*y2*,...], [*z1*,*z2*,...]) Graphic object
points (*matrix*) Graphic object

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`.

2D

`points ([`*x1*`,`*y1*`], [`*x2*`,`*y2*`],...])` or `points ([`*x1*`,`*x2*`,...], [`*y1*`,`*y2*`,...])` plots points `[x1,y1]`, `[x2,y2]`, etc. If abscissas are not given, they are set to consecutive positive integers, so that `points ([`*y1*`,`*y2*`,...])` draws points `[1,`*y1*`]`, `[2,`*y2*`]`, etc. If *matrix* is a two-column or two-row matrix, `points (`*matrix*`)` draws the associated points. If *matrix* is a one-column or one-row matrix, abscissas are assigned automatically.

Example:

```
(%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]))$
```

```
(%i1) load(draw)$
(%i2) draw2d(
        points_joined = impulses,
        line_width    = 2,
        color         = red,
        points(makelist([random(20),random(50)],k,1,10)))$
```

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.

Examples:

One tridimensional sample,

```
(%i1) load(draw)$
(%i2) load (numericalio)$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) draw3d(title = "Daily average wind speeds",
             point_size = 2,
             points(args(submatrix (s2, 4, 5))) )$
```

Two tridimensional samples,

```
(%i1) load(draw)$
(%i2) load (numericalio)$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) draw3d(
         title = "Daily average wind speeds. Two data sets",
         point_size = 2,
         key        = "Sample from stations 1, 2 and 3",
         points(args(submatrix (s2, 4, 5))),
         point_type = 4,
         key        = "Sample from stations 1, 4 and 5",
         points(args(submatrix (s2, 2, 3))) )$
```

polygon ([[*x1*,*y1*], [*x2*,*y2*],...]) Graphic object
polygon ([*x1*,*x2*,...], [*y1*,*y2*,...]) Graphic object

Draws polygons in 2D.

2D

`polygon ([[`*x1*`,`*y1*`], [`*x2*`,`*y2*`],...])` or `polygon ([`*x1*`,`*x2*`,...], [`*y1*`,`*y2*`,...])`: plots on the plane a polygon with vertices `[`*x1*`,`*y1*`]`, `[`*x2*`,`*y2*`]`, etc..

This object is affected by the following *graphic options*: `transparent`, `fill_color`, `border`, `line_width`, `key`, `line_type` and `color`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(color      = "#e245f0",
             line_width = 8,
             polygon([[3,2],[7,2],[5,5]]),
             border     = false,
```

```
                    fill_color  = yellow,
                    polygon([[5,2],[9,2],[7,5]]) )$
```

rectangle ([*x1*,*y1*], [*x2*,*y2*]) Graphic object

Draws rectangles in 2D.

2D

`rectangle ([x1,y1], [x2,y2])` draws a rectangle with opposite vertices `[x1,y1]` and `[x2,y2]`.

This object is affected by the following *graphic options*: `transparent`, `fill_color`, `border`, `line_width`, `key`, `line_type` and `color`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(fill_color  = red,
             line_width  = 6,
             line_type   = dots,
             transparent = false,
             fill_color  = blue,
             rectangle([-2,-2],[8,-1]), /* opposite vertices */
             transparent = true,
             line_type   = solid,
             line_width  = 1,
             rectangle([9,4],[2,-1.5]),
             xrange      = [-3,10],
             yrange      = [-3,4.5] )$
```

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);
```

ellipse (*xc, yc, a, b, ang1, ang2*) Graphic object

Draws ellipses and circles in 2D.

2D

`ellipse (`*xc*`, `*yc*`, `*a*`, `*b*`, `*ang1*`, `*ang2*`)` plots an ellipse centered at `[`*xc*`, `*yc*`]` with horizontal and vertical semi axis *a* and *b*, respectively, from angle *ang1* 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] )$
```

label ([*string*,*x*,*y*],...) Graphic object
label ([*string*,*x*,*y*,*z*],...) Graphic object

Writes labels in 2D and 3D.

This object is affected by the following *graphic options*: `label_alignment`, `label_orientation` and `color`.

2D

`label([`*string*`,`*x*`,`*y*`])` writes the *string* at point `[`*x*`,`*y*`]`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(yrange = [0.1,1.4],
             color = "red",
             label(["Label in red",0,0.3]),
             color = "#0000ff",
             label(["Label in blue",0,0.6]),
             color = "light-blue",
             label(["Label in light-blue",0,0.9],
                   ["Another light-blue",0,1.2])  )$
```

3D

`label([`*string*`,`*x*`,`*y*`,`*z*`])` writes the *string* at point `[`*x*`,`*y*`,`*z*`]`.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(explicit(exp(sin(x)+cos(x^2)),x,-3,3,y,-3,3),
```

```
                    color = red,
                    label(["UP 1",-2,0,3], ["UP 2",1.5,0,4]),
                    color = blue,
                    label(["DOWN 1",2,0,-3]) )$
```

vector ([*x*,*y*], [*dx*,*dy*]) Graphic object
vector ([*x*,*y*,*z*], [*dx*,*dy*,*dz*]) Graphic object

Draws vectors in 2D and 3D.

This object is affected by the following *graphic options*: `head_both`, `head_length`, `head_angle`, `head_type`, `line_width`, `line_type`, `key` and `color`.

2D

`vector([`*x*`,`*y*`], [`*dx*`,`*dy*`])` plots vector `[`*dx*`,`*dy*`]` with origin in `[`*x*`,`*y*`]`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(xrange      = [0,12],
             yrange      = [0,10],
             head_length = 1,
             vector([0,1],[5,5]), /* default type */
             head_type = 'empty,
             vector([3,1],[5,5]),
             head_both = true,
             head_type = 'nofilled,
             line_type = dots,
             vector([6,1],[5,5]))$
```

3D

`vector([`*x*`,`*y*`,`*z*`], [`*dx*`,`*dy*`,`*dz*`])` plots vector `[`*dx*`,`*dy*`,`*dz*`]` with origin in `[`*x*`,`*y*`,`*z*`]`.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(color = cyan,
             vector([0,0,0],[1,1,1]/sqrt(3)),
             vector([0,0,0],[1,-1,0]/sqrt(2)),
             vector([0,0,0],[1,1,-2]/sqrt(6)) )$
```

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`, `key` and `color`.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(key   = "Gauss",
             color = "#a02c00",
             explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3),
             yv_grid     = 10,
             color = blue,
             key   = "Plane",
             explicit(x+y,x,-5,5,y,-5,5),
             surface_hide = true)$
```

See also `filled_func` for filled functions.

implicit (*fcn*,*x*,*xmin*,*xmax*,*y*,*ymin*,*ymax*) Graphic object
implicit (*fcn*,*x*,*xmin*,*xmax*,*y*,*ymin*,*ymax*,*z*,*zmin*,*zmax*) Graphic object

Draws implicit functions in 2D and 3D.

2D

`implicit(`*fcn*`,`*x*`,`*xmin*`,`*xmax*`,`*y*`,`*ymin*`,`*ymax*`)` plots the implicit function defined by *fcn*, with variable *x* taking values from *xmin* to *xmax*, and variable *y* taking values from *ymin* to *ymax*.

This object is affected by the following *graphic options*: `ip_grid`, `ip_grid_in`, `line_width`, `line_type`, `key` and `color`.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(terminal  = eps,
             grid      = true,
             line_type = solid,
             key       = "y^2=x^3-2*x+1",
             implicit(y^2=x^3-2*x+1, x, -4,4, y, -4,4),
             line_type = dots,
             key       = "x^3+y^3 = 3*x*y^2-x-1",
             implicit(x^3+y^3 = 3*x*y^2-x-1, x,-4,4, y,-4,4),
             title     = "Two implicit functions" )$
```

3D

`implicit (`*fcn*`,`*x*`,`*xmin*`,`*xmax*`, `*y*`,`*ymin*`,`*ymax*`, `*z*`,`*zmin*`,`*zmax*`)` plots the implicit surface defined by *fcn*, with variable *x* taking values from *xmin* to *xmax*, variable *y*

taking values from *ymin* to *ymax* and variable *z* taking values from *zmin* to *zmax*. This object implements the *marching cubes algorithm*.

This object is affected by the following *graphic options*: `x_voxel`, `y_voxel`, `z_voxel`, `line_width`, `line_type`, `key` 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);
```

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) )$
```

spherical (*radius,azi,minazi,maxazi,zen,minzen,maxzen*) Graphic object

Draws 3D functions defined in spherical coordinates.

3D

`spherical` (*radius*, *azi*, *minazi*, *maxazi*, *zen*, *minzen*, *maxzen*) plots function *radius*(*azi*, *zen*) defined in spherical coordinates, with *azimuth azi* taking values from *minazi* to *maxazi* and *zenith zen* taking values from *minzen* to *maxzen*.

This object is affected by the following *graphic options*: `xu_grid`, `yv_grid`, `line_type`, `key` and `color`.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(spherical(1,a,0,2*%pi,z,0,%pi))$
```

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))$
```

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` and `color`.

2D

`parametric` (*xfun*,*yfun*,*par*,*parmin*,*parmax*) plots parametric function [*xfun*,*yfun*], with parameter *par* taking values from *parmin* to *parmax*.

Example:

```
(%i1) load(draw)$
(%i2) draw2d(explicit(exp(x),x,-1,3),
             color = red,
             key   = "This is the parametric one!!",
             parametric(2*cos(rrr),rrr^2,rrr,0,2*%pi))$
```

3D

`parametric` (*xfun*,*yfun*,*zfun*,*par*,*parmin*,*parmax*) plots parametric curve [*xfun*,*yfun*,*zfun*], with parameter *par* taking values from *parmin* to *parmax*.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(explicit(exp(sin(x)+cos(x^2)),x,-3,3,y,-3,3),
             color = royalblue,
             parametric(cos(5*u)^2,sin(7*u),u-2,u,0,2),
             color      = turquoise,
             line_width = 2,
             parametric(t^2,sin(t),2+t,t,0,2),
             surface_hide = true,
             title = "Surface & curves" )$
```

image (*im*,*x0*,*y0*,*width*,*height*) Graphic object

Renders images in 2D.

2D

`image` (*im*,*x0*,*y0*,*width*,*height*) plots image *im* in the rectangular region from vertex (*x0*,*y0*) to (x0+*width*,y0+*height*) on the real plane. Argument *im* must be a matrix of real numbers, a matrix of vectors of length three or a *picture* object.

If *im* is a matrix of real numbers or a *levels picture* object, pixel values are interpreted according to graphic option `palette`, which is a vector of length three with components ranging from -36 to +36; each value is an index for a formula mapping the levels onto red, green and blue colors, respectively:

```
 0: 0               1: 0.5           2: 1
 3: x               4: x^2           5: x^3
 6: x^4             7: sqrt(x)       8: sqrt(sqrt(x))
 9: sin(90x)       10: cos(90x)     11: |x-0.5|
12: (2x-1)^2       13: sin(180x)    14: |cos(180x)|
15: sin(360x)      16: cos(360x)    17: |sin(360x)|
18: |cos(360x)|    19: |sin(720x)|  20: |cos(720x)|
21: 3x             22: 3x-1         23: 3x-2
24: |3x-1|         25: |3x-2|       26: (3x-1)/2
27: (3x-2)/2       28: |(3x-1)/2|   29: |(3x-2)/2|
30: x/0.32-0.78125                  31: 2*x-0.84
32: 4x;1;-2x+1.84;x/0.08-11.5
33: |2*x - 0.5|    34: 2*x          35: 2*x - 0.5
36: 2*x - 1
```

negative numbers mean negative colour component.

`palette = gray` and `palette = color` are short cuts for `palette = [3,3,3]` and `palette = [7,5,15]`, respectively.

If *im* is a matrix of vectors of length three or an *rgb picture* object, they are interpreted as red, green and blue color components.

Examples:

If *im* is a matrix of real numbers, pixel values are interpreted according to graphic option `palette`.

```
(%i1) load(draw)$
(%i2) im: apply(
           'matrix,
            makelist(makelist(random(200),i,1,30),i,1,30))$
(%i3) /* palette = color, default */
      draw2d(image(im,0,0,30,30))$
(%i4) draw2d(palette = gray, image(im,0,0,30,30))$
(%i5) draw2d(palette = [15,20,-4],
             colorbox=false,
             image(im,0,0,30,30))$
```

See also `colorbox`.

If *im* is a matrix of vectors of length three, they are interpreted as red, green and blue color components.

```
(%i1) load(draw)$
(%i2) im: apply(
            'matrix,
            makelist(
              makelist([random(300),
                        random(300),
                        random(300)],i,1,30),i,1,30))$
```

```
(%i3) draw2d(image(im,0,0,30,30))$
```

Package `draw` automatically loads package `picture`. In this example, a level picture object is built by hand and then rendered.

```
(%i1) load(draw)$
(%i2) im: make_level_picture([45,87,2,134,204,16],3,2);
(%o2)       picture(level, 3, 2, {Array:  #(45 87 2 134 204 16)})
(%i3) /* default color palette */
      draw2d(image(im,0,0,30,30))$
(%i4) /* gray palette */
      draw2d(palette = gray,
             image(im,0,0,30,30))$
```

An xpm file is read and then rendered.

```
(%i1) load(draw)$
(%i2) im: read_xpm("myfile.xpm")$
(%i3) draw2d(image(im,0,0,10,7))$
```

See also `make_level_picture`, `make_rgb_picture` and `read_xpm`.

URL `http://www.telefonica.net/web2/biomates/maxima/gpdraw/image` contains more elaborated examples.

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`.

geomap (*numlist*) Graphic object
geomap (*numlist,3Dprojection*) Graphic object

Draws cartographic maps in 2D and 3D.

2D

This function works together with global variable `boundaries_array`.

Argument *numlist* is a list containing numbers or lists of numbers. All these numbers must be integers greater or equal than zero, representing the components of global array `boundaries_array`.

Each component of `boundaries_array` is an array of floating point quantities, the coordinates of a polygonal segment or map boundary.

`geomap` (*numlist*) flattens its arguments and draws the associated boundaries in `boundaries_array`.

This object is affected by the following *graphic options*: `line_width`, `line_type` and `color`.

Examples:

A simple map defined by hand:

```
(%i1) load(draw)$
(%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 `draw` does not automatically load `worldmap`.

```
(%i1) load(draw)$
(%i2) load(worldmap)$
(%i3) c1: gr2d(geomap(Canada,United_States,
                      Mexico,Cuba))$
(%i4) c2: gr2d(geomap(Africa))$
(%i5) c3: gr2d(geomap(Oceania,China,Japan))$
(%i6) c4: gr2d(geomap(France,Portugal,Spain,
                      Morocco,Western_Sahara))$
(%i7) draw(columns  = 2,
           c1,c2,c3,c4)$
```

Package `worldmap` is also useful for plotting countries as polygons. In this case, graphic object `geomap` is no longer necessary and the `polygon` object is used instead. Since lists are now used and not arrays, maps rendering will be slower. See also `make_poly_country` and `make_poly_continent` to understand the following code.

```
(%i1) load(draw)$
(%i2) load(worldmap)$
```

```
(%i3) 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))$
(%i4) apply(draw2d, mymap)$
```

3D

`geomap` (*numlist*) projects map boundaries on the sphere of radius 1 centered at (0,0,0). It is possible to change the sphere or the projection type by using `geomap` (*numlist*,*3Dprojection*).

Available 3D projections:

- `[spherical_projection,`*x*`,`*y*`,`*z*`,`*r*`]`: projects map boundaries on the sphere of radius *r* centered at (*x*,*y*,*z*).

```
(%i1) load(draw)$
(%i2) load(worldmap)$
(%i3) draw3d(geomap(Australia), /* default projection */
             geomap(Australia,
                    [spherical_projection,2,2,2,3]))$
```

- `[cylindrical_projection,`*x*`,`*y*`,`*z*`,`*r*`,`*rc*`]`: re-projects spherical map boundaries on the cylinder of radius *rc* and axis passing through the poles of the globe of radius *r* centered at (*x*,*y*,*z*).

```
(%i1) load(draw)$
(%i2) load(worldmap)$
(%i3) draw3d(geomap([America_coastlines,Eurasia_coastlines],
                    [cylindrical_projection,2,2,2,3,4]))$
```

- `[conic_projection,`*x*`,`*y*`,`*z*`,`*r*`,`*alpha*`]`: re-projects spherical map boundaries on the cones of angle *alpha*, with axis passing through the poles of the globe of radius *r* centered at (*x*,*y*,*z*). Both the northern and southern cones are tangent to sphere.

```
(%i1) load(draw)$
(%i2) load(worldmap)$
(%i3) draw3d(geomap(World_coastlines,
                    [conic_projection,0,0,0,1,90]))$
```

See also `http://www.telefonica.net/web2/biomates/maxima/gpdraw/geomap` for more elaborated examples.

parametric_surface

Graphic object

(*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`, `key` and `color`.

Example:

```
(%i1) load(draw)$
(%i2) draw3d(title          = "Sea shell",
             xu_grid        = 100,
             yv_grid        = 25,
             rot_vertical   = 100,
             rot_horizontal = 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) )$
```

draw (*gr2d, ..., gr3d, ..., options, ...*) Function

Plots a series of scenes; its arguments are `gr2d` and/or `gr3d` objects, together with some options. By default, the scenes are put together in one column.

Function `draw` accepts the following global options: `terminal`, `columns`, `pic_width`, `pic_height`, `eps_width`, `eps_height`, `file_name` and `delay`.

Functions `draw2d` and `draw3d` are short cuts to be used when only one scene is required, in two or three dimensions, respectively.

To make use of this function, write first `load(draw)`.

Example:

```
(%i1) load(draw)$
(%i2) scene1: gr2d(title="Ellipse",
                   nticks=30,
                   parametric(2*cos(t),5*sin(t),t,0,2*%pi))$
(%i3) scene2: gr2d(title="Triangle",
                   polygon([4,5,7],[6,4,2]))$
(%i4) draw(scene1, scene2, columns = 2)$
```

The two draw sentences are equivalent:

```
(%i1) load(draw)$
(%i2) draw(gr3d(explicit(x^2+y^2,x,-1,1,y,-1,1)));
(%o2)                          [gr3d(explicit)]
(%i3) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1));
(%o3)                          [gr3d(explicit)]
```

An animated gif file:

```
(%i1) load(draw)$
(%i2) draw(
        delay     = 100,
        file_name = "zzz",
```

```
        terminal  = 'animated_gif,
        gr2d(explicit(x^2,x,-1,1)),
        gr2d(explicit(x^3,x,-1,1)),
        gr2d(explicit(x^4,x,-1,1)));
End of animation sequence
(%o2)          [gr2d(explicit), gr2d(explicit), gr2d(explicit)]
```

See also `gr2d`, `gr3d`, `draw2d` and `draw3d`..

draw2d (*option, graphic_object, ...*) Function

This function is a short cut for `draw(gr2d(`*options*`, ..., `*graphic_object*`, ...))`.

It can be used to plot a unique scene in 2d.

To make use of this function, write first `load(draw)`.

See also `draw` and `gr2d`.

draw3d (*option, graphic_object, ...*) Function

This function is a short cut for `draw(gr3d(`*options*`, ..., `*graphic_object*`, ...))`.

It can be used to plot a unique scene in 3d.

To make use of this function, write first `load(draw)`.

See also `draw` and `gr3d`.

48.3 Functions and Variables for pictures

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)})
```

picturep (*x*) Function

Returns `true` if the argument is a well formed image, and `false` otherwise.

picture_equalp (*x,y*) Function

Returns `true` in case of equal pictures, and `false` otherwise.

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)})
```

take_channel (*im,color*) Function

If argument *color* is `red`, `green` or `blue`, function `take_channel` returns the corresponding color channel of picture *im*. Example:

```
(%i1) load(draw)$
(%i2) red: make_level_picture(matrix([3,2],[7,260]));
(%o2)           picture(level, 2, 2, {Array:  #(3 2 7 255)})
(%i3) green: make_level_picture(matrix([54,23],[73,-9]));
(%o3)           picture(level, 2, 2, {Array:  #(54 23 73 0)})
(%i4) blue: make_level_picture(matrix([123,82],[45,32.5698]));
(%o4)          picture(level, 2, 2, {Array:  #(123 82 45 33)})
(%i5) make_rgb_picture(red,green,blue);
(%o5) picture(rgb, 2, 2,
              {Array:  #(3 54 123 2 23 82 7 73 45 255 0 33)})
(%i6) take_channel(%,'green);  /* simple quote!!! */
(%o6)           picture(level, 2, 2, {Array:  #(54 23 73 0)})
```

negative_picture (*pic*) Function

Returns the negative of a (*level* or *rgb*) picture.

rgb2level (*pic*) Function

Transforms an *rgb* picture into a *level* one by averaging the red, green and blue channels.

get_pixel (*pic,x,y*) Function

Returns pixel from picture. Coordinates x and y range from 0 to `width-1` and `height-1`, respectively.

read_xpm (*xpm_file*) Function

Reads a file in xpm and returns a picture object.

48.4 Functions and Variables for worldmap

region_boundaries (*x1,y1,x2,y2*) Function

Detects polygonal segments of global variable `boundaries_array` contained in the rectangle with vertices (*x1,y1*) -upper left- and (*x2,y2*) -bottom right-.

Example:

Returns segment numbers for plotting southern Italy.

```
(%i1) load(draw)$
(%i2) load(worldmap)$
(%i3) region_boundaries(10.4,41.5,20.7,35.4);
(%o3)                 [1846, 1863, 1864, 1881, 1888, 1894]
(%i4) draw2d(geomap(%))$
```

numbered_boundaries (*nlist*) Function

Draws a list of polygonal segments (boundaries), labeled by its numbers (`boundaries_array` coordinates). This is of great help when building new geographical entities.

Example:

Map of Europe labeling borders with their component number in `boundaries_array`.

```
(%i1) load(draw)$
(%i2) load(worldmap)$
(%i3) european_borders:
           region_boundaries(-31.81,74.92,49.84,32.06)$
(%i4) numbered_boundaries(european_borders)$
```

make_polygon (*nlist*) Function

Returns a `polygon` object from boundary indices. Argument *nlist* is a list of components of `boundaries_array`.

Example:

Bhutan is defined by boundary numbers 171, 173 and 1143, so that `make_polygon([171,173,1143])` appends arrays of coordinates `boundaries_array[171]`, `boundaries_array[173]` and `boundaries_array[1143]` and returns a `polygon` object suited to be plotted by `draw`. To avoid an error message, arrays must be compatible in the sense that any two consecutive arrays have two coordinates in the

extremes in common. In this example, the two first components of `boundaries_array[171]` are equal to the last two coordinates of `boundaries_array[173]`, and the two first of `boundaries_array[173]` are equal to the two first of `boundaries_array[1143]`; in conclussion, boundary numbers 171, 173 and 1143 (in this order) are compatible and the colored polygon can be drawn.

```
(%i1) load(draw)$
(%i2) load(worldmap)$
(%i3) Bhutan;
(%o3)                          [[171, 173, 1143]]
(%i4) boundaries_array[171];
(%o4) {Array:
       #(88.750549 27.14727 88.806351 27.25305 88.901367 27.282221
         88.917877 27.321039)}
(%i5) boundaries_array[173];
(%o5) {Array:
       #(91.659554 27.76511 91.6008 27.66666 91.598022 27.62499
         91.631348 27.536381 91.765533 27.45694 91.775253 27.4161
         92.007751 27.471939 92.11441 27.28583 92.015259 27.168051
         92.015533 27.08083 92.083313 27.02277 92.112183 26.920271
         92.069977 26.86194 91.997192 26.85194 91.915253 26.893881
         91.916924 26.85416 91.8358 26.863331 91.712479 26.799999
         91.542191 26.80444 91.492188 26.87472 91.418854 26.873329
         91.371353 26.800831 91.307457 26.778049 90.682457 26.77417
         90.392197 26.903601 90.344131 26.894159 90.143044 26.75333
         89.98996 26.73583 89.841919 26.70138 89.618301 26.72694
         89.636093 26.771111 89.360786 26.859989 89.22081 26.81472
         89.110237 26.829161 88.921631 26.98777 88.873016 26.95499
         88.867737 27.080549 88.843307 27.108601 88.750549
         27.14727)}
(%i6) boundaries_array[1143];
(%o6) {Array:
       #(91.659554 27.76511 91.666924 27.88888 91.65831 27.94805
         91.338028 28.05249 91.314972 28.096661 91.108856 27.971109
         91.015808 27.97777 90.896927 28.05055 90.382462 28.07972
         90.396088 28.23555 90.366074 28.257771 89.996353 28.32333
         89.83165 28.24888 89.58609 28.139999 89.35997 27.87166
         89.225517 27.795 89.125793 27.56749 88.971077 27.47361
         88.917877 27.321039)}
(%i7) Bhutan_polygon: make_polygon([171,173,1143])$
(%i8) draw2d(Bhutan_polygon)$
```

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(draw)$
(%i2) load(worldmap)$
(%i3) make_poly_country(India)$
```

```
(%i4) apply(draw2d, %)$
```

make_poly_continent (*continent_name*) Function
make_poly_continent (*country_list*) Function

Makes the necessary polygons to draw a colored continent or a list of countries.

Example:

```
(%i1) load(draw)$
(%i2) load(worldmap)$
(%i3) /* A continent */
      make_poly_continent(Africa)$
(%i4) apply(draw2d, %)$
(%i5) /* A list of countries */
      make_poly_continent([Germany,Denmark,Poland])$
(%i6) apply(draw2d, %)$
```

49 dynamics

49.1 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 explicitely the two independent variables, and the horizontal range for `orbits` no longer requires a step size; the range should only specify the variable name, and the minimum and maximum values; the number of steps can now be changed with the option *nticks*.

49.2 Functions and Variables for dynamics

chaosgame ([[*x1*, *y1*]...[*xm*, *ym*]], [*x0*, *y0*], *b*, *n*, ..., *options*, ...); Function

Implements the so-called chaos game: the initial point (*x0*, *y0*) is plotted and then one of the *m* points [*x1*, *y1*]...[*xm*, *ym*] will be selected at random. The next point plotted will be on the segment from the previous point plotted to the point chosen randomly, at a distance from the random point which will be *b* times that segment's length. The procedure is repeated *n* times.

evolution (*F*, *y0*, *n*, ..., *options*, ...); Function

Draws *n+1* points in a two-dimensional graph, where the horizontal coordinates of the points are the integers 0, 1, 2, ..., *n*, and the vertical coordinates are the corresponding values *y(n)* of the sequence defined by the recurrence relation

$$y_{n+1} = F(y_n)$$

With initial value *y(0)* equal to *y0*. *F* must be an expression that depends only on one variable (in the example, it depend on *y*, but any other variable can be used), *y0* must be a real number and *n* must be a positive integer.

evolution2d ([*F*, *G*], [*u*, *v*], [*u0*, *y0*], *n*, ..., *options*, ...); Function

Shows, in a two-dimensional plot, the first *n+1* points in the sequence of points defined by the two-dimensional discrete dynamical system with recurrence relations

$$\begin{cases} u_{n+1} = F(u_n, v_n) \\ v_{n+1} = G(u_n, v_n) \end{cases}$$

With initial values *u0* and *v0*. *F* and *G* must be two expressions that depend only on two variables, *u* and *v*, which must be named explicitely in a list.

ifs ([*r1*, ..., *rm*], [*A1*, ..., *Am*], [[*x1*, *y1*], ..., [*xm*, *ym*]], [*x0*, *y0*], *n*, ..., *options*, ...); Function

Implements the Iterated Function System method. This method is similar to the method described in the function `chaosgame`, but instead of shrinking the segment from the current point to the randomly chosen point, the 2 components of that segment will be multiplied by the 2 by 2 matrix *Ai* that corresponds to the point chosen randomly.

The random choice of one of the *m* attractive points can be made with a non-uniform probability distribution defined by the weights *r1*,...,*rm*. Those weights are given in cumulative form; for instance if there are 3 points with probabilities 0.2, 0.5 and 0.3, the weights *r1*, *r2* and *r3* could be 2, 7 and 10.

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 o 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 Mandelbrot 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.

orbits (*F, y0, n1, n2,* [*x, x0, xf, xstep*], *...options...*); Function

Draws the orbits diagram for a family of one-dimensional discrete dynamical systems, with one parameter *x*; that kind of diagram is used to study the bifurcations of a one-dimensional discrete system.

The function *F(y)* defines a sequence with a starting value of *y0*, as in the case of the function `evolution`, but in this case that function will also depend on a parameter *x* that will take values in the interval from *x0* to *xf* with increments of *xstep*. Each value used for the parameter *x* is shown on the horizontal axis. The vertical axis will show the *n2* values of the sequence *y(n1+1)*,..., *y(n1+n2+1)* obtained after letting the sequence evolve *n1* iterations.

rk (*ODE, var, initial, domain*) Function

rk ([*ODE1,...,ODEm*], [*v1,...,vm*], [*init1,...,initm*], *domain*) Function

The first form solves numerically one first-order ordinary differential equation, and the second form solves a system of m of those equations, using the 4th order Runge-Kutta method. *var* represents the dependent variable. *ODE* must be an expression that depends only on the independent and dependent variables and defines the derivative of the dependent variable with respect to the independent variable.

The independent variable is specified with `domain`, which must be a list of four elements as, for instance:

```
[t, 0, 10, 0.1]
```

the first element of the list identifies the independent variable, the second and third elements are the initial and final values for that variable, and the last element sets the increments that should be used within that interval.

If *m* equations are going to be solved, there should be *m* dependent variables *v1*, *v2*, ..., *vm*. The initial values for those variables will be *init1*, *init2*, ..., *initm*. There will still be just one independent variable defined by `domain`, as in the previous case. *ODE1*, ..., *ODEm* are the expressions that define the derivatives of each dependent variable in terms of the independent variable. The only variables that may appear in those expressions are the independent variable and any of the dependent variables. It is important to give the derivatives *ODE1*, ..., *ODEm* in the list in exactly the same order used for the dependent variables; for instance, the third element in the list will be interpreted as the derivative of the third dependent variable.

The program will try to integrate the equations from the initial value of the independent variable until its last value, using constant increments. If at some step one of the dependent variables takes an absolute value too large, the integration will be interrupted at that point. The result will be a list with as many elements as the number of iterations made. Each element in the results list is itself another list with *m*+1 elements: the value of the independent variable, followed by the values of the dependent variables corresponding to that point.

staircase (*F*, *y0*, *n*, *...options...*); Function

Draws a staircase diagram for the sequence defined by the recurrence relation

$$y_{n+1} = F(y_n)$$

The interpretation and allowed values of the input parameters is the same as for the function `evolution`. A staircase diagram consists of a plot of the function *F(y)*, together with the line *G(y)* = *y*. A vertical segment is drawn from the point (*y0*, *y0*) on that line until the point where it intersects the function *F*. From that point a horizontal segment is drawn until it reaches the point (*y1*, *y1*) on the line, and the procedure is repeated *n* times until the point (*yn*, *yn*) is reached.

Options

Each option is a list of two or more items. The first item is the name of the option, and the remainder comprises the arguments for the option.

The options accepted by the functions `evolution`, `evolution2d`, `staircase`, `orbits`, `ifs` and `chaosgame` are the same as the options for `plot2d`. In addition to those options, `orbits` accepts and extra option *pixels* that sets up the maximum number of different points that will be represented in the vertical direction.

The following options are accepted by the functions `julia` and `mandelbrot`:

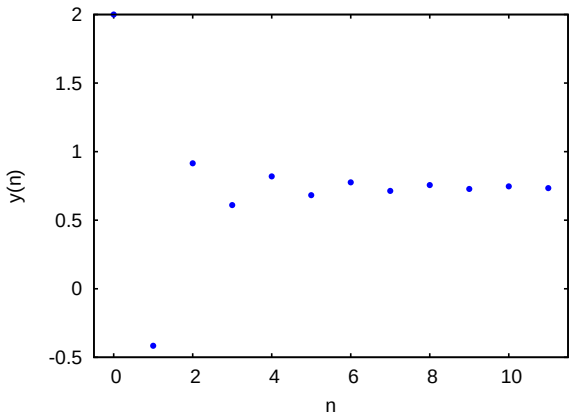
- *size* takes either one or two arguments. If only one argument is given, the width and height of the graphic file created will be equal to that value, in pixels. If two arguments are given, they will define the width and height. The default value is 400 pixels for both the width and height. If the two values are not equal, the set will appear distorted.
- *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.
- *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.
- *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 *huerange*.
- *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.

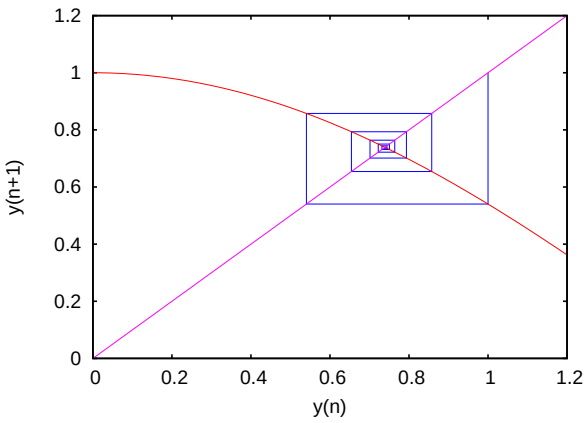
- *value* sets the value of the colors used for points not belonging to the set. It must be between 0 and 1; the higher the value, the brighter the colors. The default is 0.96
- *color* must be followed by three parameters that define the hue, saturation and value, for the color used to represent the 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)),...

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

(%i2) evolution(cos(y), 2, 11);

(%i3) staircase(cos(y), 1, 11, [y, 0, 1.2]);
```



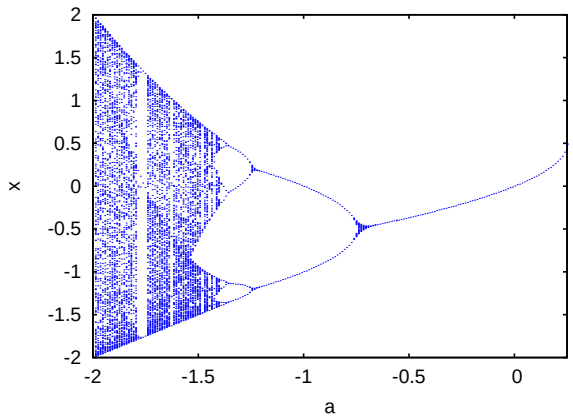


If your system is slow, you'll have to reduce the number of iterations in the following examples. And if the dots appear too small in your monitor, you might want to try a different style, such as `[`*style*`,[`*points*`,0.8]]`.

Orbits diagram for the quadratic map, with a parameter *a*.

$$x_{n+1} = a + x_n^2$$

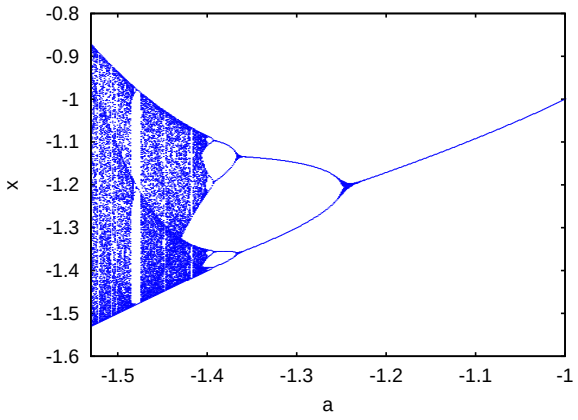
```
(%i4) orbits(x^2+a, 0, 50, 200, [a, -2, 0.25], [style, dots]);
```



To enlarge the region around the lower bifurcation near x = -1.25 use:

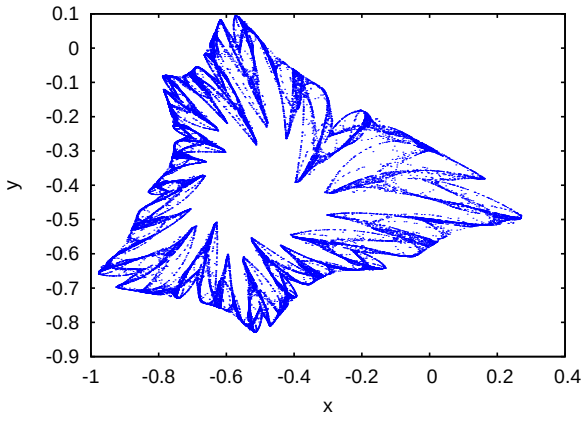
```
(%i5) orbits(x+y^2, 0, 100, 400, [a,-1,-1.53], [x,-1.6,-0.8],
```

```
[nticks, 400], [style,dots]);
```



Evolution of a two-dimensional system that leads to a fractal:

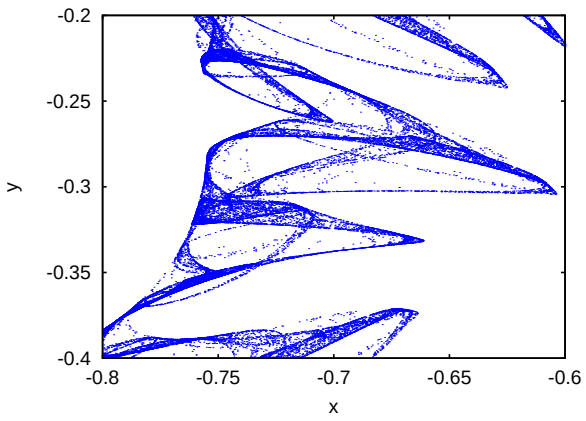
```
(%i6) f: 0.6*x*(1+2*x)+0.8*y*(x-1)-y^2-0.9$

(%i7) g: 0.1*x*(1-6*x+4*y)+0.1*y*(1+9*y)-0.4$

(%i8) evolution2d([f,g], [x,y], [-0.5,0], 50000, [style,dots]);
```



And an enlargement of a small region in that fractal:

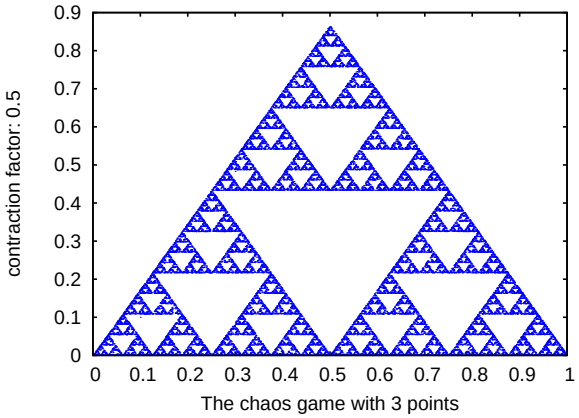
```
(%i9) evolution2d([f,g], [x,y], [-0.5,0], 300000, [x,-0.8,-0.6],
```

```
[y,-0.4,-0.2], [style, dots]);
```



A plot of Sierpinsky's triangle, obtained with the chaos game:

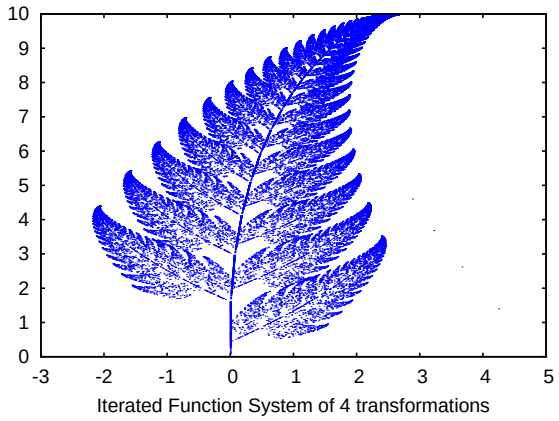
```
(%i9) chaosgame([[0, 0], [1, 0], [0.5, sqrt(3)/2]], [0.1, 0.1], 1/2,
                30000, [style, dots]);
```



Barnsley's fern, obtained with an Iterated Function System:

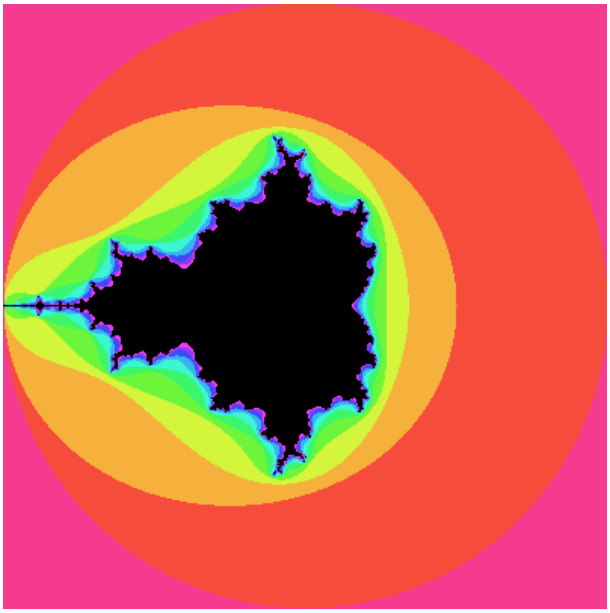
```
(%i10) a1: matrix([0.85,0.04],[-0.04,0.85])$

(%i11) a2: matrix([0.2,-0.26],[0.23,0.22])$

(%i12) a3: matrix([-0.15,0.28],[0.26,0.24])$

(%i13) a4: matrix([0,0],[0,0.16])$

(%i14) p1: [0,1.6]$

(%i15) p2: [0,1.6]$

(%i16) p3: [0,0.44]$
```

```
(%i17) p4: [0,0]$

(%i18) w: [85,92,99,100]$

(%i19) ifs(w, [a1,a2,a3,a4], [p1,p2,p3,p4], [5,0], 50000, [style,dots]);
```



To create a file named *dynamics9.xpm* with a graphical representation of the Mandelbrot set, with 12 colors, use:

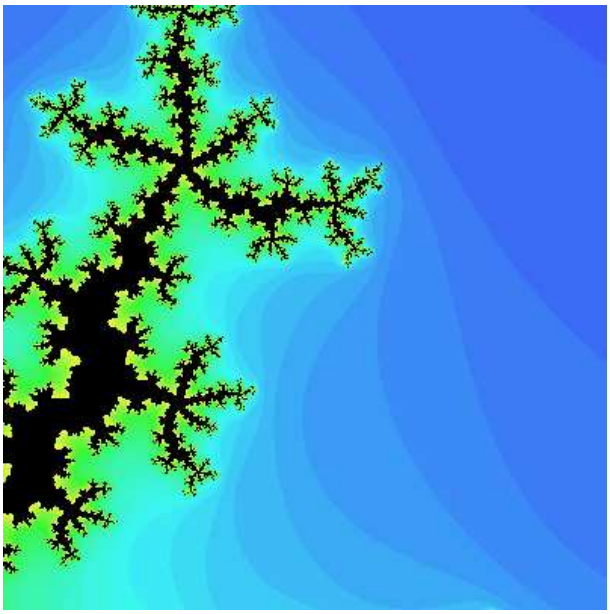
```
mandelbrot([filename,"dynamics9"])$
```



and the Julia set for the number (-0.55 + i 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.



To solve numerically the differential equation

$$\frac{dx}{dt} = t - x^2$$

With initial value x(t=0) = 1, in the interval of t from 0 to 8 and with increments of 0.1 for t, use:

```
(%i20) results: rk(t-x^2,x,1,[t,0,8,0.1])$
```

the results will be saved in the list `results`.

To solve numerically the system:

$$\begin{cases} \frac{dx}{dt} = 4 - x^2 - 4y^2 \\ \frac{dy}{dt} = y^2 - x^2 + 1 \end{cases}$$

for t between 0 and 4, and with values of -1.25 and 0.75 for x and y at t=0:

```
(%i21) sol: rk([4-x^2-4*y^2,y^2-x^2+1],[x,y],[-1.25,0.75],[t,0,4,0.02])$
```

50 f90

50.1 Functions and Variables for f90

f90 (*expr*) Function

The `f90` command is an update to the original maxima `fortran` command. The primary difference is the way long lines are broken.

In the next example, notice how the `fortran` command breaks lines within symbols. The `f90` command never breaks within a symbol.

```
(%i1) load("f90")$

(%i2) expr:expand((xxx+yyy+7)^4);
         4            3         3        2    2
(%o2) yyy  + 4 xxx yyy  + 28 yyy  + 6 xxx  yyy
             2          2        3             2
 + 84 xxx yyy  + 294 yyy  + 4 xxx  yyy + 84 xxx  yyy
                               4         3          2
 + 588 xxx yyy + 1372 yyy + xxx  + 28 xxx  + 294 xxx
 + 1372 xxx + 2401
(%i3) fortran(expr);
      yyy**4+4*xxx*yyy**3+28*yyy**3+6*xxx**2*yyy**2+84*xxx*yyy**2+294*yy
     1   y**2+4*xxx**3*yyy+84*xxx**2*yyy+588*xxx*yyy+1372*yyy+xxx**4+28*
     2   xxx**3+294*xxx**2+1372*xxx+2401
(%o3)                         done
(%i4) f90(expr);
yyy**4+4*xxx*yyy**3+28*yyy**3+6*xxx**2*yyy**2+84*xxx*yyy**2+294* &
     yyy**2+4*xxx**3*yyy+84*xxx**2*yyy+588*xxx*yyy+1372*yyy+xxx** &
     4+28*xxx**3+294*xxx**2+1372*xxx+2401
(%o4)                         done
```

The `f90` implementation was done as a quick hack. It is not a necessarily a good example upon which to base other language translations.

To use this function write first `load("f90")`.

51 ggf

51.1 Functions and Variables for ggf

GGFINFINITY Option variable

Default value: 3

This is an option variable for function `ggf`.

When computing the continued fraction of the generating function, a partial quotient having a degree (strictly) greater than *GGFINFINITY* will be discarded and the current convergent will be considered as the exact value of the generating function; most often the degree of all partial quotients will be 0 or 1; if you use a greater value, then you should give enough terms in order to make the computation accurate enough.

See also `ggf`.

GGFCFMAX Option variable

Default value: 3

This is an option variable for function `ggf`.

When computing the continued fraction of the generating function, if no good result has been found (see the *GGFINFINITY* flag) after having computed *GGFCFMAX* partial quotients, the generating function will be considered as not being a fraction of two polynomials and the function will exit. Put freely a greater value for more complicated generating functions.

See also `ggf`.

ggf (*l*) Function

Compute the generating function (if it is a fraction of two polynomials) of a sequence, its first terms being given. *l* is a list of numbers.

The solution is returned as a fraction of two polynomials. If no solution has been found, it returns with `done`.

This function is controlled by global variables *GGFINFINITY* and *GGFCFMAX*. See also *GGFINFINITY* and *GGFCFMAX*.

To use this function write first `load("ggf")`.

52 graphs

52.1 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)`.

52.2 Functions and Variables for graphs

52.2.1 Building graphs

create_graph (*v_list*, *e_list*) Function
create_graph (*n*, *e_list*) Function
create_graph (*v_list*, *e_list*, *directed*) Function

Creates a new graph on the set of vertices *v_list* and with edges *e_list*.

v_list is a list of vertices (`[v1, v2,..., vn]`) or a list of vertices together with vertex labels (`[[v1,l1], [v2,l2],..., [vn,ln]]`).

n is the number of vertices. Vertices will be identified by integers from 0 to n-1.

e_list is a list of edges (`[e1, e2,..., em]`) or a list of edges together with edge-weights (`[[e1, w1], ..., [em, wm]]`).

If *directed* is not `false`, a directed graph will be returned.

Example 1: create a cycle on 3 vertices:

```
(%i1) load (graphs)$
(%i2) g : create_graph([1,2,3], [[1,2], [2,3], [1,3]])$
(%i3) print_graph(g)$
Graph on 3 vertices with 3 edges.
Adjacencies:
  3 :  1  2
  2 :  3  1
  1 :  3  2
```

Example 2: create a cycle on 3 vertices with edge weights:

```
(%i1) load (graphs)$
(%i2) g : create_graph([1,2,3], [[[1,2], 1.0], [[2,3], 2.0],
                                [[1,3], 3.0]])$
(%i3) print_graph(g)$
Graph on 3 vertices with 3 edges.
Adjacencies:
  3 :  1  2
  2 :  3  1
  1 :  3  2
```

Example 3: create a directed graph:

```
(%i1) load (graphs)$
(%i2) d : create_graph(
        [1,2,3,4],
        [
         [1,3], [1,4],
         [2,3], [2,4]
        ],
        'directed = true)$
(%i3) print_graph(d)$
Digraph on 4 vertices with 4 arcs.
Adjacencies:
  4 :
  3 :
  2 :  4  3
  1 :  4  3
```

copy_graph (*g*) Function

Returns a copy of the graph *g*.

circulant_graph (*n*, *d*) Function

Returns the circulant graph with parameters *n* and *d*.

Example:

```
(%i1) load (graphs)$
(%i2) g : circulant_graph(10, [1,3])$
(%i3) print_graph(g)$
Graph on 10 vertices with 20 edges.
Adjacencies:
  9 :  2  6  0  8
  8 :  1  5  9  7
  7 :  0  4  8  6
  6 :  9  3  7  5
  5 :  8  2  6  4
  4 :  7  1  5  3
  3 :  6  0  4  2
  2 :  9  5  3  1
  1 :  8  4  2  0
  0 :  7  3  9  1
```

clebsch_graph () Function
Returns the Clebsch graph.

complement_graph (*g*) Function
Returns the complement of the graph *g*.

complete_bipartite_graph (*n*, *m*) Function
Returns the complete bipartite graph on *n*+*m* vertices.

complete_graph (*n*) Function
Returns the complete graph on *n* vertices.

cycle_digraph (*n*) Function
Returns the directed cycle on *n* vertices.

cycle_graph (*n*) Function
Returns the cycle on *n* vertices.

cube_graph (*n*) Function
Returns the *n*-dimensional cube.

dodecahedron_graph () Function
Returns the dodecahedron graph.

empty_graph (*n*) Function
Returns the empty graph on *n* vertices.

flower_snark (*n*) Function
Returns the flower graph on *4n* vertices.
Example:

```
(%i1) load (graphs)$
(%i2) f5 : flower_snark(5)$
(%i3) chromatic_index(f5);
(%o3)                          4
```

from_adjacency_matrix (*A*) Function
Returns the graph represented by its adjacency matrix *A*.

frucht_graph () Function
Returns the Frucht graph.

graph_product (*g1*, *g1*) Function
Returns the direct product of graphs *g1* and *g2*.
Example:

```
(%i1) load (graphs)$
(%i2) grid : graph_product(path_graph(3), path_graph(4))$
(%i3) draw_graph(grid)$
```

graph_union (*g1*, *g1*) Function
Returns the union (sum) of graphs *g1* and *g2*.

grid_graph (*n*, *m*) Function
Returns the *n* x *m* grid.

grotzch_graph () Function
Returns the Grotzch graph.

heawood_graph () Function
Returns the Heawood graph.

icosahedron_graph () Function
Returns the icosahedron graph.

induced_subgraph (*V*, *g*) Function
Returns the graph induced on the subset *V* of vertices of the graph *g*.

Example:

```
(%i1) load (graphs)$
(%i2) p : petersen_graph()$
(%i3) V : [0,1,2,3,4]$
(%i4) g : induced_subgraph(V, p)$
(%i5) print_graph(g)$
Graph on 5 vertices with 5 edges.
Adjacencies:
  4 :  3  0
  3 :  2  4
  2 :  1  3
  1 :  0  2
  0 :  1  4
```

line_graph (*g*) Function
Returns the line graph of the graph *g*.

make_graph (*vrt*, *f*) Function
make_graph (*vrt*, *f*, *oriented*) Function
Creates a graph using a predicate function *f*.

vrt is a list/set of vertices or an integer. If *vrt* is an integer, then vertices of the graph will be integers from 1 to *vrt*.

f is a predicate function. Two vertices *a* and *b* will be connected if `f(a,b)=true`.

If *directed* is not *false*, then the graph will be directed.

Example 1:

```
(%i1) load(graphs)$
(%i2) g : make_graph(powerset({1,2,3,4,5}, 2), disjointp)$
(%i3) is_isomorphic(g, petersen_graph());
```

```
(%o3)                          true
(%i4) get_vertex_label(1, g);
(%o4)                        {1, 2}
```

Example 2:

```
(%i1) load(graphs)$
(%i2) f(i, j) := is (mod(j, i)=0)$
(%i3) g : make_graph(20, f, directed=true)$
(%i4) out_neighbors(4, g);
(%o4)                     [8, 12, 16, 20]
(%i5) in_neighbors(18, g);
(%o5)                     [1, 2, 3, 6, 9]
```

mycielski_graph (*g*) Function

Returns the mycielskian graph of the graph *g*.

new_graph () Function

Returns the graph with no vertices and no edges.

path_digraph (*n*) Function

Returns the directed path on *n* vertices.

path_graph (*n*) Function

Returns the path on *n* vertices.

petersen_graph () Function
petersen_graph (*n*, *d*) Function

Returns the petersen graph *P*_{*n*,*d*}. The default values for *n* and *d* are `n=5` and `d=2`.

random_bipartite_graph (*a*, *b*, *p*) Function

Returns a random bipartite graph on `a+b` vertices. Each edge is present with probability *p*.

random_digraph (*n*, *p*) Function

Returns a random directed graph on *n* vertices. Each arc is present with probability *p*.

random_regular_graph (*n*) Function
random_regular_graph (*n*, *d*) Function

Returns a random *d*-regular graph on *n* vertices. The default value for *d* is `d=3`.

random_graph (*n*, *p*) Function

Returns a random graph on *n* vertices. Each edge is present with probability *p*.

random_graph1 (*n*, *m*) Function

Returns a random graph on *n* vertices and random *m* edges.

random_network (*n*, *p*, *w*) Function

Returns a random network on *n* vertices. Each arc is present with probability *p* and has a weight in the range `[0,w]`. The function returns a list `[network, source, sink]`.

Example:

```
(%i1) load (graphs)$
(%i2) [net, s, t] : random_network(50, 0.2, 10.0);
(%o2)                     [DIGRAPH, 50, 51]
(%i3) max_flow(net, s, t)$
(%i4) first(%);
(%o4)                     27.65981397932507
```

random_tournament (*n*) Function

Returns a random tournament on *n* vertices.

random_tree (*n*) Function

Returns a random tree on *n* vertices.

tutte_graph () Function

Returns the Tutte graph.

underlying_graph (*g*) Function

Returns the underlying graph of the directed graph *g*.

wheel_graph (*n*) Function

Returns the wheel graph on *n+1* vertices.

52.2.2 Graph properties

adjacency_matrix (*gr*) Function

Returns the adjacency matrix of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) c5 : cycle_graph(4)$
(%i3) adjacency_matrix(c5);
                          [ 0  1  0  1 ]
                          [             ]
                          [ 1  0  1  0 ]
(%o3)                     [             ]
                          [ 0  1  0  1 ]
                          [             ]
                          [ 1  0  1  0 ]
```

average_degree (*gr*) Function

Returns the average degree of vertices in the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) average_degree(grotzch_graph());
                               40
(%o2)                          --
                               11
```

biconected_components (*gr*) Function

Returns the (vertex sets of) 2-connected components of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g : create_graph(
            [1,2,3,4,5,6,7],
            [
             [1,2],[2,3],[2,4],[3,4],
             [4,5],[5,6],[4,6],[6,7]
            ])$
(%i3) biconnected_components(g);
(%o3)        [[6, 7], [4, 5, 6], [1, 2], [2, 3, 4]]
```

bipartition (*gr*) Function

Returns a bipartition of the vertices of the graph *gr* or an empty list if *gr* is not bipartite.

Example:

```
(%i1) load (graphs)$
(%i2) h : heawood_graph()$
(%i3) [A,B]:bipartition(h);
(%o3)  [[8, 12, 6, 10, 0, 2, 4], [13, 5, 11, 7, 9, 1, 3]]
(%i4) draw_graph(h, show_vertices=A, program=circular)$
```

chromatic_index (*gr*) Function

Returns the chromatic index of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) p : petersen_graph()$
(%i3) chromatic_index(p);
(%o3)                          4
```

chromatic_number (*gr*) Function

Returns the chromatic number of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) chromatic_number(cycle_graph(5));
(%o2)                          3
(%i3) chromatic_number(cycle_graph(6));
(%o3)                          2
```

clear_edge_weight (*e, gr*) Function

Removes the weight of the edge *e* in the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g : create_graph(3, [[[0,1], 1.5], [[1,2], 1.3]])$
(%i3) get_edge_weight([0,1], g);
(%o3)                          1.5
(%i4) clear_edge_weight([0,1], g)$
(%i5) get_edge_weight([0,1], g);
(%o5)                           1
```

clear_vertex_label (*v, gr*) Function

Removes the label of the vertex *v* in the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g : create_graph([[0,"Zero"], [1, "One"]], [[0,1]])$
(%i3) get_vertex_label(0, g);
(%o3)                         Zero
(%i4) clear_vertex_label(0, g);
(%o4)                         done
(%i5) get_vertex_label(0, g);
(%o5)                         false
```

connected_components (*gr*) Function

Returns the (vertex sets of) connected components of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g: graph_union(cycle_graph(5), path_graph(4))$
(%i3) connected_components(g);
(%o3)            [[1, 2, 3, 4, 0], [8, 7, 6, 5]]
```

diameter (*gr*) Function

Returns the diameter of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) diameter(dodecahedron_graph());
(%o2)                           5
```

edge_coloring (*gr*) Function

Returns an optimal coloring of the edges of the graph *gr*.

The function returns the chromatic index and a list representing the coloring of the edges of *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) p : petersen_graph()$
(%i3) [ch_index, col] : edge_coloring(p);
(%o3) [4, [[[0, 5], 3], [[5, 7], 1], [[0, 1], 1], [[1, 6], 2],
[[6, 8], 1], [[1, 2], 3], [[2, 7], 4], [[7, 9], 2], [[2, 3], 2],
[[3, 8], 3], [[5, 8], 2], [[3, 4], 1], [[4, 9], 4], [[6, 9], 3],
[[0, 4], 2]]]
(%i4) assoc([0,1], col);
(%o4)                          1
(%i5) assoc([0,5], col);
(%o5)                          3
```

degree_sequence (*gr*) Function

Returns the list of vertex degrees of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) degree_sequence(random_graph(10, 0.4));
(%o2)            [2, 2, 3, 3, 3, 4, 4, 4, 4, 5]
```

edges (*gr*) Function

Returns the list of edges (arcs) in a (directed) graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) edges(complete_graph(4));
(%o2)   [[2, 3], [1, 3], [1, 2], [0, 3], [0, 2], [0, 1]]
```

get_edge_weight (*e*, *gr*) Function
get_edge_weight (*e*, *gr*, *ifnot*) Function

Returns the weight of the edge *e* in the graph *gr*.

If there is no weight assigned to the edge, the function returns 1. If the edge is not present in the graph, the function signals an error or returns the optional argument *ifnot*.

Example:

```
(%i1) load (graphs)$
(%i2) c5 : cycle_graph(5)$
(%i3) get_edge_weight([1,2], c5);
(%o3)                            1
(%i4) set_edge_weight([1,2], 2.0, c5);
(%o4)                         done
(%i5) get_edge_weight([1,2], c5);
(%o5)                          2.0
```

get_vertex_label (*v*, *gr*) Function

Returns the label of the vertex *v* in the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g : create_graph([[0,"Zero"], [1, "One"]], [[0,1]])$
(%i3) get_vertex_label(0, g);
(%o3)                          Zero
```

graph_charpoly (*gr*, *x*) Function

Returns the characteristic polynomial (in variable *x*) of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) p : petersen_graph()$
(%i3) graph_charpoly(p, x), factor;
                                      5        4
(%o3)               (x - 3) (x - 1)  (x + 2)
```

graph_center (*gr*) Function

Returns the center of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g : grid_graph(5,5)$
(%i3) graph_center(g);
(%o3)                          [12]
```

graph_eigenvalues (*gr*) Function

Returns the eigenvalues of the graph *gr*. The function returns eigenvalues in the same format as maxima `eigenvalue` function.

Example:

```
(%i1) load (graphs)$
(%i2) p : petersen_graph()$
(%i3) graph_eigenvalues(p);
(%o3)               [[3, - 2, 1], [1, 4, 5]]
```

graph_periphery (*gr*) Function

Returns the periphery of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g : grid_graph(5,5)$
(%i3) graph_periphery(g);
(%o3)                     [0, 4, 20, 24]
```

graph_size (*gr*) Function

Returns the number of vertices in the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) p : petersen_graph()$
(%i3) graph_size(p);
(%o3)                           10
```

graph_order (*gr*) Function

Returns the number of edges in the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) p : petersen_graph()$
(%i3) graph_order(p);
(%o3)                          15
```

girth (*gr*) Function

Returns the length of the shortest cycle in *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g : heawood_graph()$
(%i3) girth(g);
(%o3)                           6
```

hamilton_cycle (*gr*) Function

Returns the Hamilton cycle of the graph *gr* or an empty list if *gr* is not hamiltonian.

Example:

```
(%i1) load (graphs)$
(%i2) c : cube_graph(3)$
(%i3) hc : hamilton_cycle(c);
(%o3)              [7, 3, 2, 6, 4, 0, 1, 5, 7]
(%i4) draw_graph(c, show_edges=vertices_to_cycle(hc))$
```

hamilton_path (*gr*) Function

Returns the Hamilton path of the graph *gr* or an empty list if *gr* does not have a Hamilton path.

Example:

```
(%i1) load (graphs)$
(%i2) p : petersen_graph()$
(%i3) hp : hamilton_path(p);
(%o3)            [0, 5, 7, 2, 1, 6, 8, 3, 4, 9]
(%i4) draw_graph(p, show_edges=vertices_to_path(hp))$
```

isomorphism (*gr1*, *gr2*) Function

Returns a hash table representing an isomorphism between graphs/digraphs *gr1* and *gr2*. If *gr1* and *gr2* are not isomorphic, it returns `false`.

Example:

```
(%i1) load (graphs)$
(%i2) clk5:complement_graph(line_graph(complete_graph(5)))$
(%i3) hash_table_data(isomorphism(clk5, petersen_graph()));
(%o3) [8 -> 9, 7 -> 8, 4 -> 7, 3 -> 6, 1 -> 5, 0 -> 4, 5 -> 3,
                                          6 -> 2, 2 -> 1, 9 -> 0]
```

in_neighbors (*v*, *gr*) Function

Returns the list of in-neighbors of the vertex *v* in the directed graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) p : path_digraph(3)$
(%i3) in_neighbors(2, p);
(%o3)                          [1]
(%i4) out_neighbors(2, p);
(%o4)                          []
```

is_biconnected (*gr*) Function

Returns `true` if *gr* is 2-connected and `false` otherwise.

Example:

```
(%i1) load (graphs)$
(%i2) is_biconnected(cycle_graph(5));
(%o2)                          true
(%i3) is_biconnected(path_graph(5));
(%o3)                          false
```

is_bipartite (*gr*) Function

Returns `true` if *gr* is bipartite (2-colorable) and `false` otherwise.

Example:

```
(%i1) load (graphs)$
(%i2) is_bipartite(petersen_graph());
(%o2)                          false
(%i3) is_bipartite(heawood_graph());
(%o3)                          true
```

is_connected (*gr*) Function

Returns `true` if the graph *gr* is connected and `false` otherwise.

Example:

```
(%i1) load (graphs)$
(%i2) is_connected(graph_union(cycle_graph(4), path_graph(3)));
(%o2)                          false
```

is_digraph (*gr*) Function

Returns `true` if *gr* is a directed graph and `false` otherwise.

Example:

```
(%i1) load (graphs)$
(%i2) is_digraph(path_graph(5));
(%o2)                          false
(%i3) is_digraph(path_digraph(5));
(%o3)                          true
```

is_edge_in_graph (*e, gr*) Function

Returns `true` if *e* is an edge (arc) in the (directed) graph *g* and `false` otherwise.

Example:

```
(%i1) load (graphs)$
(%i2) c4 : cycle_graph(4)$
(%i3) is_edge_in_graph([2,3], c4);
(%o3)                          true
(%i4) is_edge_in_graph([3,2], c4);
(%o4)                          true
(%i5) is_edge_in_graph([2,4], c4);
(%o5)                          false
(%i6) is_edge_in_graph([3,2], cycle_digraph(4));
(%o6)                          false
```

is_graph (*gr*) Function

Returns `true` if *gr* is a graph and `false` otherwise.

Example:

```
(%i1) load (graphs)$
(%i2) is_graph(path_graph(5));
(%o2)                          true
(%i3) is_graph(path_digraph(5));
(%o3)                          false
```

is_graph_or_digraph (*gr*) Function

Returns `true` if *gr* is a graph or a directed graph and `false` otherwise.

Example:

```
(%i1) load (graphs)$
(%i2) is_graph_or_digraph(path_graph(5));
(%o2)                          true
(%i3) is_graph_or_digraph(path_digraph(5));
(%o3)                          true
```

is_isomorphic (*gr1, gr2*) Function

Returns `true` if graphs/digraphs *gr1* and *gr2* are isomorphic and `false` otherwise.

See also `isomorphism`.

Example:

```
(%i1) load (graphs)$
(%i2) clk5:complement_graph(line_graph(complete_graph(5)))$
(%i3) is_isomorphic(clk5, petersen_graph());
(%o3)                          true
```

is_planar (*gr*) Function

Returns `true` if *gr* is a planar graph and `false` otherwise.

The algorithm used is the Demoucron's algorithm, which is a quadratic time algorithm.

Example:

```
(%i1) load (graphs)$
(%i2) is_planar(dodecahedron_graph());
(%o2)                          true
(%i3) is_planar(petersen_graph());
(%o3)                          false
(%i4) is_planar(petersen_graph(10,2));
(%o4)                          true
```

is_sconnected (*gr*) Function

Returns `true` if the directed graph *gr* is strongly connected and `false` otherwise.

Example:

```
(%i1) load (graphs)$
(%i2) is_sconnected(cycle_digraph(5));
(%o2)                          true
(%i3) is_sconnected(path_digraph(5));
(%o3)                          false
```

is_vertex_in_graph (*v*, *gr*) Function

Returns `true` if *v* is a vertex in the graph *g* and `false` otherwise.

Example:

```
(%i1) load (graphs)$
(%i2) c4 : cycle_graph(4)$
(%i3) is_vertex_in_graph(0, c4);
(%o3)                          true
(%i4) is_vertex_in_graph(6, c4);
(%o4)                          false
```

is_tree (*gr*) Function

Returns `true` if *gr* is a tree and `false` otherwise.

Example:

```
(%i1) load (graphs)$
(%i2) is_tree(random_tree(4));
(%o2)                          true
(%i3) is_tree(graph_union(random_tree(4), random_tree(5)));
(%o3)                          false
```

laplacian_matrix (*gr*) Function

Returns the laplacian matrix of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) laplacian_matrix(cycle_graph(5));
                    [  2   - 1   0    0   - 1 ]
                    [                        ]
                    [ - 1   2   - 1   0    0  ]
                    [                        ]
(%o2)               [  0   - 1   2   - 1   0  ]
```

```
[                          ]
[  0    0   - 1   2   - 1 ]
[                          ]
[ - 1   0    0   - 1   2  ]
```

max_clique (*gr*) Function

Returns a maximum clique of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g : random_graph(100, 0.5)$
(%i3) max_clique(g);
(%o3)       [20, 28, 42, 44, 47, 65, 69, 75, 87, 98]
```

max_degree (*gr*) Function

Returns the maximal degree of vertices of the graph *gr* and a vertex of maximal degree.

Example:

```
(%i1) load (graphs)$
(%i2) g : random_graph(100, 0.02)$
(%i3) max_degree(g);
(%o3)                        [6, 70]
(%i4) vertex_degree(95, g);
(%o4)                           3
```

max_flow (*net*, *s*, *t*) Function

Returns a maximum flow through the network *net* with the source *s* and the sink *t*.

The function returns the value of the maximal flow and a list representing the weights of the arcs in the optimal flow.

Example:

```
(%i1) load (graphs)$
(%i2) net : create_graph(
  [1,2,3,4,5,6],
  [[[1,2], 1.0],
   [[1,3], 0.3],
   [[2,4], 0.2],
   [[2,5], 0.3],
   [[3,4], 0.1],
   [[3,5], 0.1],
   [[4,6], 1.0],
   [[5,6], 1.0]],
  directed=true)$
(%i3) [flow_value, flow] : max_flow(net, 1, 6);
(%o3) [0.7, [[[1, 2], 0.5], [[1, 3], 0.2], [[2, 4], 0.2],
[[2, 5], 0.3], [[3, 4], 0.1], [[3, 5], 0.1], [[4, 6], 0.3],
[[5, 6], 0.4]]]
(%i4) fl : 0$
(%i5) for u in out_neighbors(1, net)
```

```
     do fl : fl + assoc([1, u], flow)$
(%i6) fl;
(%o6)                          0.7
```

max_independent_set (*gr*) Function

Returns a maximum independent set of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) d : dodecahedron_graph()$
(%i3) mi : max_independent_set(d);
(%o3)              [0, 3, 5, 9, 10, 11, 18, 19]
(%i4) draw_graph(d, show_vertices=mi)$
```

max_matching (*gr*) Function

Returns a maximum matching of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) d : dodecahedron_graph()$
(%i3) m : max_matching(d);
(%o3) [[1, 2], [3, 4], [0, 15], [11, 16], [12, 17], [13, 18],
                                  [14, 19], [6, 10], [8, 9], [5, 7]]
(%i4) draw_graph(d, show_edges=m)$
```

min_degree (*gr*) Function

Returns the minimum degree of vertices of the graph *gr* and a vertex of minimum degree.

Example:

```
(%i1) load (graphs)$
(%i2) g : random_graph(100, 0.1)$
(%i3) min_degree(g);
(%o3)                          [4, 83]
(%i4) vertex_degree(21, g);
(%o4)                            12
```

min_vertex_cover (*gr*) Function

Returns the minimum vertex cover of the graph *gr*.

minimum_spanning_tree (*gr*) Function

Returns the minimum spanning tree of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g : graph_product(path_graph(10), path_graph(10))$
(%i3) t : minimum_spanning_tree(g)$
(%i4) draw_graph(g, show_edges=edges(t))$
```

neighbors (*v*, *gr*) Function

Returns the list of neighbors of the vertex *v* in the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) p : petersen_graph()$
(%i3) neighbors(3, p);
(%o3)                        [4, 8, 2]
```

odd_girth (*gr*) Function

Returns the length of the shortest odd cycle in the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g : graph_product(cycle_graph(4), cycle_graph(7))$
(%i3) girth(g);
(%o3)                             4
(%i4) odd_girth(g);
(%o4)                             7
```

out_neighbors (*v*, *gr*) Function

Returns the list of out-neighbors of the vertex *v* in the directed graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) p : path_digraph(3)$
(%i3) in_neighbors(2, p);
(%o3)                            [1]
(%i4) out_neighbors(2, p);
(%o4)                            []
```

planar_embedding (*gr*) Function

Returns the list of facial walks in a planar embedding of *gr* and `false` if *gr* is not a planar graph.

The graph *gr* must be biconnected.

The algorithm used is the Demoucron's algorithm, which is a quadratic time algorithm.

Example:

```
(%i1) load (graphs)$
(%i2) planar_embedding(grid_graph(3,3));
(%o2) [[3, 6, 7, 8, 5, 2, 1, 0], [4, 3, 0, 1], [3, 4, 7, 6],
                                        [8, 7, 4, 5], [1, 2, 5, 4]]
```

print_graph (*gr*) Function

Prints some information about the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) c5 : cycle_graph(5)$
(%i3) print_graph(c5)$
Graph on 5 vertices with 5 edges.
Adjacencies:
  4 :  0  3
  3 :  4  2
  2 :  3  1
  1 :  2  0
  0 :  4  1
(%i4) dc5 : cycle_digraph(5)$
(%i5) print_graph(dc5)$
Digraph on 5 vertices with 5 arcs.
Adjacencies:
  4 :  0
  3 :  4
  2 :  3
  1 :  2
  0 :  1
(%i6) out_neighbors(0, dc5);
(%o6)                          [1]
```

radius (*gr*) Function

Returns the radius of the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) radius(dodecahedron_graph());
(%o2)                           5
```

set_edge_weight (*e*, *w*, *gr*) Function

Assigns the weight *w* to the edge *e* in the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g : create_graph([1, 2], [[[1,2], 1.2]])$
(%i3) get_edge_weight([1,2], g);
(%o3)                           1.2
(%i4) set_edge_weight([1,2], 2.1, g);
(%o4)                          done
(%i5) get_edge_weight([1,2], g);
(%o5)                           2.1
```

set_vertex_label (*v*, *l*, *gr*) Function

Assigns the label *l* to the vertex *v* in the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g : create_graph([[1, "One"], [2, "Two"]], [[1,2]])$
(%i3) get_vertex_label(1, g);
```

```
(%o3)                          One
(%i4) set_vertex_label(1, "oNE", g);
(%o4)                         done
(%i5) get_vertex_label(1, g);
(%o5)                          oNE
```

shortest_path (*u, v, gr*) Function

Returns the shortest path from *u* to *v* in the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) d : dodecahedron_graph()$
(%i3) path : shortest_path(0, 7, d);
(%o3)                  [0, 1, 19, 13, 7]
(%i4) draw_graph(d, show_edges=vertices_to_path(path))$
```

strong_components (*gr*) Function

Returns the strong components of a directed graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) t : random_tournament(4)$
(%i3) strong_components(t);
(%o3)                [[2], [0], [1], [3]]
(%i4) vertex_out_degree(3, t);
(%o4)                          2
```

topological_sort (*dag*) Function

Returns a topological sorting of the vertices of a directed graph *dag* or an empty list if *dag* is not a directed acyclic graph.

Example:

```
(%i1) load (graphs)$
(%i2) g:create_graph(
         [1,2,3,4,5],
         [
          [1,2], [2,5], [5,3],
          [5,4], [3,4], [1,3]
         ],
         directed=true)$
(%i3) topological_sort(g);
(%o3)                   [1, 2, 5, 3, 4]
```

vertex_degree (*v, gr*) Function

Returns the degree of the vertex *v* in the graph *gr*.

vertex_distance (*u, v, gr*) Function

Returns the length of the shortest path between *u* and *v* in the (directed) graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) d : dodecahedron_graph()$
(%i3) vertex_distance(0, 7, d);
(%o3)                          4
(%i4) shortest_path(0, 7, d);
(%o4)                  [0, 1, 19, 13, 7]
```

vertex_eccentricity (*v*, *gr*) Function

Returns the eccentricity of the vertex *v* in the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g:cycle_graph(7)$
(%i3) vertex_eccentricity(0, g);
(%o3)                          3
```

vertex_in_degree (*v*, *gr*) Function

Returns the in-degree of the vertex *v* in the directed graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) p5 : path_digraph(5)$
(%i3) print_graph(p5)$
Digraph on 5 vertices with 4 arcs.
Adjacencies:
  4 :
  3 :  4
  2 :  3
  1 :  2
  0 :  1
(%i4) vertex_in_degree(4, p5);
(%o4)                          1
(%i5) in_neighbors(4, p5);
(%o5)                         [3]
```

vertex_out_degree (*v*, *gr*) Function

Returns the out-degree of the vertex *v* in the directed graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) t : random_tournament(10)$
(%i3) vertex_out_degree(0, t);
(%o3)                          6
(%i4) out_neighbors(0, t);
(%o4)                 [9, 6, 4, 3, 2, 1]
```

vertices (*gr*) Function

Returns the list of vertices in the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) vertices(complete_graph(4));
(%o2)                     [3, 2, 1, 0]
```

52.2.3 Modifying graphs

add_edge (*e, gr*) Function

Adds the edge *e* to the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) p : path_graph(4)$
(%i3) neighbors(0, p);
(%o3)                          [1]
(%i4) add_edge([0,3], p);
(%o4)                         done
(%i5) neighbors(0, p);
(%o5)                        [3, 1]
```

add_edges (*e_list, gr*) Function

Adds all edges in the list *e_list* to the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g : empty_graph(3)$
(%i3) add_edges([[0,1],[1,2]], g)$
(%i4) print_graph(g)$
Graph on 3 vertices with 2 edges.
Adjacencies:
  2 :  1
  1 :  2  0
  0 :  1
```

add_vertex (*v, gr*) Function

Adds the vertex *v* to the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g : path_graph(2)$
(%i3) add_vertex(2, g)$
(%i4) print_graph(g)$
Graph on 3 vertices with 1 edges.
Adjacencies:
  2 :
  1 :  0
  0 :  1
```

add_vertices (*v_list, gr*) Function

Adds all vertices in the list *v_list* to the graph *gr*.

connect_vertices (*v_list*, *u_list*, *gr*) Function

Connects all vertices from the list *v_list* with the vertices in the list *u_list* in the graph *gr*.

v_list and *u_list* can be single vertices or lists of vertices.

Example:

```
(%i1) load (graphs)$
(%i2) g : empty_graph(4)$
(%i3) connect_vertices(0, [1,2,3], g)$
(%i4) print_graph(g)$
Graph on 4 vertices with 3 edges.
Adjacencies:
  3 :  0
  2 :  0
  1 :  0
  0 :  3  2  1
```

contract_edge (*e*, *gr*) Function

Contracts the edge *e* in the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) g: create_graph(
      8, [[0,3],[1,3],[2,3],[3,4],[4,5],[4,6],[4,7]])$
(%i3) print_graph(g)$
Graph on 8 vertices with 7 edges.
Adjacencies:
  7 :  4
  6 :  4
  5 :  4
  4 :  7  6  5  3
  3 :  4  2  1  0
  2 :  3
  1 :  3
  0 :  3
(%i4) contract_edge([3,4], g)$
(%i5) print_graph(g)$
Graph on 7 vertices with 6 edges.
Adjacencies:
  7 :  3
  6 :  3
  5 :  3
  3 :  5  6  7  2  1  0
  2 :  3
  1 :  3
  0 :  3
```

remove_edge (*e*, *gr*) Function

Removes the edge *e* from the graph *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) c3 : cycle_graph(3)$
(%i3) remove_edge([0,1], c3)$
(%i4) print_graph(c3)$
Graph on 3 vertices with 2 edges.
Adjacencies:
  2 :  0  1
  1 :  2
  0 :  2
```

remove_vertex (*v*, *gr*) Function

Removes the vertex *v* from the graph *gr*.

vertex_coloring (*gr*) Function

Returns an optimal coloring of the vertices of the graph *gr*.

The function returns the chromatic number and a list representing the coloring of the vertices of *gr*.

Example:

```
(%i1) load (graphs)$
(%i2) p:petersen_graph()$
(%i3) vertex_coloring(p);
(%o3) [3, [[0, 2], [1, 3], [2, 2], [3, 3], [4, 1], [5, 3],
                                    [6, 1], [7, 1], [8, 2], [9, 2]]]
```

52.2.4 Reading and writing to files

dimacs_export (*gr*, *fl*) Function
dimacs_export (*gr*, *fl*, *comment1*, ..., *commentn*) Function

Exports the graph into the file *fl* in the DIMACS format. Optional comments will be added to the top of the file.

dimacs_import (*fl*) Function

Returns the graph from file *fl* in the DIMACS format.

graph6_decode (*str*) Function

Returns the graph encoded in the graph6 format in the string *str*.

graph6_encode (*gr*) Function

Returns a string which encodes the graph *gr* in the graph6 format.

graph6_export (*gr_list*, *fl*) Function

Exports graphs in the list *gr_list* to the file *fl* in the graph6 format.

graph6_import (*fl*) Function

Returns a list of graphs from the file *fl* in the graph6 format.

sparse6_decode (*str*) Function
Returns the graph encoded in the sparse6 format in the string *str*.

sparse6_encode (*gr*) Function
Returns a string which encodes the graph *gr* in the sparse6 format.

sparse6_export (*gr_list*, *fl*) Function
Exports graphs in the list *gr_list* to the file *fl* in the sparse6 format.

sparse6_import (*fl*) Function
Returns a list of graphs from the file *fl* in the sparse6 format.

52.2.5 Visualization

draw_graph (*graph*) Function
draw_graph (*graph*, *option1*, ..., *optionk*) Function
Draws the graph using the `draw` package.

The algorithm used to position vertices is specified by the optional argument *program*. The default value is `program=spring_embedding`. *spring_embedding* can also use the graphviz programs for positioning vertices, but graphviz must be installed separately.

Optional arguments to the *draw_graph* function can be:

- *show_id=show*: if *show* is *true* then ids of the vertices are displayed.
- *show_label=show*: if *show* is *true* then labels of the vertices are displayed.
- *label_alignment=pos*: how to align the label/id of the vertices. Can be `left`, `center` or `right`. The default is `left`.
- *show_weight=show*: if *show* is *true* then weights of the edges are displayed.
- *vertex_type=type*: defines how vertices are displayed. See the *point_type* option for the `draw` package.
- *vertex_size=size*: the size of vertices.
- *vertex_color=c*: color used for displaying vertices.
- *show_vertices=v_list*: display vertices in the list *v_list* using a different color.
- *show_vertex_type=type*: defines how vertices in *show_vertices* are displayed. See the *point_type* option for the `draw` package.
- *show_vertex_size=size*: the size of vertices in *show_vertices*.
- *show_vertex_color=c*: color used for displaying vertices in the *show_vertices* list.
- *vertex_partition=part*: 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=col*: coloring of the vertices. The coloring *col* must be specified in the format as returned by *vertex_coloring*.
- *edge_color=c*: color used for displaying edges.
- *edge_width=width*: the width of edges.

- *edge_type=type*: defines how edges are displayed. See the *line_type* option for the `draw` package.
- *show_edges=e_list*: display edges in the list *e_list* using a different color.
- *show_edge_color=c*: color used for displaying edges in the *show_edges* list.
- *show_edge_width=width*: the width of edges in *show_edges*.
- *show_edge_type=type*: defines how edges in *show_edges* are displayed. See the *line_type* option for the `draw` package.
- *edge_partition=partition*: a partition `[[e1,e2,...],...,[ek,...,em]]` of edges of the graph. The edges of each list in the partition will be drawn using a different color.
- *edge_coloring=col*: the coloring of edges. The coloring *col* must be specified in the format as returned by the function *edge_coloring*.
- *redraw=r*: if *redraw* is `true`, vertex positions are recomputed even if the positions have been saved from a previous drawing of the graph.
- *head_angle=angle*: the angle for the arrows displayed on arcs (in directed graphs). Default value: 15.
- *head_length=len*: the length for the arrows displayed on arcs (in directed graphs). Default value: 0.1.
- *spring_embedding_depth=depth*: the number of iterations in the spring embedding graph drawing algorithm. Default value: 50.
- *terminal=term*: the terminal used for drawing (see the *terminal* option in the `draw` package).
- *file_name=file*: the filename of the drawing if terminal is not screen.
- *program=prg*: 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`.

Example 1:

```
(%i1) load (graphs)$
(%i2) g:grid_graph(10,10)$
(%i3) m:max_matching(g)$
(%i4) draw_graph(g,
   spring_embedding_depth=100,
   show_edges=m, edge_type=dots,
   vertex_size=0)$
```

Example 2:

```
(%i1) load (graphs)$
(%i2) g:create_graph(16,
    [
     [0,1],[1,3],[2,3],[0,2],[3,4],[2,4],
```

```
     [5,6],[6,4],[4,7],[6,7],[7,8],[7,10],[7,11],
     [8,10],[11,10],[8,9],[11,12],[9,15],[12,13],
     [10,14],[15,14],[13,14]
    ])$
(%i3) t:minimum_spanning_tree(g)$
(%i4) draw_graph(
    g,
    show_edges=edges(t),
    show_edge_width=4,
    show_edge_color=green,
    vertex_type=filled_square,
    vertex_size=2
    )$
```

Example 3:

```
(%i1) load (graphs)$
(%i2) g:create_graph(16,
    [
     [0,1],[1,3],[2,3],[0,2],[3,4],[2,4],
     [5,6],[6,4],[4,7],[6,7],[7,8],[7,10],[7,11],
     [8,10],[11,10],[8,9],[11,12],[9,15],[12,13],
     [10,14],[15,14],[13,14]
    ])$
(%i3) mi : max_independent_set(g)$
(%i4) draw_graph(
    g,
    show_vertices=mi,
    show_vertex_type=filled_up_triangle,
    show_vertex_size=2,
    edge_color=cyan,
    edge_width=3,
    show_id=true,
    text_color=brown
    )$
```

Example 4:

```
(%i1) load (graphs)$
(%i2) net : create_graph(
    [0,1,2,3,4,5],
    [
     [[0,1], 3], [[0,2], 2],
     [[1,3], 1], [[1,4], 3],
     [[2,3], 2], [[2,4], 2],
     [[4,5], 2], [[3,5], 2]
    ],
    directed=true
    )$
(%i3) draw_graph(
    net,
    show_weight=true,
```

```
vertex_size=0,
show_vertices=[0,5],
show_vertex_type=filled_square,
head_length=0.2,
head_angle=10,
edge_color="dark-green",
text_color=blue
)$
```

draw_graph_program Option variable

Default value: *spring_embedding*.

The default value for the program used to position vertices in `draw_graph` program.

vertices_to_path (*v_list*) Function

Converts a list *v_list* of vertices to a list of edges of the path defined by *v_list*.

vertices_to_cycle (*v_list*) Function

Converts a list *v_list* of vertices to a list of edges of the cycle defined by *v_list*.

53 grobner

53.1 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.

53.1.1 Notes on the grobner package

The package was written by

Marek Rychlik

`http://alamos.math.arizona.edu`

and is released 2002-05-24 under the terms of the General Public License(GPL) (see file '`grobner.lisp`'. This documentation was extracted from the files

'`README`', '`grobner.lisp`', '`grobner.demo`', '`grobner-demo.output`'

by Günter Nowak. Suggestions for improvement of the documentation can be discussed at the *maxima*-mailing-list `maxima@math.utexas.edu`. The code is a little bit out of date now. Modern implementation use the fast *F4* algorithm described in

A new efficient algorithm for computing Gröbner bases (F4)
Jean-Charles Faugère
LIP6/CNRS Université Paris VI
January 20, 1999

53.1.2 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

53.2 Functions and Variables for grobner

53.2.1 Global switches for grobner

poly_monomial_order Option variable

Default value: `lex`

This global switch controls which monomial order is used in polynomial and Groebner Bases calculations. If not set, `lex` will be used.

poly_coefficient_ring Option variable

Default value: `expression_ring`

This switch indicates the coefficient ring of the polynomials that will be used in grobner calculations. If not set, *maxima's* general expression ring will be used. This variable may be set to `ring_of_integers` if desired.

poly_primary_elimination_order Option variable

Default value: `false`

Name of the default order for eliminated variables in elimination-based functions. If not set, `lex` will be used.

poly_secondary_elimination_order Option variable

Default value: `false`

Name of the default order for kept variables in elimination-based functions. If not set, `lex` will be used.

poly_elimination_order Option variable

Default value: `false`

Name of the default elimination order used in elimination calculations. If set, it overrides the settings in variables `poly_primary_elimination_order` and `poly_secondary_elimination_order`. The user must ensure that this is a true elimination order valid for the number of eliminated variables.

poly_return_term_list Option variable

Default value: `false`

If set to `true`, all functions in this package will return each polynomial as a list of terms in the current monomial order rather than a *maxima* general expression.

poly_grobner_debug Option variable

Default value: `false`

If set to `true`, produce debugging and tracing output.

poly_grobner_algorithm Option variable

Default value: `buchberger`

Possible values:

`buchberger`

`parallel_buchberger`

`gebauer_moeller`

The name of the algorithm used to find the Groebner Bases.

poly_top_reduction_only Option variable

Default value: `false`

If not `false`, use top reduction only whenever possible. Top reduction means that division algorithm stops after the first reduction.

53.2.2 Simple operators in grobner

`poly_add`, `poly_subtract`, `poly_multiply` and `poly_expt` are the arithmetical operations on polynomials. These are performed using the internal representation, but the results are converted back to the *maxima* general form.

poly_add (*poly1, poly2, varlist*) Function

Adds two polynomials *poly1* and *poly2*.

```
(%i1) poly_add(z+x^2*y,x-z,[x,y,z]);
                                    2
(%o1)                              x  y + x
```

poly_subtract (*poly1, poly2, varlist*) Function

Subtracts a polynomial *poly2* from *poly1*.

```
(%i1) poly_subtract(z+x^2*y,x-z,[x,y,z]);
                                      2
(%o1)                          2 z + x  y - x
```

poly_multiply (*poly1, poly2, varlist*) Function

Returns the product of polynomials *poly1* and *poly2*.

```
(%i2) poly_multiply(z+x^2*y,x-z,[x,y,z])-(z+x^2*y)*(x-z),expand;
(%o1)                                  0
```

poly_s_polynomial (*poly1, poly2, varlist*) Function

Returns the *syzygy polynomial* (*S-polynomial*) of two polynomials *poly1* and *poly2*.

poly_primitive_part (*poly1, varlist*) Function

Returns the polynomial *poly* divided by the GCD of its coefficients.

```
(%i1) poly_primitive_part(35*y+21*x,[x,y]);
(%o1)                              5 y + 3 x
```

poly_normalize (*poly, varlist*) Function

Returns the polynomial *poly* divided by the leading coefficient. It assumes that the division is possible, which may not always be the case in rings which are not fields.

53.2.3 Other functions in grobner

poly_expand (*poly, varlist*) Function

This function parses polynomials to internal form and back. It is equivalent to `expand(`*poly*`)` if *poly* parses correctly to a polynomial. If the representation is not compatible with a polynomial in variables *varlist*, the result is an error. It can be used to test whether an expression correctly parses to the internal representation. The following examples illustrate that indexed and transcendental function variables are allowed.

```
(%i1) poly_expand((x-y)*(y+x),[x,y]);
                                     2    2
(%o1)                               x  - y
(%i2) poly_expand((y+x)^2,[x,y]);
                                 2           2
(%o2)                           y  + 2 x y + x
(%i3) poly_expand((y+x)^5,[x,y]);
                 5      4         2  3       3  2      4      5
(%o3)           y  + 5 x y  + 10 x  y  + 10 x  y  + 5 x  y + x
(%i4) poly_expand(-1-x*exp(y)+x^2/sqrt(y),[x]);
                                       2
                                    y      x
(%o4)                     - x %e  + ------- - 1
                                     sqrt(y)

(%i5) poly_expand(-1-sin(x)^2+sin(x),[sin(x)]);
                                 2
(%o5)                     - sin (x) + sin(x) - 1
```

poly_expt (*poly, number, varlist*) Function

exponentitates *poly* by a positive integer *number*. If *number* is not a positive integer number an error will be raised.

```
(%i1) poly_expt(x-y,3,[x,y])-(x-y)^3,expand;
(%o1)                                0
```

poly_content (*poly. varlist*) Function

`poly_content` extracts the GCD of its coefficients

```
(%i1) poly_content(35*y+21*x,[x,y]);
(%o1)                                7
```

poly_pseudo_divide (*poly*, *polylist*, *varlist*) Function

Pseudo-divide a polynomial *poly* by the list of n polynomials *polylist*. Return multiple values. The first value is a list of quotients a. The second value is the remainder r. The third argument is a scalar coefficient c, such that $c * poly$ can be divided by *polylist* within the ring of coefficients, which is not necessarily a field. Finally, the fourth value is an integer count of the number of reductions performed. The resulting objects satisfy the equation:

$$c * poly = \sum_{i=1}^{n} (a_i * polylist_i) + r$$

poly_exact_divide (*poly1*, *poly2*, *varlist*) Function

Divide a polynomial *poly1* by another polynomial *poly2*. Assumes that exact division with no remainder is possible. Returns the quotient.

poly_normal_form (*poly*, *polylist*, *varlist*) Function

`poly_normal_form` finds the normal form of a polynomial *poly* with respect to a set of polynomials *polylist*.

poly_buchberger_criterion (*polylist*, *varlist*) Function

Returns `true` if *polylist* is a Groebner basis with respect to the current term order, by using the Buchberger criterion: for every two polynomials $h1$ and $h2$ in *polylist* the S-polynomial $S(h1, h2)$ reduces to 0 *modulo* *polylist*.

poly_buchberger (*polylist_fl* *varlist*) Function

`poly_buchberger` performs the Buchberger algorithm on a list of polynomials and returns the resulting Groebner basis.

53.2.4 Standard postprocessing of Groebner Bases

The *k-th elimination ideal* I_k of an ideal I over $K[x_1, ..., x_1]$ is $I \cap K[x_{k+1}, ..., x_n]$.

The *colon ideal* $I : J$ is the ideal $\{h | \forall w \in J : wh \in I\}$.

The ideal $I : p^\infty$ is the ideal $\{h | \exists n \in N : p^n h \in I\}$.

The ideal $I : J^\infty$ is the ideal $\{h | \exists n \in N, \exists p \in J : p^n h \in I\}$.

The *radical ideal* $\sqrt{I}$ is the ideal $\{h | \exists n \in N : h^n \in I\}$.

poly_reduction (*polylist*, *varlist*) Function

`poly_reduction` reduces a list of polynomials *polylist*, so that each polynomial is fully reduced with respect to the other polynomials.

poly_minimization (*polylist*, *varlist*) Function

Returns a sublist of the polynomial list *polylist* spanning the same monomial ideal as *polylist* but minimal, i.e. no leading monomial of a polynomial in the sublist divides the leading monomial of another polynomial.

poly_normalize_list (*polylist, varlist*) Function

`poly_normalize_list` applies `poly_normalize` to each polynomial in the list. That means it divides every polynomial in a list *polylist* by its leading coefficient.

poly_grobner (*polylist, varlist*) Function

Returns a Groebner basis of the ideal span by the polynomials *polylist*. Affected by the global flags.

poly_reduced_grobner (*polylist, varlist*) Function

Returns a reduced Groebner basis of the ideal span by the polynomials *polylist*. Affected by the global flags.

poly_depends_p (*poly, var, varlist*) Function

`poly_depends` tests whether a polynomial depends on a variable *var*.

poly_elimination_ideal (*polylist, number, varlist*) Function

`poly_elimination_ideal` returns the grobner basis of the *number*-th elimination ideal of an ideal specified as a list of generating polynomials (not necessarily Groebner basis).

poly_colon_ideal (*polylist1, polylist2, varlist*) Function

Returns the reduced Groebner basis of the colon ideal

$I(polylist1) : I(polylist2)$

where *polylist*1 and *polylist*2 are two lists of polynomials.

poly_ideal_intersection (*polylist1, polylist2, varlist*) Function

`poly_ideal_intersection` returns the intersection of two ideals.

poly_lcm (*poly1, poly2, varlist*) Function

Returns the lowest common multiple of *poly1* and *poly2*.

poly_gcd (*poly1, poly2, varlist*) Function

Returns the greatest common divisor of *poly1* and *poly2*.

poly_grobner_equal (*polylist1, polylist2, varlist*) Function

`poly_grobner_equal` tests whether two Groebner Bases generate the same ideal. Returns `true` if two lists of polynomials *polylist1* and *polylist2*, assumed to be Groebner Bases, generate the same ideal, and `false` otherwise. This is equivalent to checking that every polynomial of the first basis reduces to 0 modulo the second basis and vice versa. Note that in the example below the first list is not a Groebner basis, and thus the result is `false`.

```
(%i1) poly_grobner_equal([y+x,x-y],[x,y],[x,y]);
(%o1)                          false
```

poly_grobner_subsetp (*polylist1, polylist2, varlist*) Function

`poly_grobner_subsetp` tests whether an ideal generated by *polylist1* is contained in the ideal generated by *polylist2*. For this test to always succeed, *polylist2* must be a Groebner basis.

poly_grobner_member (*poly, polylist, varlist*) Function

Returns `true` if a polynomial *poly* belongs to the ideal generated by the polynomial list *polylist*, which is assumed to be a Groebner basis. Returns `false` otherwise.

`poly_grobner_member` tests whether a polynomial belongs to an ideal generated by a list of polynomials, which is assumed to be a Groebner basis. Equivalent to `normal_form` being 0.

poly_ideal_saturation1 (*polylist, poly, varlist*) Function

Returns the reduced Groebner basis of the saturation of the ideal

$$I(polylist) : poly^{\infty}$$

Geometrically, over an algebraically closed field, this is the set of polynomials in the ideal generated by *polylist* which do not identically vanish on the variety of *poly*.

poly_ideal_saturation (*polylist1, polylist2, varlist*) Function

Returns the reduced Groebner basis of the saturation of the ideal

$$I(polylist1) : I(polylist2)^{\infty}$$

Geometrically, over an algebraically closed field, this is the set of polynomials in the ideal generated by *polylist1* which do not identically vanish on the variety of *polylist2*.

poly_ideal_polysaturation1 (*polylist1, polylist2, varlist*) Function

polylist2 ist a list of n polynomials `[poly1,...,polyn]`. Returns the reduced Groebner basis of the ideal

$$I(polylist) : poly1^{\infty} : ... : polyn^{\infty}$$

obtained by a sequence of successive saturations in the polynomials of the polynomial list *polylist2* of the ideal generated by the polynomial list *polylist1*.

poly_ideal_polysaturation (*polylist, polylistlist, varlist*) Function

polylistlist is a list of n list of polynomials `[polylist1,...,polylistn]`. Returns the reduced Groebner basis of the saturation of the ideal

$$I(polylist) : I(polylist_1)^{\infty} : ... : I(polylist_n)^{\infty}$$

poly_saturation_extension (*poly, polylist, varlist1, varlist2*) Function

`poly_saturation_extension` implements the famous Rabinowitz trick.

poly_polysaturation_extension (*poly, polylist, varlist1, varlist2*) Function

54 impdiff

54.1 Functions and Variables for impdiff

implicit_derivative (*f,indvarlist,orderlist,depvar*) Function

This subroutine computes implicit derivatives of multivariable functions. *f* is an array function, the indexes are the derivative degree in the *indvarlist* order; *indvarlist* is the independent variable list; *orderlist* is the order desired; and *depvar* is the dependent variable.

To use this function write first `load("impdiff")`.

55 implicit_plot

55.1 Functions and Variables for implicit_plot

implicit_plot (*expr*, *x_range*, *y_range*) Function
implicit_plot ([*expr_1*, ..., *expr_n*], *x_range*, *y_range*) Function

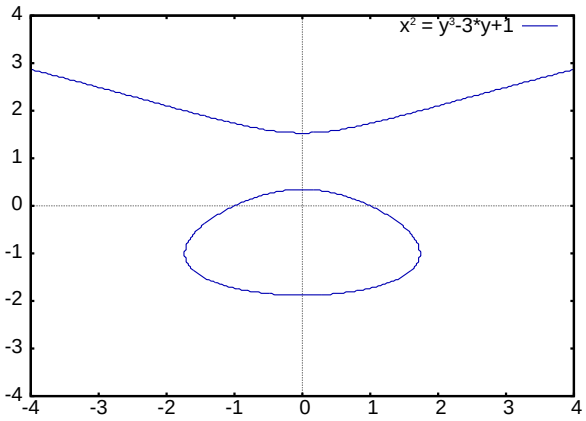
Displays a plot of one or more expressions in implicit form. *expr* is the expression to be plotted, *x_range* the range of the horizontal axis and *y_range* the range of vertical axis. `implicit_plot` respects global setting for the gnuplot driver set by the *set_plot_option* function. Options can also 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"]);
```



56 interpol

56.1 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'*.

56.2 Functions and Variables for interpol

lagrange (*points*) Function
lagrange (*points, option*) Function

Computes the polynomial interpolation by the Lagrangian method. Argument *points* must be either:

- a two column matrix, `p:matrix([2,4],[5,6],[9,3])`,
- a list of pairs, `p: [[2,4],[5,6],[9,3]]`,
- a list of numbers, `p: [4,6,3]`, in which case the abscissas will be assigned automatically to 1, 2, 3, etc.

In the first two cases the pairs are ordered with respect to the first coordinate before making computations.

With the *option* argument it is possible to select the name for the independent variable, which is `'x` by default; to define another one, write something like `varname='z`.

Note that when working with high degree polynomials, floating point evaluations are instable.

Examples:

```
(%i1) load(interpol)$
(%i2) p:[[7,2],[8,2],[1,5],[3,2],[6,7]]$
(%i3) lagrange(p);
                    4         3          2
               73 x    701 x    8957 x    5288 x   186
(%o3)          ----- - ------ + ------- - ------ + ---
                420      210        420       105      5
(%i4) f(x):=''%;
                       4         3          2
                  73 x    701 x    8957 x    5288 x   186
(%o4)    f(x) := ----- - ------ + ------- - ------ + ---
                  420      210        420       105      5
(%i5) /* Evaluate the polynomial at some points */
      map(f,[2.3,5/7,%pi]);
                              919062
(%o5)  [- 1.567534999999992, ------,
                              84035
                          4          3           2
```

```
                    73 %pi    701 %pi    8957 %pi    5288 %pi   186
                    ------- - -------- + --------- - -------- + ---]
                      420       210          420         105       5
(%i6) %,numer;
(%o6) [- 1.567534999999992, 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);
                 4        3         2
              73 w    701 w    8957 w    5288 w   186
(%o9)         ----- - ------ + ------- - ------ + ---
               420     210       420      105      5
```

charfun2 (*x*, *a*, *b*) Function

Returns `true` if number *x* belongs to the interval $[a, b)$, and `false` otherwise.

linearinterpol (*points*) Function
linearinterpol (*points*, *option*) Function

Computes the polynomial interpolation by the linear method. Argument *points* must be either:

- a two column matrix, `p:matrix([2,4],[5,6],[9,3])`,
- a list of pairs, `p: [[2,4],[5,6],[9,3]]`,
- a list of numbers, `p: [4,6,3]`, in which case the abscissas will be assigned automatically to 1, 2, 3, etc.

In the first two cases the pairs are ordered with respect to the first coordinate before making computations.

With the *option* argument it is possible to select the name for the independent variable, which is `'x` by default; to define another one, write something like `varname='z`.

Examples:

```
(%i1) load(interpol)$
(%i2) p: matrix([7,2],[8,3],[1,5],[3,2],[6,7])$
(%i3) linearinterpol(p);
        13   3 x
(%o3)  (-- - ---) charfun2(x, minf, 3)
        2     2
 + (x - 5) charfun2(x, 7, inf) + (37 - 5 x) charfun2(x, 6, 7)
    5 x
```

```
 + (--- - 3) charfun2(x, 3, 6)
     3

(%i4) f(x):=''%;
                  13   3 x
(%o4)  f(x) := (-- - ---) charfun2(x, minf, 3)
                  2     2
 + (x - 5) charfun2(x, 7, inf) + (37 - 5 x) charfun2(x, 6, 7)
    5 x
 + (--- - 3) charfun2(x, 3, 6)
     3
(%i5)  /* Evaluate the polynomial at some points */
       map(f,[7.3,25/7,%pi]);
                             62  5 %pi
(%o5)                 [2.3, --, ----- - 3]
                             21    3
(%i6) %,numer;
(%o6)  [2.3, 2.952380952380953, 2.235987755982989]
(%i7) load(draw)$  /* load draw package */
(%i8)  /* Plot the polynomial together with points */
       draw2d(
         color      = red,
         key        = "Linear interpolator",
         explicit(f(x),x,-5,20),
         point_size = 3,
         color      = blue,
         key        = "Sample points",
         points(args(p)))$
(%i9)  /* Change variable name */
       linearinterpol(p, varname='s);
       13   3 s
(%o9) (-- - ---) charfun2(s, minf, 3)
       2     2
 + (s - 5) charfun2(s, 7, inf) + (37 - 5 s) charfun2(s, 6, 7)
    5 s
 + (--- - 3) charfun2(s, 3, 6)
     3
```

cspline (*points*) Function
cspline (*points*, *option1*, *option2*, ...) Function

Computes the polynomial interpolation by the cubic splines method. Argument *points* must be either:

- a two column matrix, `p:matrix([2,4],[5,6],[9,3])`,
- a list of pairs, `p: [[2,4],[5,6],[9,3]]`,
- a list of numbers, `p: [4,6,3]`, in which case the abscissas will be assigned automatically to 1, 2, 3, etc.

In the first two cases the pairs are ordered with respect to the first coordinate before making computations.

There are three options to fit specific needs:

- `'d1`, default `'unknown`, is the first derivative at x_1; if it is `'unknown`, the second derivative at x_1 is made equal to 0 (natural cubic spline); if it is equal to a number, the second derivative is calculated based on this number.
- `'dn`, default `'unknown`, is the first derivative at x_n; if it is `'unknown`, the second derivative at x_n is made equal to 0 (natural cubic spline); if it is equal to a number, the second derivative is calculated based on this number.
- `'varname`, default `'x`, is the name of the independent variable.

Examples:

```
(%i1) load(interpol)$
(%i2) p:[[7,2],[8,2],[1,5],[3,2],[6,7]]$
(%i3) /* Unknown first derivatives at the extremes
         is equivalent to natural cubic splines */
      cspline(p);
              3         2
       1159 x    1159 x    6091 x   8283
(%o3)  (------- - ------- - ------ + ----) charfun2(x, minf, 3)
        3288      1096      3288    1096
           3         2
      2587 x    5174 x    494117 x   108928
 + (- ------- + ------- - -------- + ------) charfun2(x, 7, inf)
       1644       137       1644      137
          3          2
   4715 x    15209 x    579277 x   199575
 + (------- - -------- + -------- - ------) charfun2(x, 6, 7)
     1644       274        1644      274
            3         2
      3287 x    2223 x    48275 x   9609
 + (- ------- + ------- - ------- + ----) charfun2(x, 3, 6)
       4932       274       1644     274

(%i4) f(x):=''%$
(%i5) /* Some evaluations */
      map(f,[2.3,5/7,%pi]), numer;
(%o5) [1.991460766423356, 5.823200187269903, 2.227405312429507]
(%i6) load(draw)$  /* load draw package */
(%i7) /* Plotting interpolating function */
      draw2d(
        color      = red,
        key        = "Cubic splines",
        explicit(f(x),x,0,10),
        point_size = 3,
        color      = blue,
        key        = "Sample points",
        points(p))$
(%i8) /* New call, but giving values at the derivatives */
      cspline(p,d1=0,dn=0);
              3         2
```

```
        1949 x    11437 x    17027 x   1247
(%o8)  (------- - -------- + ------- + ----) charfun2(x, minf, 3)
         2256       2256       2256     752
            3          2
      1547 x    35581 x    68068 x   173546
 + (- ------- + -------- - ------- + ------) charfun2(x, 7, inf)
        564       564        141      141
         3          2
    607 x    35147 x    55706 x   38420
 + (------ - -------- + ------- - -----) charfun2(x, 6, 7)
     188       564        141      47
             3         2
      3895 x    1807 x    5146 x   2148
 + (- ------- + ------- - ------ + ----) charfun2(x, 3, 6)
       5076       188       141      47
(%i8) /* Defining new interpolating function */
      g(x):=''%$
(%i9) /* Plotting both functions together */
      draw2d(
        color      = black,
        key        = "Cubic splines (default)",
        explicit(f(x),x,0,10),
        color      = red,
        key        = "Cubic splines (d1=0,dn=0)",
        explicit(g(x),x,0,10),
        point_size = 3,
        color      = blue,
        key        = "Sample points",
        points(p))$
```

57 lapack

57.1 Introduction to lapack

`lapack` is a Common Lisp translation (via the program `f2c`) of the Fortran library LAPACK, as obtained from the SLATEC project.

57.2 Functions and Variables for lapack

dgeev (*A*) Function
dgeev (*A*, *right_p*, *left_p*) Function

Computes the eigenvalues and, optionally, the eigenvectors of a matrix A. All elements of A must be integer or floating point numbers. A must be square (same number of rows and columns). A might or might not be symmetric.

`dgeev`(*A*) computes only the eigenvalues of A. `dgeev`(*A*, *right_p*, *left_p*) computes the eigenvalues of A and the right eigenvectors when *right_p* = `true` and the left eigenvectors when *left_p* = `true`.

A list of three items is returned. The first item is a list of the eigenvalues. The second item is `false` or the matrix of right eigenvectors. The third item is `false` or the matrix of left eigenvectors.

The right eigenvector $v(j)$ (the j-th column of the right eigenvector matrix) satisfies

$A.v(j) = lambda(j).v(j)$

where $lambda(j)$ is the corresponding eigenvalue. The left eigenvector $u(j)$ (the j-th column of the left eigenvector matrix) satisfies

$u(j) * *H.A = lambda(j).u(j) * *H$

where $u(j) * *H$ denotes the conjugate transpose of $u(j)$. The Maxima function `ctranspose` computes the conjugate transpose.

The computed eigenvectors are normalized to have Euclidean norm equal to 1, and largest component has imaginary part equal to zero.

Example:

```
(%i1) load (lapack)$
(%i2) fpprintprec : 6;
(%o2)                           6
(%i3) M : matrix ([9.5, 1.75], [3.25, 10.45]);
                          [ 9.5   1.75  ]
(%o3)                     [             ]
                          [ 3.25  10.45 ]
(%i4) dgeev (M);
(%o4)          [[7.54331, 12.4067], false, false]
(%i5) [L, v, u] : dgeev (M, true, true);
                             [ - .666642  - .515792 ]
(%o5) [[7.54331, 12.4067], [                       ],
                             [  .745378   - .856714 ]
```

```
                                          [ - .856714  - .745378 ]
                                          [                      ]]
                                          [  .515792   - .666642 ]
(%i6) D : apply (diag_matrix, L);
                        [ 7.54331     0    ]
(%o6)                   [                  ]
                        [    0     12.4067 ]
(%i7) M . v - v . D;
                  [      0.0       - 8.88178E-16 ]
(%o7)             [                              ]
                  [ - 8.88178E-16       0.0      ]
(%i8) transpose (u) . M - D . transpose (u);
                       [ 0.0  - 4.44089E-16 ]
(%o8)                  [                    ]
                       [ 0.0       0.0      ]
```

dgesvd (*A*) Function
dgesvd (*A*, *left_p*, *right_p*) Function

Computes the singular value decomposition (SVD) of a matrix *A*, comprising the singular values and, optionally, the left and right singular vectors. All elements of *A* must be integer or floating point numbers. *A* might or might not be square (same number of rows and columns).

Let *m* be the number of rows, and *n* the number of columns of *A*. The singular value decomposition of *A* comprises three matrices, *U*, *Sigma*, and *V^T*, such that

$A = U.Sigma.V^T$

where *U* is an *m*-by-*m* unitary matrix, *Sigma* is an *m*-by-*n* diagonal matrix, and *V^T* is an *n*-by-*n* unitary matrix.

Let *sigma*[*i*] be a diagonal element of *Sigma*, that is, *Sigma*[*i*, *i*] = *sigma*[*i*]. The elements *sigma*[*i*] are the so-called singular values of *A*; these are real and nonnegative, and returned in descending order. The first $min(m, n)$ columns of *U* and *V* are the left and right singular vectors of *A*. Note that `dgesvd` returns the transpose of *V*, not *V* itself.

`dgesvd(`*A*`)` computes only the singular values of *A*. `dgesvd(`*A*`,` *left_p*`,` *right_p*`)` computes the singular values of *A* and the left singular vectors when *left_p* = `true` and the right singular vectors when *right_p* = `true`.

A list of three items is returned. The first item is a list of the singular values. The second item is `false` or the matrix of left singular vectors. The third item is `false` or the matrix of right singular vectors.

Example:

```
(%i1) load (lapack)$
(%i2) fpprintprec : 6;
(%o2)                          6
(%i3) M: matrix([1, 2, 3], [3.5, 0.5, 8], [-1, 2, -3], [4, 9, 7]);
                         [  1    2    3  ]
                         [               ]
                         [ 3.5  0.5   8  ]
```

```
(%o3)                    [               ]
                         [ - 1   2   - 3 ]
                         [               ]
                         [  4    9    7  ]
(%i4) dgesvd (M);
(%o4)      [[14.4744, 6.38637, .452547], false, false]
(%i5) [sigma, U, VT] : dgesvd (M, true, true);
(%o5) [[14.4744, 6.38637, .452547],
[ - .256731  .00816168   .959029    - .119523 ]
[                                             ]
[ - .526456   .672116   - .206236   - .478091 ]
[                                             ],
[  .107997   - .532278  - .0708315  - 0.83666 ]
[                                             ]
[ - .803287  - .514659  - .180867    .239046  ]
[ - .374486  - .538209  - .755044 ]
[                                 ]
[  .130623   - .836799   0.5317   ]]
[                                 ]
[ - .917986   .100488    .383672  ]
(%i6) m : length (U);
(%o6)                            4
(%i7) n : length (VT);
(%o7)                            3
(%i8) Sigma:
       genmatrix(lambda ([i, j], if i=j then sigma[i] else 0),
                 m, n);
                 [ 14.4744     0        0    ]
                 [                           ]
                 [    0     6.38637     0    ]
(%o8)            [                           ]
                 [    0        0     .452547 ]
                 [                           ]
                 [    0        0        0    ]
(%i9) U . Sigma . VT - M;
          [  1.11022E-15        0.0       1.77636E-15 ]
          [                                           ]
          [  1.33227E-15    1.66533E-15       0.0     ]
(%o9)     [                                           ]
          [ - 4.44089E-16  - 8.88178E-16  4.44089E-16 ]
          [                                           ]
          [  8.88178E-16    1.77636E-15   8.88178E-16 ]
(%i10) transpose (U) . U;
       [     1.0      5.55112E-17    2.498E-16     2.77556E-17  ]
       [                                                        ]
       [ 5.55112E-17      1.0       5.55112E-17    4.16334E-17  ]
(%o10) [                                                        ]
       [  2.498E-16   5.55112E-17       1.0       - 2.08167E-16 ]
       [                                                        ]
```

```
       [ 2.77556E-17  4.16334E-17  - 2.08167E-16       1.0      ]
(%i11) VT . transpose (VT);
          [      1.0           0.0      - 5.55112E-17 ]
          [                                           ]
(%o11)    [      0.0           1.0       5.55112E-17  ]
          [                                           ]
          [ - 5.55112E-17  5.55112E-17       1.0      ]
```

dlange (*norm, A*) Function
zlange (*norm, A*) Function

Computes a norm or norm-like function of the matrix A.

`max` Compute $max(abs(A(i,j)))$ where i and j range over the rows and columns, respectively, of A. Note that this function is not a proper matrix norm.

`one_norm` Compute the $L[1]$ norm of A, that is, the maximum of the sum of the absolute value of elements in each column.

`inf_norm` Compute the $L[inf]$ norm of A, that is, the maximum of the sum of the absolute value of elements in each row.

`frobenius`
Compute the Frobenius norm of A, that is, the square root of the sum of squares of the matrix elements.

58 lbfgs

58.1 Introduction to lbfgs

`lbfgs` is an implementation of the L-BFGS algorithm [1] to solve unconstrained minimization problems via a limited-memory quasi-Newton (BFGS) algorithm. It is called a limited-memory method because a low-rank approximation of the Hessian matrix inverse is stored instead of the entire Hessian inverse. The program was originally written in Fortran [2] by Jorge Nocedal, incorporating some functions originally written by Jorge J. Moré and David J. Thuente, and translated into Lisp automatically via the program `f2cl`. The Maxima package `lbfgs` comprises the translated code plus an interface function which manages some details.

References:

[1] D. Liu and J. Nocedal. "On the limited memory BFGS method for large scale optimization". *Mathematical Programming B* 45:503–528 (1989)

[2] `http://netlib.org/opt/lbfgs_um.shar`

58.2 Functions and Variables for lbfgs

lbfgs (*FOM, 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 $normgradFOM < epsilonmax(1, normX)$.

The algorithm applied is a limited-memory quasi-Newton (BFGS) algorithm [1]. It is called a limited-memory method because a low-rank approximation of the Hessian matrix inverse is stored instead of the entire Hessian inverse. Each iteration of the algorithm is a line search, that is, a search along a ray in the variables *X*, with the search direction computed from the approximate Hessian inverse. The FOM is always decreased by a successful line search. Usually (but not always) the norm of the gradient of FOM also decreases.

iprint controls progress messages printed by `lbfgs`.

`iprint[1]`

iprint`[1]` controls the frequency of progress messages.

`iprint[1] < 0`

No progress messages.

`iprint[1] = 0`

Messages at the first and last iterations.

`iprint[1] > 0`

Print a message every *iprint*`[1]` iterations.

`iprint[2]`

iprint`[2]` controls the verbosity of progress messages.

`iprint[2] = 0`
: Print out iteration count, number of evaluations of *FOM*, value of *FOM*, norm of the gradient of *FOM*, and step length.

`iprint[2] = 1`
: Same as *iprint*`[2] = 0`, plus *X0* and the gradient of *FOM* evaluated at *X0*.

`iprint[2] = 2`
: Same as *iprint*`[2] = 1`, plus values of X at each iteration.

`iprint[2] = 3`
: Same as *iprint*`[2] = 2`, plus the gradient of *FOM* at each iteration.

The columns printed by `lbfgs` are the following.

`I` Number of iterations. It is incremented for each line search.

`NFN` Number of evaluations of the figure of merit.

`FUNC` Value of the figure of merit at the end of the most recent line search.

`GNORM` Norm of the gradient of the figure of merit at the end of the most recent line search.

`STEPLENGTH`
: An internal parameter of the search algorithm.

Additional information concerning details of the algorithm are found in the comments of the original Fortran code [2].

See also `lbfgs_nfeval_max` and `lbfgs_ncorrections`.

References:

[1] D. Liu and J. Nocedal. "On the limited memory BFGS method for large scale optimization". *Mathematical Programming B* 45:503–528 (1989)

[2] `http://netlib.org/opt/lbfgs_um.shar`

Examples:

The same FOM as computed by FGCOMPUTE in the program sdrive.f in the LBFGS package from Netlib. Note that the variables in question are subscripted variables. The FOM has an exact minimum equal to zero at $u[k] = 1$ for $k = 1, ..., 8$.

```
(%i1) load (lbfgs);
(%o1)   /usr/share/maxima/5.10.0cvs/share/lbfgs/lbfgs.mac
(%i2) t1[j] := 1 - u[j];
(%o2)                      t1  := 1 - u
                             j         j
(%i3) t2[j] := 10*(u[j + 1] - u[j]^2);
                                                 2
(%o3)                t2  := 10 (u      - u )
                       j         j + 1    j
(%i4) n : 8;
(%o4)                           8
```

```
(%i5) FOM : sum (t1[2*j - 1]^2 + t2[2*j - 1]^2, j, 1, n/2);
                     2 2          2             2 2          2
(%o5) 100 (u  - u )  + (1 - u )  + 100 (u  - u )  + (1 - u )
            8    7           7           6    5           5
                        2 2           2              2 2           2
         + 100 (u  - u )  + (1 - u )  + 100 (u  - u )  + (1 - u )
                 4    3           3           2    1           1
(%i6) lbfgs (FOM, '[u[1],u[2],u[3],u[4],u[5],u[6],u[7],u[8]],
       [-1.2, 1, -1.2, 1, -1.2, 1, -1.2, 1], 1e-3, [1, 0]);
*************************************************
  N=    8   NUMBER OF CORRECTIONS=25
       INITIAL VALUES
 F=  9.680000000000000D+01   GNORM=  4.657353755084532D+02
*************************************************
```

I	NFN	FUNC	GNORM	STEPLENGTH
1	3	1.651479526340304D+01	4.324359291335977D+00	7.926153934390631D-04
2	4	1.650209316638371D+01	3.575788161060007D+00	1.000000000000000D+00
3	5	1.645461701312851D+01	6.230869903601577D+00	1.000000000000000D+00
4	6	1.636867301275588D+01	1.177589920974980D+01	1.000000000000000D+00
5	7	1.612153014409201D+01	2.292797147151288D+01	1.000000000000000D+00
6	8	1.569118407390628D+01	3.687447158775571D+01	1.000000000000000D+00
7	9	1.510361958398942D+01	4.501931728123680D+01	1.000000000000000D+00
8	10	1.391077875774294D+01	4.526061463810632D+01	1.000000000000000D+00
9	11	1.165625686278198D+01	2.748348965356917D+01	1.000000000000000D+00
10	12	9.859422687859137D+00	2.111494974231644D+01	1.000000000000000D+00
11	13	7.815442521732281D+00	6.110762325766556D+00	1.000000000000000D+00
12	15	7.346380905773160D+00	2.165281166714631D+01	1.285316401779533D-01
13	16	6.330460634066370D+00	1.401220851762050D+01	1.000000000000000D+00
14	17	5.238763939851439D+00	1.702473787613255D+01	1.000000000000000D+00
15	18	3.754016790406701D+00	7.981845727704576D+00	1.000000000000000D+00
16	20	3.001238402309352D+00	3.925482944716691D+00	2.333129631296807D-01
17	22	2.794390709718290D+00	8.243329982546473D+00	2.503577283782332D-01
18	23	2.563783562918759D+00	1.035413426521790D+01	1.000000000000000D+00
19	24	2.019429976377856D+00	1.065187312346769D+01	1.000000000000000D+00
20	25	1.428003167670903D+00	2.475962450826961D+00	1.000000000000000D+00
21	27	1.197874264861340D+00	8.441707983493810D+00	4.303451060808756D-01
22	28	9.023848941942773D-01	1.113189216635162D+01	1.000000000000000D+00
23	29	5.508226405863770D-01	2.380830600326308D+00	1.000000000000000D+00
24	31	3.902893258815567D-01	5.625595816584421D+00	4.834988416524465D-01
25	32	3.207542206990315D-01	1.149444645416472D+01	1.000000000000000D+00
26	33	1.874468266362791D-01	3.632482152880997D+00	1.000000000000000D+00
27	34	9.575763380706598D-02	4.816497446154354D+00	1.000000000000000D+00
28	35	4.085145107543406D-02	2.087009350166495D+00	1.000000000000000D+00
29	36	1.931106001379290D-02	3.886818608498966D+00	1.000000000000000D+00
30	37	6.894000721499670D-03	3.198505796342214D+00	1.000000000000000D+00
31	38	1.443296033051864D-03	1.590265471025043D+00	1.000000000000000D+00

```
32   39  1.571766603154336D-04  3.098257063980634D-01  1.000000000000000D+00
33   40  1.288011776581970D-05  1.207784183577257D-02  1.000000000000000D+00
34   41  1.806140173752971D-06  4.587890233385193D-02  1.000000000000000D+00
35   42  1.769004645459358D-07  1.790537375052208D-02  1.000000000000000D+00
36   43  3.312164100763217D-10  6.782068426119681D-04  1.000000000000000D+00

 THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
 IFLAG = 0
(%o6) [u  = 1.000005339815974, u  = 1.000009942839805,
        1                        2
u  = 1.000005339815974, u  = 1.000009942839805,
 3                       4
u  = 1.000005339815974, u  = 1.000009942839805,
 5                       6
u  = 1.000005339815974, u  = 1.000009942839805]
 7                       8
```

A regression problem. The FOM is the mean square difference between the predicted value $F(X[i])$ and the observed value $Y[i]$. The function F is a bounded monotone function (a so-called "sigmoidal" function). In this example, `lbfgs` computes approximate values for the parameters of F and `plot2d` displays a comparison of F with the observed data.

```
(%i1) load (lbfgs);
(%o1)   /usr/share/maxima/5.10.0cvs/share/lbfgs/lbfgs.mac
(%i2) FOM : '((1/length(X))*sum((F(X[i]) - Y[i])^2, i, 1,
                                                    length(X)));
                                  2
                      sum((F(X ) - Y ) , i, 1, length(X))
                              i     i
(%o2)                 ----------------------------------
                                length(X)
(%i3) X : [1, 2, 3, 4, 5];
(%o3)                      [1, 2, 3, 4, 5]
(%i4) Y : [0, 0.5, 1, 1.25, 1.5];
(%o4)                  [0, 0.5, 1, 1.25, 1.5]
(%i5) F(x) := A/(1 + exp(-B*(x - C)));
                                      A
(%o5)              F(x) := ----------------------
                           1 + exp((- B) (x - C))
(%i6) ''FOM;
                  A                2             A                2
(%o6) ((----------------- - 1.5)  + (----------------- - 1.25)
          - B (5 - C)                  - B (4 - C)
        %e             + 1           %e             + 1
            A              2             A                2
 + (----------------- - 1)  + (----------------- - 0.5)
      - B (3 - C)                  - B (2 - C)
    %e             + 1           %e             + 1
              2
```

```
                A
 + --------------------)/5
      - B (1 - C)     2
   (%e            + 1)
(%i7) estimates : lbfgs (FOM, '[A, B, C], [1, 1, 1], 1e-4, [1, 0]);
************************************************
  N=    3   NUMBER OF CORRECTIONS=25
        INITIAL VALUES
 F=  1.348738534246918D-01   GNORM=  2.000215531936760D-01
************************************************

I  NFN  FUNC                    GNORM                   STEPLENGTH

1    3  1.177820636622582D-01  9.893138394953992D-02  8.554435968992371D-01
2    6  2.302653892214013D-02  1.180098521565904D-01  2.100000000000000D+01
3    8  1.496348495303005D-02  9.611201567691633D-02  5.257340567840707D-01
4    9  7.900460841091139D-03  1.325041647391314D-02  1.000000000000000D+00
5   10  7.314495451266917D-03  1.510670810312237D-02  1.000000000000000D+00
6   11  6.750147275936680D-03  1.914964958023047D-02  1.000000000000000D+00
7   12  5.850716021108205D-03  1.028089194579363D-02  1.000000000000000D+00
8   13  5.778664230657791D-03  3.676866074530332D-04  1.000000000000000D+00
9   14  5.777818823650782D-03  3.010740179797255D-04  1.000000000000000D+00

 THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
 IFLAG = 0
(%o7) [A = 1.461933911464101, B = 1.601593973254802,
                                              C = 2.528933072164854]
(%i8) plot2d ([F(x), [discrete, X, Y]], [x, -1, 6]), ''estimates;
(%o8)
```

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`.

59 lindstedt

59.1 Functions and Variables for lindstedt

Lindstedt (*eq,pvar,torder,ic*) Function

This is a first pass at a Lindstedt code. It can solve problems with initial conditions entered, which can be arbitrary constants, (just not *%k1* and *%k2*) where the initial conditions on the perturbation equations are $z[i] = 0, z'[i] = 0$ for $i > 0$. *ic* is the list of initial conditions.

Problems occur when initial conditions are not given, as the constants in the perturbation equations are the same as the zero order equation solution. Also, problems occur when the initial conditions for the perturbation equations are not $z[i] = 0, z'[i] = 0$ for $i > 0$, such as the Van der Pol equation.

Example:

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

(%i2) load("lindstedt")$

(%i3) Lindstedt('diff(x,t,2)+x-(e*x^3)/6,e,2,[1,0]);
                2
               e  (cos(5 T) - 24 cos(3 T) + 23 cos(T))
(%o3) [[[- -------------------------------------
                           36864
     e (cos(3 T) - cos(T))
   - --------------------- + cos(T)],
               192
               2
           7 e    e
   T = (- ---- - -- + 1) t]]
           3072   16
```

To use this function write first `load("makeOrders")` and `load("lindstedt")`.

60 linearalgebra

60.1 Introduction to linearalgebra

`linearalgebra` is a collection of functions for linear algebra.

Example:

```
(%i1) M : matrix ([1, 2], [1, 2]);
                              [ 1  2 ]
(%o1)                         [      ]
                              [ 1  2 ]
(%i2) nullspace (M);
                                 [  1  ]
                                 [     ]
(%o2)                       span([   1 ])
                                 [ - - ]
                                 [   2 ]
(%i3) columnspace (M);
                                  [ 1 ]
(%o3)                          span([   ])
                                  [ 1 ]
(%i4) ptriangularize (M - z*ident(2), z);
                             [ 1   2 - z   ]
(%o4)                        [             ]
                             [           2 ]
                             [ 0  3 z - z  ]
(%i5) M : matrix ([1, 2, 3], [4, 5, 6], [7, 8, 9]) - z*ident(3);
                        [ 1 - z    2      3   ]
                        [                     ]
(%o5)                   [   4    5 - z    6   ]
                        [                     ]
                        [   7      8    9 - z ]
(%i6) MM : ptriangularize (M, z);
                 [ 4  5 - z            6           ]
                 [                                 ]
                 [                2                ]
                 [     66        z    102 z   132   ]
                 [ 0   --      - -- + ----- + ---   ]
(%o6)            [     49        7     49     49    ]
                 [                                 ]
                 [                3        2       ]
                 [           49 z    245 z    147 z ]
                 [ 0    0    ----- - ------ - ----- ]
                 [            264      88      44   ]
(%i7) algebraic : true;
(%o7)                          true
(%i8) tellrat (MM [3, 3]);
                           3       2
(%o8)                  [z  - 15 z  - 18 z]
```

```
(%i9) MM : ratsimp (MM);
                [ 4  5 - z           6          ]
                [                               ]
                [               2               ]
(%o9)           [     66      7 z  - 102 z - 132 ]
                [ 0   --    - ------------------ ]
                [     49             49         ]
                [                               ]
                [ 0    0            0           ]
(%i10) nullspace (MM);
                         [        1         ]
                         [                  ]
                         [   2              ]
                         [  z  - 14 z - 16  ]
                         [  --------------  ]
(%o10)             span([        8         ])
                         [                  ]
                         [    2             ]
                         [   z  - 18 z - 12 ]
                         [ - --------------- ]
                         [         12       ]
(%i11) M : matrix ([1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12],
                   [13, 14, 15, 16]);
                        [ 1   2   3   4  ]
                        [                ]
                        [ 5   6   7   8  ]
(%o11)                  [                ]
                        [ 9   10  11  12 ]
                        [                ]
                        [ 13  14  15  16 ]
(%i12) columnspace (M);
                             [ 1  ]  [ 2  ]
                             [    ]  [    ]
                             [ 5  ]  [ 6  ]
(%o12)                 span([    ], [    ])
                             [ 9  ]  [ 10 ]
                             [    ]  [    ]
                             [ 13 ]  [ 14 ]
(%i13) apply ('orthogonal_complement, args (nullspace (transpose (M))));
                             [ 0 ]  [  1  ]
                             [   ]  [     ]
                             [ 1 ]  [  0  ]
(%o13)                 span([   ], [     ])
                             [ 2 ]  [ - 1 ]
                             [   ]  [     ]
                             [ 3 ]  [ - 2 ]
```

60.2 Functions and Variables for linearalgebra

addmatrices (*f, M_1, ..., M_n*) Function

Using the function *f* as the addition function, return the sum of the matrices *M_1*, ..., *M_n*. The function *f* must accept any number of arguments (a Maxima nary function).

Examples:

```
(%i1) m1 : matrix([1,2],[3,4])$
(%i2) m2 : matrix([7,8],[9,10])$
(%i3) addmatrices('max,m1,m2);
(%o3) matrix([7,8],[9,10])
(%i4) addmatrices('max,m1,m2,5*m1);
(%o4) matrix([7,10],[15,20])
```

blockmatrixp (*M*) Function

Return true if and only if *M* is a matrix and every entry of *M* is a matrix.

columnop (*M, i, j, theta*) Function

If *M* is a matrix, return the matrix that results from doing the column operation `C_i <- C_i - theta * C_j`. If *M* doesn't have a row *i* or *j*, signal an error.

columnswap (*M, i, j*) Function

If *M* is a matrix, swap columns *i* and *j*. If *M* doesn't have a column *i* or *j*, signal an error.

columnspace (*M*) Function

If *M* is a matrix, return `span (v_1, ..., v_n)`, where the set `{v_1, ..., v_n}` is a basis for the column space of *M*. The span of the empty set is `{0}`. Thus, when the column space has only one member, return `span ()`.

copy (*e*) Function

Return a copy of the Maxima expression *e*. Although *e* can be any Maxima expression, the copy function is the most useful when *e* is either a list or a matrix; consider:

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

Let's try the same experiment, but this time let *mm* be a copy of *m*

```
(%i6) m : [1,[2,3]]$
(%i7) mm : copy(m)$
(%i8) mm[2][1] : x$
(%i9) m;
(%o9)                       [1,[2,3]]
(%i10) mm;
(%o10)                      [1,[x,3]]
```

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

cholesky (*M*) Function
cholesky (*M*, *field*) Function

Return the Cholesky factorization of the matrix selfadjoint (or hermitian) matrix *M*. The second argument defaults to 'generalring.' For a description of the possible values for *field*, see `lu_factor`.

ctranspose (*M*) Function

Return the complex conjugate transpose of the matrix *M*. The function `ctranspose` uses `matrix_element_transpose` to transpose each matrix element.

diag_matrix (*d_1*, *d_2*,...,*d_n*) Function

Return a diagonal matrix with diagonal entries *d_1*, *d_2*,...,*d_n*. When the diagonal entries are matrices, the zero entries of the returned matrix are zero matrices of the appropriate size; for example:

```
(%i1) diag_matrix(diag_matrix(1,2),diag_matrix(3,4));

                              [ [ 1  0 ]  [ 0  0 ] ]
                              [ [      ]  [      ] ]
                              [ [ 0  2 ]  [ 0  0 ] ]
(%o1)                         [                    ]
                              [ [ 0  0 ]  [ 3  0 ] ]
                              [ [      ]  [      ] ]
                              [ [ 0  0 ]  [ 0  4 ] ]
(%i2) diag_matrix(p,q);

                                    [ p  0 ]
(%o2)                               [      ]
                                    [ 0  q ]
```

dotproduct (*u*, *v*) Function

Return the dotproduct of vectors *u* and *v*. This is the same as `conjugate (transpose (u)) . v`. The arguments *u* and *v* must be column vectors.

eigens_by_jacobi (*A*) Function
eigens_by_jacobi (*A*, *field_type*) Function

Computes the eigenvalues and eigenvectors of *A* by the method of Jacobi rotations. *A* must be a symmetric matrix (but it need not be positive definite nor positive semidefinite). *field_type* indicates the computational field, either `floatfield` or `bigfloatfield`. If *field_type* is not specified, it defaults to `floatfield`.

The elements of *A* must be numbers or expressions which evaluate to numbers via `float` or `bfloat` (depending on *field_type*).

Examples:

```
(%i1) S: matrix([1/sqrt(2), 1/sqrt(2)],[-1/sqrt(2), 1/sqrt(2)]);
                        [     1         1    ]
                        [  -------   ------- ]
                        [  sqrt(2)   sqrt(2) ]
(%o1)                   [                    ]
```

```
                     [      1         1    ]
                     [ - -------   ------- ]
                     [   sqrt(2)   sqrt(2) ]
(%i2) L : matrix ([sqrt(3), 0], [0, sqrt(5)]);
                      [ sqrt(3)     0    ]
(%o2)                 [                  ]
                      [    0     sqrt(5) ]
(%i3) M : S . L . transpose (S);
            [ sqrt(5)   sqrt(3)  sqrt(5)   sqrt(3) ]
            [ ------- + -------  ------- - ------- ]
            [    2         2        2         2    ]
(%o3)       [                                      ]
            [ sqrt(5)   sqrt(3)  sqrt(5)   sqrt(3) ]
            [ ------- - -------  ------- + ------- ]
            [    2         2        2         2    ]
(%i4) eigens_by_jacobi (M);
The largest percent change was 0.1454972243679
The largest percent change was 0.0
number of sweeps: 2
number of rotations: 1
(%o4) [[1.732050807568877, 2.23606797749979],
                         [  0.70710678118655   0.70710678118655 ]
                         [                                      ]]
                         [ - 0.70710678118655  0.70710678118655 ]
(%i5) float ([[sqrt(3), sqrt(5)], S]);
(%o5) [[1.732050807568877, 2.23606797749979],
                         [  0.70710678118655   0.70710678118655 ]
                         [                                      ]]
                         [ - 0.70710678118655  0.70710678118655 ]
(%i6) eigens_by_jacobi (M, bigfloatfield);
The largest percent change was 1.454972243679028b-1
The largest percent change was 0.0b0
number of sweeps: 2
number of rotations: 1
(%o6) [[1.732050807568877b0, 2.23606797749979b0],
                  [  7.071067811865475b-1   7.071067811865475b-1 ]
                  [                                              ]]
                  [ - 7.071067811865475b-1  7.071067811865475b-1 ]
```

get_lu_factors (*x*) Function

When *x* = `lu_factor` (*A*), then `get_lu_factors` returns a list of the form `[P, L, U]`, where *P* is a permutation matrix, *L* is lower triangular with ones on the diagonal, and *U* is upper triangular, and *A* = *P L U*.

hankel (*col*) Function

hankel (*col, row*) Function

Return a Hankel matrix *H*. The first column of *H* is *col*; except for the first entry, the last row of *H* is *row*. The default for *row* is the zero vector with the same length as *col*.

hessian (*f*, *x*) Function

Returns the Hessian matrix of *f* with respect to the list of variables *x*. The `(i, j)`-th element of the Hessian matrix is `diff(`*f*`, `*x*`[i], 1, `*x*`[j], 1)`.

Examples:

```
(%i1) hessian (x * sin (y), [x, y]);
                       [   0       cos(y)   ]
(%o1)                  [                    ]
                       [ cos(y)  - x sin(y) ]
(%i2) depends (F, [a, b]);
(%o2)                        [F(a, b)]
(%i3) hessian (F, [a, b]);
                           [   2      2   ]
                           [  d F    d F  ]
                           [  ---   ----- ]
                           [    2   da db ]
                           [  da          ]
(%o3)                      [              ]
                           [   2      2   ]
                           [  d F    d F  ]
                           [ -----   ---  ]
                           [ da db     2  ]
                           [         db   ]
```

hilbert_matrix (*n*) Function

Return the *n* by *n* Hilbert matrix. When *n* isn't a positive integer, signal an error.

identfor (*M*) Function

identfor (*M*, *fld*) Function

Return an identity matrix that has the same shape as the matrix *M*. The diagonal entries of the identity matrix are the multiplicative identity of the field *fld*; the default for *fld* is *generalring*.

The first argument *M* should be a square matrix or a non-matrix. When *M* is a matrix, each entry of *M* can be a square matrix – thus *M* can be a blocked Maxima matrix. The matrix can be blocked to any (finite) depth.

See also `zerofor`

invert_by_lu (*M*, (*rng generalring*)) Function

Invert a matrix *M* by using the LU factorization. The LU factorization is done using the ring *rng*.

jacobian (*f*, *x*) Function

Returns the Jacobian matrix of the list of functions *f* with respect to the list of variables *x*. The `(i, j)`-th element of the Jacobian matrix is `diff(`*f*`[i], `*x*`[j])`.

Examples:

```
(%i1) jacobian ([sin (u - v), sin (u * v)], [u, v]);
                    [ cos(v - u)  - cos(v - u) ]
```

```
(%o1)             [                          ]
                  [ v cos(u v)   u cos(u v)  ]
(%i2) depends ([F, G], [y, z]);
(%o2)                  [F(y, z), G(y, z)]
(%i3) jacobian ([F, G], [y, z]);
                             [ dF  dF ]
                             [ --  -- ]
                             [ dy  dz ]
(%o3)                        [        ]
                             [ dG  dG ]
                             [ --  -- ]
                             [ dy  dz ]
```

kronecker_product (*A*, *B*) Function

Return the Kronecker product of the matrices *A* and *B*.

listp (*e*, *p*) Function

listp (*e*) Function

Given an optional argument *p*, return `true` if *e* is a Maxima list and *p* evaluates to `true` for every list element. When `listp` is not given the optional argument, return `true` if *e* is a Maxima list. In all other cases, return `false`.

locate_matrix_entry (*M*, *r_1*, *c_1*, *r_2*, *c_2*, *f*, *rel*) Function

The first argument must be a matrix; the arguments *r_1* through *c_2* determine a sub-matrix of *M* that consists of rows *r_1* through *r_2* and columns *c_1* through *c_2*.

Find a entry in the sub-matrix *M* that satisfies some property. Three cases:

(1) *rel* = `'bool` and *f* a predicate:

Scan the sub-matrix from left to right then top to bottom, and return the index of the first entry that satisfies the predicate *f*. If no matrix entry satisfies *f*, return `false`.

(2) *rel* = `'max` and *f* real-valued:

Scan the sub-matrix looking for an entry that maximizes *f*. Return the index of a maximizing entry.

(3) *rel* = `'min` and *f* real-valued:

Scan the sub-matrix looking for an entry that minimizes *f*. Return the index of a minimizing entry.

lu_backsub (*M*, *b*) Function

When *M* = `lu_factor (`*A*`,` *field*`)`, then `lu_backsub (`*M*`,` *b*`)` solves the linear system *A x* = *b*.

lu_factor (*M*, *field*) Function

Return a list of the form `[`*LU*`,` *perm*`,` *fld*`]`, or `[`*LU*`,` *perm*`,` *fld*`,` *lower-cnd upper-cnd*`]`, where

(1) The matrix *LU* contains the factorization of *M* in a packed form. Packed form means three things: First, the rows of *LU* are permuted according to the list *perm*.

If, for example, *perm* is the list `[3,2,1]`, the actual first row of the LU factorization is the third row of the matrix LU. Second, the lower triangular factor of m is the lower triangular part of LU with the diagonal entries replaced by all ones. Third, the upper triangular factor of M is the upper triangular part of LU.

(2) When the field is either `floatfield` or `complexfield`, the numbers *lower-cnd* and *upper-cnd* are lower and upper bounds for the infinity norm condition number of M. For all fields, the condition number might not be estimated; for such fields, `lu_factor` returns a two item list. Both the lower and upper bounds can differ from their true values by arbitrarily large factors. (See also `mat_cond`.)

The argument M must be a square matrix.

The optional argument *fld* must be a symbol that determines a ring or field. The pre-defined fields and rings are:

(a) `generalring` – the ring of Maxima expressions, (b) `floatfield` – the field of floating point numbers of the type double, (c) `complexfield` – the field of complex floating point numbers of the type double, (d) `crering` – the ring of Maxima CRE expressions, (e) `rationalfield` – the field of rational numbers, (f) `runningerror` – track the all floating point rounding errors, (g) `noncommutingring` – the ring of Maxima expressions where multiplication is the non-commutative dot operator.

When the field is `floatfield`, `complexfield`, or `runningerror`, the algorithm uses partial pivoting; for all other fields, rows are switched only when needed to avoid a zero pivot.

Floating point addition arithmetic isn't associative, so the meaning of 'field' differs from the mathematical definition.

A member of the field `runningerror` is a two member Maxima list of the form `[x,n]`,where x is a floating point number and `n` is an integer. The relative difference between the 'true' value of `x` and `x` is approximately bounded by the machine epsilon times `n`. The running error bound drops some terms that of the order the square of the machine epsilon.

There is no user-interface for defining a new field. A user that is familiar with Common Lisp should be able to define a new field. To do this, a user must define functions for the arithmetic operations and functions for converting from the field representation to Maxima and back. Additionally, for ordered fields (where partial pivoting will be used), a user must define functions for the magnitude and for comparing field members. After that all that remains is to define a Common Lisp structure `mring`. The file `mring` has many examples.

To compute the factorization, the first task is to convert each matrix entry to a member of the indicated field. When conversion isn't possible, the factorization halts with an error message. Members of the field needn't be Maxima expressions. Members of the `complexfield`, for example, are Common Lisp complex numbers. Thus after computing the factorization, the matrix entries must be converted to Maxima expressions.

See also `get_lu_factors`.

Examples:

```
(%i1) w[i,j] := random (1.0) + %i * random (1.0);
```

```
(%o1)           w     := random(1.) + %i random(1.)
                 i, j
(%i2) showtime : true$
Evaluation took 0.00 seconds (0.00 elapsed)
(%i3) M : genmatrix (w, 100, 100)$
Evaluation took 7.40 seconds (8.23 elapsed)
(%i4) lu_factor (M, complexfield)$
Evaluation took 28.71 seconds (35.00 elapsed)
(%i5) lu_factor (M, generalring)$
Evaluation took 109.24 seconds (152.10 elapsed)
(%i6) showtime : false$

(%i7) M : matrix ([1 - z, 3], [3, 8 - z]);
                        [ 1 - z    3   ]
(%o7)                   [              ]
                        [   3    8 - z ]
(%i8) lu_factor (M, generalring);
          [ 1 - z         3        ]
          [                        ]
(%o8)    [[   3            9       ], [1, 2], generalring]
          [ -----  - z - ----- + 8 ]
          [ 1 - z        1 - z     ]
(%i9) get_lu_factors (%);
                   [   1    0 ]  [ 1 - z         3        ]
         [ 1  0 ]  [          ]  [                        ]
(%o9)  [[      ], [   3      ], [                 9       ]]
         [ 0  1 ]  [ -----  1 ]  [   0    - z - ----- + 8 ]
                   [ 1 - z    ]  [              1 - z     ]
(%i10) %[1] . %[2] . %[3];
                        [ 1 - z    3   ]
(%o10)                  [              ]
                        [   3    8 - z ]
```

mat_cond (*M*, *1*) Function
mat_cond (*M*, *inf*) Function

Return the *p*-norm matrix condition number of the matrix *m*. The allowed values for *p* are 1 and *inf*. This function uses the LU factorization to invert the matrix *m*. Thus the running time for `mat_cond` is proportional to the cube of the matrix size; `lu_factor` determines lower and upper bounds for the infinity norm condition number in time proportional to the square of the matrix size.

mat_norm (*M*, *1*) Function
mat_norm (*M*, *inf*) Function
mat_norm (*M*, *frobenius*) Function

Return the matrix *p*-norm of the matrix *M*. The allowed values for *p* are 1, `inf`, and `frobenius` (the Frobenius matrix norm). The matrix *M* should be an unblocked matrix.

matrixp (*e*, *p*) Function
matrixp (*e*) Function

Given an optional argument *p*, return `true` if *e* is a matrix and *p* evaluates to `true` for every matrix element. When `matrixp` is not given an optional argument, return `true` if `e` is a matrix. In all other cases, return `false`.

See also `blockmatrixp`

matrix_size (*M*) Function

Return a two member list that gives the number of rows and columns, respectively of the matrix *M*.

mat_fullunblocker (*M*) Function

If *M* is a block matrix, unblock the matrix to all levels. If *M* is a matrix, return *M*; otherwise, signal an error.

mat_trace (*M*) Function

Return the trace of the matrix *M*. If *M* isn't a matrix, return a noun form. When *M* is a block matrix, `mat_trace(M)` returns the same value as does `mat_trace(mat_unblocker(m))`.

mat_unblocker (*M*) Function

If *M* is a block matrix, unblock *M* one level. If *M* is a matrix, `mat_unblocker (M)` returns *M*; otherwise, signal an error.

Thus if each entry of *M* is matrix, `mat_unblocker (M)` returns an unblocked matrix, but if each entry of *M* is a block matrix, `mat_unblocker (M)` returns a block matrix with one less level of blocking.

If you use block matrices, most likely you'll want to set `matrix_element_mult` to `"."` and `matrix_element_transpose` to `'transpose`. See also `mat_fullunblocker`.

Example:

```
(%i1) A : matrix ([1, 2], [3, 4]);
                              [ 1  2 ]
(%o1)                         [      ]
                              [ 3  4 ]
(%i2) B : matrix ([7, 8], [9, 10]);
                              [ 7  8  ]
(%o2)                         [       ]
                              [ 9  10 ]
(%i3) matrix ([A, B]);
                        [ [ 1  2 ]  [ 7  8  ] ]
(%o3)                   [ [      ]  [       ] ]
                        [ [ 3  4 ]  [ 9  10 ] ]
(%i4) mat_unblocker (%);
                            [ 1  2  7  8  ]
(%o4)                       [             ]
                            [ 3  4  9  10 ]
```

nonnegintegerp (*n*) Function

Return `true` if and only if *n* `>= 0` and *n* is an integer.

nullspace (*M*) Function

If *M* is a matrix, return `span (v_1, ..., v_n)`, where the set `{v_1, ..., v_n}` is a basis for the nullspace of *M*. The span of the empty set is `{0}`. Thus, when the nullspace has only one member, return `span ()`.

nullity (*M*) Function

If *M* is a matrix, return the dimension of the nullspace of *M*.

orthogonal_complement (*v_1, ..., v_n*) Function

Return `span (u_1, ..., u_m)`, where the set `{u_1, ..., u_m}` is a basis for the orthogonal complement of the set `(v_1, ..., v_n)`.

Each vector *v_1* through *v_n* must be a column vector.

polynomialp (*p, L, coeffp, exponp*) Function
polynomialp (*p, L, coeffp*) Function
polynomialp (*p, L*) Function

Return `true` if *p* is a polynomial in the variables in the list *L*, The predicate *coeffp* must evaluate to `true` for each coefficient, and the predicate *exponp* must evaluate to `true` for all exponents of the variables in *L*. If you want to use a non-default value for *exponp*, you must supply *coeffp* with a value even if you want to use the default for *coeffp*.

`polynomialp (`*p*`, `*L*`, `*coeffp*`)` is equivalent to `polynomialp (`*p*`, `*L*`, `*coeffp*`, 'nonnegintegerp)`.

`polynomialp (`*p*`, `*L*`)` is equivalent to `polynomialp (`*p*`, L, 'constantp, 'nonnegintegerp)`.

The polynomial needn't be expanded:

```
(%i1) polynomialp ((x + 1)*(x + 2), [x]);
(%o1)                         true
(%i2) polynomialp ((x + 1)*(x + 2)^a, [x]);
(%o2)                         false
```

An example using non-default values for coeffp and exponp:

```
(%i1) polynomialp ((x + 1)*(x + 2)^(3/2), [x], numberp, numberp);
(%o1)                         true
(%i2) polynomialp ((x^(1/2) + 1)*(x + 2)^(3/2), [x], numberp,
                                                        numberp);
(%o2)                         true
```

Polynomials with two variables:

```
(%i1) polynomialp (x^2 + 5*x*y + y^2, [x]);
(%o1)                         false
(%i2) polynomialp (x^2 + 5*x*y + y^2, [x, y]);
(%o2)                         true
```

polytocompanion (*p, x*) Function

If *p* is a polynomial in *x*, return the companion matrix of *p*. For a monic polynomial *p* of degree *n*, we have `p = (-1)^n charpoly (polytocompanion (p, x))`.

When *p* isn't a polynomial in *x*, signal an error.

ptriangularize (*M, v*) Function

If *M* is a matrix with each entry a polynomial in *v*, return a matrix *M2* such that

(1) *M2* is upper triangular,

(2) *M2* = *E_n* ... *E_1* *M*, where *E_1* through *E_n* are elementary matrices whose entries are polynomials in *v*,

(3) `|det (M)| = |det (M2)|`,

Note: This function doesn't check that every entry is a polynomial in *v*.

rowop (*M, i, j, theta*) Function

If *M* is a matrix, return the matrix that results from doing the row operation `R_i <- R_i - theta * R_j`. If *M* doesn't have a row *i* or *j*, signal an error.

rank (*M*) Function

Return the rank of that matrix *M*. The rank is the dimension of the column space. Example:

```
(%i1) rank(matrix([1,2],[2,4]));
(%o1)                                1
(%i2) rank(matrix([1,b],[c,d]));
Proviso:  {d - b c # 0}
(%o2)                                2
```

rowswap (*M, i, j*) Function

If *M* is a matrix, swap rows *i* and *j*. If *M* doesn't have a row *i* or *j*, signal an error.

toeplitz (*col*) Function

toeplitz (*col, row*) Function

Return a Toeplitz matrix *T*. The first first column of *T* is *col*; except for the first entry, the first row of *T* is *row*. The default for *row* is complex conjugate of *col*. Example:

```
(%i1)  toeplitz([1,2,3],[x,y,z]);

                                    [ 1  y  z ]
                                    [         ]
(%o1)                               [ 2  1  y ]
                                    [         ]
                                    [ 3  2  1 ]
(%i2)  toeplitz([1,1+%i]);

                                [   1     1 - %I ]
(%o2)                           [                ]
                                [ %I + 1    1    ]
```

vandermonde_matrix ([*x_1, ..., x_n*]) Function

Return a *n* by *n* matrix whose *i*-th row is `[1, x_i, x_i^2, ... x_i^(n-1)]`.

zerofor (*M*) Function

zerofor (*M, fld*) Function

Return a zero matrix that has the same shape as the matrix *M*. Every entry of the zero matrix is the additive identity of the field *fld*; the default for *fld* is *generalring*.

The first argument *M* should be a square matrix or a non-matrix. When *M* is a matrix, each entry of *M* can be a square matrix – thus *M* can be a blocked Maxima matrix. The matrix can be blocked to any (finite) depth.

See also `identfor`

zeromatrixp (*M*) Function

If *M* is not a block matrix, return `true` if `is (equal (e, 0))` is true for each element *e* of the matrix *M*. If *M* is a block matrix, return `true` if `zeromatrixp` evaluates to `true` for each element of *e*.

61 lsquares

61.1 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.

61.2 Functions and Variables for lsquares

lsquares_estimates (*D*, *x*, *e*, *a*) Function
lsquares_estimates (*D*, *x*, *e*, *a*, *initial* = *L*, *tol* = *t*) Function

Estimate parameters *a* to best fit the equation *e* in the variables *x* and *a* to the data *D*, as determined by the method of least squares. `lsquares_estimates` first seeks an exact solution, and if that fails, then seeks an approximate solution.

The return value is a list of lists of equations of the form `[a = ..., b = ..., c = ...]`. Each element of the list is a distinct, equivalent minimum of the mean square error.

The data *D* must be a matrix. Each row is one datum (which may be called a 'record' or 'case' in some contexts), and each column contains the values of one variable across all data. The list of variables *x* gives a name for each column of *D*, even the columns which do not enter the analysis. The list of parameters *a* gives the names of the parameters for which estimates are sought. The equation *e* is an expression or equation in the variables *x* and *a*; if *e* is not an equation, it is treated the same as *e* `= 0`.

Additional arguments to `lsquares_estimates` are specified as equations and passed on verbatim to the function `lbfgs` which is called to find estimates by a numerical method when an exact result is not found.

If some exact solution can be found (via `solve`), the data *D* may contain non-numeric values. However, if no exact solution is found, each element of *D* must have a numeric value. This includes numeric constants such as `%pi` and `%e` as well as literal numbers (integers, rationals, ordinary floats, and bigfloats). Numerical calculations are carried out with ordinary floating-point arithmetic, so all other kinds of numbers are converted to ordinary floats for calculations.

`load(lsquares)` loads this function.

See also `lsquares_estimates_exact`, `lsquares_estimates_approximate`, `lsquares_mse`, `lsquares_residuals`, and `lsquares_residual_mse`.

Examples:

A problem for which an exact solution is found.

```
(%i1) load (lsquares)$
(%i2) M : matrix (
        [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                                    [ 1  1  1 ]
                                    [         ]
```

```
                                    [ 3       ]
                                    [ -  1  2 ]
                                    [ 2       ]
                                    [         ]
(%o2)                               [ 9       ]
                                    [ -  2  1 ]
                                    [ 4       ]
                                    [         ]
                                    [ 3  2  2 ]
                                    [         ]
                                    [ 2  2  1 ]
(%i3) lsquares_estimates (
        M, [z,x,y], (z+D)^2 = A*x+B*y+C, [A,B,C,D]);
                         59        27      10921        107
(%o3)          [[A = - --, B = - --, C = -----, D = - ---]]
                         16        16      1024         32
```

A problem for which no exact solution is found, so `lsquares_estimates` resorts to numerical approximation.

```
(%i1) load (lsquares)$
(%i2) M : matrix ([1, 1], [2, 7/4], [3, 11/4], [4, 13/4]);
                                    [ 1  1  ]
                                    [       ]
                                    [    7  ]
                                    [ 2  -  ]
                                    [    4  ]
                                    [       ]
(%o2)                               [    11 ]
                                    [ 3  -- ]
                                    [    4  ]
                                    [       ]
                                    [    13 ]
                                    [ 4  -- ]
                                    [    4  ]
(%i3) lsquares_estimates (
  M, [x,y], y=a*x^b+c, [a,b,c], initial=[3,3,3], iprint=[-1,0]);
(%o3) [[a = 1.387365874920637, b = .7110956639593767,
                                          c = - .4142705622439105]]
```

lsquares_estimates_exact (*MSE*, *a*) Function

Estimate parameters *a* to minimize the mean square error *MSE*, by constructing a system of equations and attempting to solve them symbolically via `solve`. The mean square error is an expression in the parameters *a*, such as that returned by `lsquares_mse`.

The return value is a list of lists of equations of the form `[a = ..., b = ..., c = ...]`. The return value may contain zero, one, or two or more elements. If two or more elements are returned, each represents a distinct, equivalent minimum of the mean square error.

See also `lsquares_estimates`, `lsquares_estimates_approximate`, `lsquares_mse`, `lsquares_residuals`, and `lsquares_residual_mse`.

Example:

```
(%i1) load (lsquares)$
(%i2) M : matrix (
         [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                              [ 1  1  1 ]
                              [         ]
                              [ 3       ]
                              [ -  1  2 ]
                              [ 2       ]
                              [         ]
(%o2)                         [ 9       ]
                              [ -  2  1 ]
                              [ 4       ]
                              [         ]
                              [ 3  2  2 ]
                              [         ]
                              [ 2  2  1 ]
(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);
               5
              ====
              \                  2                          2
               >    ((D + M      )  - C - M      B - M      A)
              /              i, 1            i, 3       i, 2
              ====
              i = 1
(%o3)         ------------------------------------------------
                                    5
(%i4) lsquares_estimates_exact (mse, [A, B, C, D]);
                     59         27       10921         107
(%o4)         [[A = - --, B = - --, C = -----, D = - ---]]
                     16         16       1024          32
```

lsquares_estimates_approximate (*MSE*, *a*, *initial* = *L*, *tol* = *t*) Function

Estimate parameters *a* to minimize the mean square error *MSE*, via the numerical minimization function `lbfgs`. The mean square error is an expression in the parameters *a*, such as that returned by `lsquares_mse`.

The solution returned by `lsquares_estimates_approximate` is a local (perhaps global) minimum of the mean square error. For consistency with `lsquares_estimates_exact`, the return value is a nested list which contains one element, namely a list of equations of the form `[a = ..., b = ..., c = ...]`.

Additional arguments to `lsquares_estimates_approximate` are specified as equations and passed on verbatim to the function `lbfgs`.

MSE must evaluate to a number when the parameters are assigned numeric values. This requires that the data from which *MSE* was constructed comprise only numeric constants such as `%pi` and `%e` and literal numbers (integers, rationals, ordinary floats,

and bigfloats). Numerical calculations are carried out with ordinary floating-point arithmetic, so all other kinds of numbers are converted to ordinary floats for calculations.

`load(lsquares)` loads this function.

See also `lsquares_estimates`, `lsquares_estimates_exact`, `lsquares_mse`, `lsquares_residuals`, and `lsquares_residual_mse`.

Example:

```
(%i1) load (lsquares)$
(%i2) M : matrix (
         [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                              [ 1  1  1 ]
                              [         ]
                              [ 3       ]
                              [ -  1  2 ]
                              [ 2       ]
                              [         ]
(%o2)                         [ 9       ]
                              [ -  2  1 ]
                              [ 4       ]
                              [         ]
                              [ 3  2  2 ]
                              [         ]
                              [ 2  2  1 ]
(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);
           5
          ====
          \                  2                          2
           >    ((D + M    )  - C - M     B - M     A)
          /            i, 1          i, 3      i, 2
          ====
          i = 1
(%o3)     ------------------------------------------
                               5
(%i4) lsquares_estimates_approximate (
                mse, [A, B, C, D], iprint = [-1, 0]);
(%o4) [[A = - 3.67850494740174, B = - 1.683070351177813,
                   C = 10.63469950148635, D = - 3.340357993175206]]
```

lsquares_mse (*D*, *x*, *e*) Function

Returns the mean square error (MSE), a summation expression, for the equation *e* in the variables *x*, with data *D*.

The MSE is defined as:

$$\frac{1}{n}\sum_{i=1}^{n}\left(\mathrm{lhs}(e_i) - \mathrm{rhs}(e_i)\right)^2$$

$$\frac{\text{====}}{\text{i = 1}}$$

$$\frac{}{\text{n}}$$

where n is the number of data and e[i] is the equation e evaluated with the variables in x assigned values from the i-th datum, D[i].

load(lsquares) loads this function.

Example:

```
(%i1) load (lsquares)$
(%i2) M : matrix (
         [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                              [ 1  1  1 ]
                              [         ]
                              [ 3       ]
                              [ -  1  2 ]
                              [ 2       ]
                              [         ]
(%o2)                         [ 9       ]
                              [ -  2  1 ]
                              [ 4       ]
                              [         ]
                              [ 3  2  2 ]
                              [         ]
                              [ 2  2  1 ]
(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);
           5
          ====
          \                  2                          2
           >    ((D + M    )  - C - M     B - M     A)
          /             i, 1          i, 3      i, 2
          ====
          i = 1
(%o3)     ------------------------------------------
                                5
(%i4) diff (mse, D);
          5
         ====
         \                              2
      4   >    (D + M    ) ((D + M    )  - C - M     B - M     A)
         /           i, 1          i, 1          i, 3      i, 2
         ====
         i = 1
(%o4) ------------------------------------------------------------
                                  5
(%i5) ''mse, nouns;
                2                  2         9 2                2
(%o5) (((D + 3)  - C - 2 B - 2 A)  + ((D + -)  - C - B - 2 A)
                                               4
```

```
          2               2         3 2               2
+ ((D + 2)  - C - B - 2 A)  + ((D + -)  - C - 2 B - A)
                                    2
          2            2
+ ((D + 1)  - C - B - A) )/5
```

lsquares_residuals (*D*, *x*, *e*, *a*) Function

Returns the residuals for the equation *e* with specified parameters *a* and data *D*.

D is a matrix, *x* is a list of variables, *e* is an equation or general expression; if not an equation, *e* is treated as if it were `e = 0`. *a* is a list of equations which specify values for any free parameters in *e* aside from *x*.

The residuals are defined as:

```
                          lhs(e ) - rhs(e )
                               i         i
```

where *e*`[i]` is the equation *e* evaluated with the variables in *x* assigned values from the `i`-th datum, *D*`[i]`, and assigning any remaining free variables from *a*.

`load(lsquares)` loads this function.

Example:

```
(%i1) load (lsquares)$
(%i2) M : matrix (
         [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                                    [ 1  1  1 ]
                                    [         ]
                                    [ 3       ]
                                    [ -  1  2 ]
                                    [ 2       ]
                                    [         ]
(%o2)                               [ 9       ]
                                    [ -  2  1 ]
                                    [ 4       ]
                                    [         ]
                                    [ 3  2  2 ]
                                    [         ]
                                    [ 2  2  1 ]
(%i3) a : lsquares_estimates (
          M, [z,x,y], (z+D)^2 = A*x+B*y+C, [A,B,C,D]);
                          59        27      10921        107
(%o3)           [[A = - --, B = - --, C = -----, D = - ---]]
                          16        16      1024         32
(%i4) lsquares_residuals (
          M, [z,x,y], (z+D)^2 = A*x+B*y+C, first(a));
                           13    13    13  13  13
(%o4)                     [--, - --, - --, --, --]
                           64    64    32  64  64
```

lsquares_residual_mse (*D*, *x*, *e*, *a*) Function

Returns the residual mean square error (MSE) for the equation *e* with specified parameters *a* and data *D*.

The residual MSE is defined as:

$$\frac{\sum_{i=1}^{n} (\mathrm{lhs}(e_i) - \mathrm{rhs}(e_i))^2}{n}$$

where *e*`[i]` is the equation *e* evaluated with the variables in *x* assigned values from the `i`-th datum, *D*`[i]`, and assigning any remaining free variables from *a*.

`load(lsquares)` loads this function.

Example:

```
(%i1) load (lsquares)$
(%i2) M : matrix (
         [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                              [ 1  1  1 ]
                              [         ]
                              [ 3       ]
                              [ -  1  2 ]
                              [ 2       ]
                              [         ]
(%o2)                         [ 9       ]
                              [ -  2  1 ]
                              [ 4       ]
                              [         ]
                              [ 3  2  2 ]
                              [         ]
                              [ 2  2  1 ]
(%i3) a : lsquares_estimates (
             M, [z,x,y], (z+D)^2 = A*x+B*y+C, [A,B,C,D]);

                    59        27      10921        107
(%o3)     [[A = - --, B = - --, C = -----, D = - ---]]
                    16        16      1024         32
(%i4) lsquares_residual_mse (
             M, [z,x,y], (z + D)^2 = A*x + B*y + C, first (a));
                               169
(%o4)                          ----
                               2560
```

plsquares (*Mat,VarList,depvars*) Function
plsquares (*Mat,VarList,depvars,maxexpon*) Function
plsquares (*Mat,VarList,depvars,maxexpon,maxdegree*) Function

Multivariable polynomial adjustment of a data table by the "least squares" method. *Mat* is a matrix containing the data, *VarList* is a list of variable names (one for each Mat column, but use `"-"` instead of varnames to ignore Mat columns), *depvars* is the

name of a dependent variable or a list with one or more names of dependent variables (which names should be in *VarList*), *maxexpon* is the optional maximum exponent for each independent variable (1 by default), and *maxdegree* is the optional maximum polynomial degree (*maxexpon* by default); note that the sum of exponents of each term must be equal or smaller than *maxdegree*, and if `maxdgree = 0` then no limit is applied.

If *depvars* is the name of a dependent variable (not in a list), `plsquares` returns the adjusted polynomial. If *depvars* is a list of one or more dependent variables, `plsquares` returns a list with the adjusted polynomial(s). The Coefficients of Determination are displayed in order to inform about the goodness of fit, which ranges from 0 (no correlation) to 1 (exact correlation). These values are also stored in the global variable *DETCOEF* (a list if *depvars* is a list).

A simple example of multivariable linear adjustment:

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

(%i2) plsquares(matrix([1,2,0],[3,5,4],[4,7,9],[5,8,10]),
                [x,y,z],z);
     Determination Coefficient for z = .9897039897039897
                        11 y - 9 x - 14
(%o2)               z = ---------------
                                3
```

The same example without degree restrictions:

```
(%i3) plsquares(matrix([1,2,0],[3,5,4],[4,7,9],[5,8,10]),
                [x,y,z],z,1,0);
     Determination Coefficient for z = 1.0
                     x y + 23 y - 29 x - 19
(%o3)             z = ----------------------
                                6
```

How many diagonals does a N-sides polygon have? What polynomial degree should be used?

```
(%i4) plsquares(matrix([3,0],[4,2],[5,5],[6,9],[7,14],[8,20]),
                [N,diagonals],diagonals,5);
     Determination Coefficient for diagonals = 1.0
                                 2
                                N  - 3 N
(%o4)               diagonals = --------
                                  2
(%i5) ev(%, N=9);   /* Testing for a 9 sides polygon */
(%o5)                 diagonals = 27
```

How many ways do we have to put two queens without they are threatened into a n x n chessboard?

```
(%i6) plsquares(matrix([0,0],[1,0],[2,0],[3,8],[4,44]),
                [n,positions],[positions],4);
     Determination Coefficient for [positions] = [1.0]
                        4       3      2
                     3 n  - 10 n  + 9 n  - 2 n
```

```
(%o6)    [positions = -------------------------]
                                  6
(%i7) ev(%[1], n=8); /* Testing for a (8 x 8) chessboard */
(%o7)                positions = 1288
```

An example with six dependent variables:

```
(%i8) mtrx:matrix([0,0,0,0,0,1,1,1],[0,1,0,1,1,1,0,0],
                  [1,0,0,1,1,1,0,0],[1,1,1,1,0,0,0,1])$
(%i8) plsquares(mtrx,[a,b,_And,_Or,_Xor,_Nand,_Nor,_Nxor],
                     [_And,_Or,_Xor,_Nand,_Nor,_Nxor],1,0);
      Determination Coefficient for
[_And, _Or, _Xor, _Nand, _Nor, _Nxor] =
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
(%o2) [_And = a b, _Or = - a b + b + a,
_Xor = - 2 a b + b + a, _Nand = 1 - a b,
_Nor = a b - b - a + 1, _Nxor = 2 a b - b - a + 1]
```

To use this function write first `load("lsquares")`.

62 makeOrders

62.1 Functions and Variables for makeOrders

makeOrders (*indvarlist*,*orderlist*) Function

Returns a list of all powers for a polynomial up to and including the arguments.

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

(%i2) makeOrders([a,b],[2,3]);
(%o2) [[0, 0], [0, 1], [0, 2], [0, 3], [1, 0], [1, 1],
            [1, 2], [1, 3], [2, 0], [2, 1], [2, 2], [2, 3]]
(%i3) expand((1+a+a^2)*(1+b+b^2+b^3));
       2  3      3    3    2  2      2    2    2
(%o3) a  b  + a b  + b  + a  b  + a b  + b  + a  b + a b
                                                    2
                                               + b + a  + a + 1
```

where `[0, 1]` is associated with the term b and `[2, 3]` with a^2b^3.

To use this function write first `load("makeOrders")`.

63 mnewton

63.1 Introduction to mnewton

`mnewton` is an implementation of Newton's method for solving nonlinear equations in one or more variables.

63.2 Functions and Variables for mnewton

newtonepsilon Option variable

Default value: `10.0^(-fpprec/2)`

Precision to determine when the `mnewton` function has converged towards the solution.

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]]
```

To use this function write first `load("mnewton")`. See also `newtonepsilon` and `newtonmaxiter`.

64 numericalio

64.1 Introduction to numericalio

`numericalio` is a collection of functions to read and write files and streams. Functions for plain-text input and output can read and write numbers (integer, float, or bigfloat), symbols, and strings. Functions for binary input and output can read and write only floating-point numbers.

If there already exists a list, matrix, or array object to store input data, `numericalio` input functions can write data into that object. Otherwise, `numericalio` can guess, to some degree, the structure of an object to store the data, and return that object.

64.1.1 Plain-text input and output

In plain-text input and output, it is assumed that each item to read or write is an atom: an integer, float, bigfloat, string, or symbol, and not a rational or complex number or any other kind of nonatomic expression. The `numericalio` functions may attempt to do something sensible faced with nonatomic expressions, but the results are not specified here and subject to change.

Atoms in both input and output files have the same format as in Maxima batch files or the interactive console. In particular, strings are enclosed in double quotes, backslash `\` prevents any special interpretation of the next character, and the question mark `?` is recognized at the beginning of a symbol to mean a Lisp symbol (as opposed to a Maxima symbol). No continuation character (to join broken lines) is recognized.

64.1.2 Separator flag values for input

The functions for plain-text input and output take an optional argument, *separator_flag*, that tells what character separates data.

For plain-text input, these values of *separator_flag* are recognized: `comma` for comma separated values, `pipe` for values separated by the vertical bar character `|`, `semicolon` for values separated by semicolon `;`, and `space` for values separated by space or tab characters. If the file name ends in `.csv` and *separator_flag* is not specified, `comma` is assumed. If the file name ends in something other than `.csv` and `separator_flag` is not specified, `space` is assumed.

In plain-text input, multiple successive space and tab characters count as a single separator. However, multiple comma, pipe, or semicolon characters are significant. Successive comma, pipe, or semicolon characters (with or without intervening spaces or tabs) are considered to have `false` between the separators. For example, `1234,,Foo` is treated the same as `1234,false,Foo`.

64.1.3 Separator flag values for output

For plain-text output, `tab`, for values separated by the tab character, is recognized as a value of *separator_flag*, as well as `comma`, `pipe`, `semicolon`, and `space`.

In plain-text output, `false` atoms are written as such; a list `[1234, false, Foo]` is written `1234,false,Foo`, and there is no attempt to collapse the output to `1234,,Foo`.

64.1.4 Binary floating-point input and output

`numericalio` functions can read and write 8-byte IEEE 754 floating-point numbers. These numbers can be stored either least significant byte first or most significant byte first, according to the global flag set by `assume_external_byte_order`. If not specified, `numericalio` assumes the external byte order is most-significant byte first.

Other kinds of numbers are coerced to 8-byte floats; `numericalio` cannot read or write non-numeric data.

Some Lisp implementations do not recognize IEEE 754 special values (positive and negative infinity, not-a-number values, denormalized values). The effect of reading such values with `numericalio` is undefined.

`numericalio` includes functions to open a stream for reading or writing a stream of bytes.

64.2 Functions and Variables for plain-text input and output

read_matrix (*S*) Function
read_matrix (*S*, *M*) Function
read_matrix (*S*, *separator_flag*) Function
read_matrix (*S*, *M*, *separator_flag*) Function

`read_matrix(S)` reads the source *S* and returns its entire content as a matrix. The size of the matrix is inferred from the input data; each line of the file becomes one row of the matrix. If some lines have different lengths, `read_matrix` complains.

`read_matrix(S, M)` read the source *S* into the matrix *M*, until *M* is full or the source is exhausted. Input data are read into the matrix in row-major order; the input need not have the same number of rows and columns as *M*.

The source *S* may be a file name or a stream.

The recognized values of *separator_flag* are `comma`, `pipe`, `semicolon`, and `space`. If *separator_flag* is not specified, the file is assumed space-delimited.

read_array (*S*, *A*) Function
read_array (*S*, *A*, *separator_flag*) Function

Reads the source *S* into the array *A*, until *A* is full or the source is exhausted. Input data are read into the array in row-major order; the input need not conform to the dimensions of *A*.

The source *S* may be a file name or a stream.

The recognized values of *separator_flag* are `comma`, `pipe`, `semicolon`, and `space`. If *separator_flag* is not specified, the file is assumed space-delimited.

read_hashed_array (*S*, *A*) Function
read_hashed_array (*S*, *A*, *separator_flag*) Function

Reads the source *S* and returns its entire content as a hashed array. The source *S* may be a file name or a stream.

`read_hashed_array` treats the first item on each line as a hash key, and associates the remainder of the line (as a list) with the key. For example, the line `567 12 17 32 55` is equivalent to `A[567]: [12, 17, 32, 55]$`. Lines need not have the same numbers of elements.

The recognized values of *separator_flag* are `comma`, `pipe`, `semicolon`, and `space`. If *separator_flag* is not specified, the file is assumed space-delimited.

read_nested_list (*S*) Function
read_nested_list (*S, separator_flag*) Function

Reads the source *S* and returns its entire content as a nested list. The source *S* may be a file name or a stream.

`read_nested_list` returns a list which has a sublist for each line of input. Lines need not have the same numbers of elements. Empty lines are *not* ignored: an empty line yields an empty sublist.

The recognized values of *separator_flag* are `comma`, `pipe`, `semicolon`, and `space`. If *separator_flag* is not specified, the file is assumed space-delimited.

read_list (*S*) Function
read_list (*S, L*) Function
read_list (*S, separator_flag*) Function
read_list (*S, L, separator_flag*) Function

`read_list(`*S*`)` reads the source *S* and returns its entire content as a flat list.

`read_list(`*S*`, `*L*`)` reads the source *S* into the list *L*, until *L* is full or the source is exhausted.

The source *S* may be a file name or a stream.

The recognized values of *separator_flag* are `comma`, `pipe`, `semicolon`, and `space`. If *separator_flag* is not specified, the file is assumed space-delimited.

write_data (*X, D*) Function
write_data (*X, D, separator_flag*) Function

Writes the object *X* to the destination *D*.

`write_data` writes a matrix in row-major order, with one line per row.

`write_data` writes an array created by `array` or `make_array` in row-major order, with a new line at the end of every slab. Higher-dimensional slabs are separated by additional new lines.

`write_data` writes a hashed array with each key followed by its associated list on one line.

`write_data` writes a nested list with each sublist on one line.

`write_data` writes a flat list all on one line.

The destination *D* may be a file name or a stream. When the destination is a file name, the global variable `file_output_append` governs whether the output file is appended or truncated. When the destination is a stream, no special action is taken by `write_data` after all the data are written; in particular, the stream remains open.

The recognized values of *separator_flag* are `comma`, `pipe`, `semicolon`, `space`, and `tab`. If *separator_flag* is not specified, the file is assumed space-delimited.

64.3 Functions and Variables for binary input and output

assume_external_byte_order (*byte_order_flag*) Function

Tells `numericalio` the byte order for reading and writing binary data. Two values of *byte_order_flag* are recognized: `lsb` which indicates least-significant byte first, also called little-endian byte order; and `msb` which indicates most-significant byte first, also called big-endian byte order.

If not specified, `numericalio` assumes the external byte order is most-significant byte first.

openr_binary (*file_name*) Function

Returns an input stream of 8-bit unsigned bytes to read the file named by *file_name*.

openw_binary (*file_name*) Function

Returns an output stream of 8-bit unsigned bytes to write the file named by *file_name*.

opena_binary (*file_name*) Function

Returns an output stream of 8-bit unsigned bytes to append the file named by *file_name*.

read_binary_matrix (*S, M*) Function

Reads binary 8-byte floating point numbers from the source S into the matrix M until M is full, or the source is exhausted. Elements of M are read in row-major order.

The source S may be a file name or a stream.

The byte order in elements of the source is specified by `assume_external_byte_order`.

read_binary_array (*S, A*) Function

Reads binary 8-byte floating point numbers from the source S into the array A until A is full, or the source is exhausted. A must be an array created by `array` or `make_array`. Elements of A are read in row-major order.

The source S may be a file name or a stream.

The byte order in elements of the source is specified by `assume_external_byte_order`.

read_binary_list (*S*) Function
read_binary_list (*S, L*) Function

`read_binary_list(`*S*`)` reads the entire content of the source S as a sequence of binary 8-byte floating point numbers, and returns it as a list. The source S may be a file name or a stream.

`read_binary_list(`*S*`, `*L*`)` reads 8-byte binary floating point numbers from the source S until the list L is full, or the source is exhausted.

The byte order in elements of the source is specified by `assume_external_byte_order`.

write_binary_data (*X*, *D*) Function

Writes the object *X*, comprising binary 8-byte IEEE 754 floating-point numbers, to the destination *D*. Other kinds of numbers are coerced to 8-byte floats. `write_binary_data` cannot write non-numeric data.

The object *X* may be a list, a nested list, a matrix, or an array created by `array` or `make_array`; *X* cannot be an undeclared array or any other type of object. `write_binary_data` writes nested lists, matrices, and arrays in row-major order.

The destination *D* may be a file name or a stream. When the destination is a file name, the global variable `file_output_append` governs whether the output file is appended or truncated. When the destination is a stream, no special action is taken by `write_binary_data` after all the data are written; in particular, the stream remains open.

The byte order in elements of the destination is specified by `assume_external_byte_order`.

65 opsubst

65.1 Functions and Variables for opsubst

opsubst (*f*,*g*,*e*) Function
opsubst (*g*=*f*,*e*) Function
opsubst ([*g1*=*f1*,*g2*=*f2*,..., *gn*=*fn*],*e*) Function

The function `opsubst` is similar to the function `subst`, except that `opsubst` only makes substitutions for the operators in an expression. In general, When *f* is an operator in the expression *e*, substitute *g* for *f* in the expression *e*.

To determine the operator, `opsubst` sets `inflag` to true. This means `opsubst` substitutes for the internal, not the displayed, operator in the expression.

Examples:

```
(%i1) load (opsubst)$

(%i2) opsubst(f,g,g(g(x)));
(%o2)                      f(f(x))
(%i3) opsubst(f,g,g(g));
(%o3)                       f(g)
(%i4) opsubst(f,g[x],g[x](z));
(%o4)                       f(z)
(%i5) opsubst(g[x],f, f(z));
(%o5)                      g (z)
                            x
(%i6) opsubst(tan, sin, sin(sin));
(%o6)                     tan(sin)
(%i7) opsubst([f=g,g=h],f(x));
(%o7)                       h(x)
```

Internally, Maxima does not use the unary negation, division, or the subtraction operators; thus:

```
(%i8) opsubst("+","-",a-b);
(%o8)                      a - b
(%i9) opsubst("f","-",-a);
(%o9)                       - a
(%i10) opsubst("^^","/",a/b);
                             a
(%o10)                       -
                             b
```

The internal representation of -a*b is *(-1,a,b); thus

```
(%i11) opsubst("[","*", -a*b);
(%o11)                   [- 1, a, b]
```

When either operator isn't a Maxima symbol, generally some other function will signal an error:

```
(%i12) opsubst(a+b,f, f(x));

Improper name or value in functional position:
b + a
 -- an error.  Quitting.  To debug this try debugmode(true);
```

However, subscripted operators are allowed:

```
(%i13) opsubst(g[5],f, f(x));
(%o13)                       g (x)
                              5
```

To use this function write first `load("opsubst")`.

66 orthopoly

66.1 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).

66.1.1 Getting Started with orthopoly

`load (orthopoly)` loads the `orthopoly` package.

To find the third-order Legendre polynomial,

```
(%i1) legendre_p (3, x);
                        3             2
             5 (1 - x)    15 (1 - x)
(%o1)      - ---------- + ---------- - 6 (1 - x) + 1
                  2             2
```

To express this as a sum of powers of x, apply *ratsimp* or *rat* to the result.

```
(%i2) [ratsimp (%), rat (%)];
                          3           3
                       5 x  - 3 x  5 x  - 3 x
(%o2)/R/              [----------, ----------]
                           2           2
```

Alternatively, make the second argument to `legendre_p` (its "main" variable) a canonical rational expression (CRE).

```
(%i1) legendre_p (3, rat (x));
                                3
                             5 x  - 3 x
(%o1)/R/                     ----------
                                2
```

For floating point evaluation, `orthopoly` uses a running error analysis to estimate an upper bound for the error. For example,

```
(%i1) jacobi_p (150, 2, 3, 0.2);
(%o1) interval(- 0.062017037936715, 1.533267919277521E-11)
```

Intervals have the form `interval (c, r)`, where c is the center and r is the radius of the interval. Since Maxima does not support arithmetic on intervals, in some situations, such

as graphics, you want to suppress the error and output only the center of the interval. To do this, set the option variable `orthopoly_returns_intervals` to `false`.

```
(%i1) orthopoly_returns_intervals : false;
(%o1)                          false
(%i2) jacobi_p (150, 2, 3, 0.2);
(%o2)                  - 0.062017037936715
```

Refer to the section see [Floating point Evaluation], page 761 for more information.

Most functions in `orthopoly` have a `gradef` property; thus

```
(%i1) diff (hermite (n, x), x);
(%o1)                       2 n H     (x)
                                 n - 1
(%i2) diff (gen_laguerre (n, a, x), x);
                 (a)                (a)
           n L   (x) - (n + a) L     (x) unit_step(n)
              n                  n - 1
(%o2)      ----------------------------------------
                              x
```

The unit step function in the second example prevents an error that would otherwise arise by evaluating with n equal to 0.

```
(%i3) ev (%, n = 0);
(%o3)                           0
```

The `gradef` property only applies to the "main" variable; derivatives with respect other arguments usually result in an error message; for example

```
(%i1) diff (hermite (n, x), x);
(%o1)                       2 n H     (x)
                                 n - 1
(%i2) diff (hermite (n, x), n);

Maxima doesn't know the derivative of hermite with respect the first
argument
 -- an error.  Quitting.  To debug this try debugmode(true);
```

Generally, functions in `orthopoly` map over lists and matrices. For the mapping to fully evaluate, the option variables `doallmxops` and `listarith` must both be `true` (the defaults). To illustrate the mapping over matrices, consider

```
(%i1) hermite (2, x);
                                    2
(%o1)                     - 2 (1 - 2 x )
(%i2) m : matrix ([0, x], [y, 0]);
                             [ 0  x ]
(%o2)                        [      ]
                             [ y  0 ]
(%i3) hermite (2, m);
                [                               2  ]
                [     - 2         - 2 (1 - 2 x ) ]
(%o3)           [                                ]
                [            2                   ]
```

```
                    [ - 2 (1 - 2 y )       - 2       ]
```

In the second example, the `i, j` element of the value is `hermite (2, m[i,j])`; this is not the same as computing `-2 + 4 m . m`, as seen in the next example.

```
(%i4) -2 * matrix ([1, 0], [0, 1]) + 4 * m . m;
                     [ 4 x y - 2      0     ]
(%o4)                [                      ]
                     [     0      4 x y - 2 ]
```

If you evaluate a function at a point outside its domain, generally `orthopoly` returns the function unevaluated. For example,

```
(%i1) legendre_p (2/3, x);
(%o1)                           P   (x)
                                 2/3
```

`orthopoly` supports translation into TeX; it also does two-dimensional output on a terminal.

```
(%i1) spherical_harmonic (l, m, theta, phi);
                              m
(%o1)                        Y (theta, phi)
                              l
(%i2) tex (%);
$$Y_{l}^{m}\left(\vartheta,\varphi\right)$$
(%o2)                           false
(%i3) jacobi_p (n, a, a - b, x/2);
                              (a, a - b) x
(%o3)                        P           (-)
                              n           2
(%i4) tex (%);
$$P_{n}^{\left(a,a-b\right)}\left({{x}\over{2}}\right)$$
(%o4)                           false
```

66.1.2 Limitations

When an expression involves several orthogonal polynomials with symbolic orders, it's possible that the expression actually vanishes, yet Maxima is unable to simplify it to zero. If you divide by such a quantity, you'll be in trouble. For example, the following expression vanishes for integers n greater than 1, yet Maxima is unable to simplify it to zero.

```
(%i1) (2*n - 1) * legendre_p (n - 1, x) * x - n * legendre_p (n, x)
      + (1 - n) * legendre_p (n - 2, x);
(%o1)  (2 n - 1) P     (x) x - n P (x) + (1 - n) P     (x)
                  n - 1           n               n - 2
```

For a specific n, we can reduce the expression to zero.

```
(%i2) ev (% ,n = 10, ratsimp);
(%o2)                             0
```

Generally, the polynomial form of an orthogonal polynomial is ill-suited for floating point evaluation. Here's an example.

```
(%i1) p : jacobi_p (100, 2, 3, x)$
```

```
(%i2) subst (0.2, x, p);
(%o2)                 3.4442767023833592E+35
(%i3) jacobi_p (100, 2, 3, 0.2);
(%o3)  interval(0.18413609135169, 6.8990300925815987E-12)
(%i4) float(jacobi_p (100, 2, 3, 2/10));
(%o4)                    0.18413609135169
```

The true value is about 0.184; this calculation suffers from extreme subtractive cancellation error. Expanding the polynomial and then evaluating, gives a better result.

```
(%i5) p : expand(p)$
(%i6) subst (0.2, x, p);
(%o6) 0.18413609766122982
```

This isn't a general rule; expanding the polynomial does not always result in an expression that is better suited for numerical evaluation. By far, the best way to do numerical evaluation is to make one or more of the function arguments floating point numbers. By doing that, specialized floating point algorithms are used for evaluation.

Maxima's `float` function is somewhat indiscriminate; if you apply `float` to an expression involving an orthogonal polynomial with a symbolic degree or order parameter, these parameters may be converted into floats; after that, the expression will not evaluate fully. Consider

```
(%i1) assoc_legendre_p (n, 1, x);
                                1
(%o1)                          P (x)
                                n
(%i2) float (%);
                               1.0
(%o2)                         P   (x)
                               n
(%i3) ev (%, n=2, x=0.9);
                              1.0
(%o3)                        P   (0.9)
                              2
```

The expression in (%o3) will not evaluate to a float; `orthopoly` doesn't recognize floating point values where it requires an integer. Similarly, numerical evaluation of the `pochhammer` function for orders that exceed `pochhammer_max_index` can be troublesome; consider

```
(%i1) x :  pochhammer (1, 10), pochhammer_max_index : 5;
(%o1)                          (1)
                                  10
```

Applying `float` doesn't evaluate x to a float

```
(%i2) float (x);
(%o2)                      (1.0)
                                10.0
```

To evaluate x to a float, you'll need to bind `pochhammer_max_index` to 11 or greater and apply `float` to x.

```
(%i3) float (x), pochhammer_max_index : 11;
(%o3)                      3628800.0
```

The default value of `pochhammer_max_index` is 100; change its value after loading `orthopoly`.

Finally, be aware that reference books vary on the definitions of the orthogonal polynomials; we've generally used the conventions of conventions of Abramowitz and Stegun.

Before you suspect a bug in orthopoly, check some special cases to determine if your definitions match those used by `orthopoly`. Definitions often differ by a normalization; occasionally, authors use "shifted" versions of the functions that makes the family orthogonal on an interval other than $(-1, 1)$. To define, for example, a Legendre polynomial that is orthogonal on $(0, 1)$, define

```
(%i1) shifted_legendre_p (n, x) := legendre_p (n, 2*x - 1)$

(%i2) shifted_legendre_p (2, rat (x));
                              2
(%o2)/R/                   6 x  - 6 x + 1
(%i3) legendre_p (2, rat (x));
                                2
                              3 x  - 1
(%o3)/R/                      --------
                                 2
```

66.1.3 Floating point Evaluation

Most functions in `orthopoly` use a running error analysis to estimate the error in floating point evaluation; the exceptions are the spherical Bessel functions and the associated Legendre polynomials of the second kind. For numerical evaluation, the spherical Bessel functions call SLATEC functions. No specialized method is used for numerical evaluation of the associated Legendre polynomials of the second kind.

The running error analysis ignores errors that are second or higher order in the machine epsilon (also known as unit roundoff). It also ignores a few other errors. It's possible (although unlikely) that the actual error exceeds the estimate.

Intervals have the form `interval (`c`, `r`)`, where c is the center of the interval and r is its radius. The center of an interval can be a complex number, and the radius is always a positive real number.

Here is an example.

```
(%i1) fpprec : 50$

(%i2) y0 : jacobi_p (100, 2, 3, 0.2);
(%o2) interval(0.1841360913516871, 6.8990300925815987E-12)
(%i3) y1 : bfloat (jacobi_p (100, 2, 3, 1/5));
(%o3) 1.8413609135168563091370224958913493690868904463668b-1
```

Let's test that the actual error is smaller than the error estimate

```
(%i4) is (abs (part (y0, 1) - y1) < part (y0, 2));
(%o4)                         true
```

Indeed, for this example the error estimate is an upper bound for the true error.

Maxima does not support arithmetic on intervals.

```
(%i1) legendre_p (7, 0.1) + legendre_p (8, 0.1);
(%o1) interval(0.18032072148437508, 3.1477135311021797E-15)
        + interval(- 0.19949294375000004, 3.3769353084291579E-15)
```

A user could define arithmetic operators that do interval math. To define interval addition, we can define

```
(%i1) infix ("@+")$

(%i2) "@+"(x,y) := interval (part (x, 1) + part (y, 1), part (x, 2)
      + part (y, 2))$

(%i3) legendre_p (7, 0.1) @+ legendre_p (8, 0.1);
(%o3) interval(- 0.019172222265624955, 6.5246488395313372E-15)
```

The special floating point routines get called when the arguments are complex. For example,

```
(%i1) legendre_p (10, 2 + 3.0*%i);
(%o1) interval(- 3.876378825E+7 %i - 6.0787748E+7,
                                          1.2089173052721777E-6)
```

Let's compare this to the true value.

```
(%i1) float (expand (legendre_p (10, 2 + 3*%i)));
(%o1)          - 3.876378825E+7 %i - 6.0787748E+7
```

Additionally, when the arguments are big floats, the special floating point routines get called; however, the big floats are converted into double floats and the final result is a double.

```
(%i1) ultraspherical (150, 0.5b0, 0.9b0);
(%o1) interval(- 0.043009481257265, 3.3750051301228864E-14)
```

66.1.4 Graphics and `orthopoly`

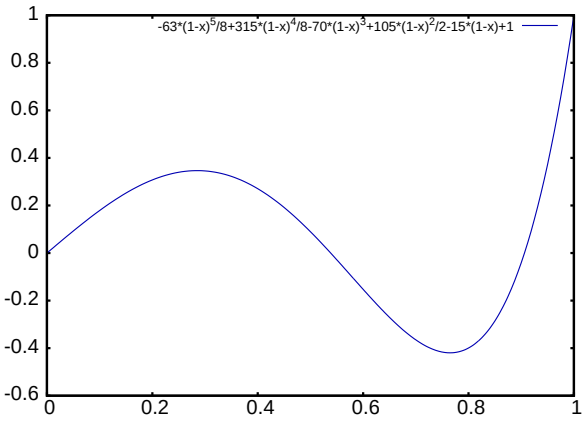
To plot expressions that involve the orthogonal polynomials, you must do two things:

1. Set the option variable `orthopoly_returns_intervals` to `false`,
2. Quote any calls to `orthopoly` functions.

If function calls aren't quoted, Maxima evaluates them to polynomials before plotting; consequently, the specialized floating point code doesn't get called. Here is an example of how to plot an expression that involves a Legendre polynomial.

```
(%i1) plot2d ('(legendre_p (5, x)), [x, 0, 1]),
                          orthopoly_returns_intervals : false;
```

```
(%o1)
```



The *entire* expression `legendre_p (5, x)` is quoted; this is different than just quoting the function name using `'legendre_p (5, x)`.

66.1.5 Miscellaneous Functions

The `orthopoly` package defines the Pochhammer symbol and a unit step function. `orthopoly` uses the Kronecker delta function and the unit step function in `gradef` statements.

To convert Pochhammer symbols into quotients of gamma functions, use `makegamma`.

```
(%i1) makegamma (pochhammer (x, n));
                          gamma(x + n)
(%o1)                     ------------
                            gamma(x)
(%i2) makegamma (pochhammer (1/2, 1/2));
                                1
(%o2)                           ---------
                                sqrt(%pi)
```

Derivatives of the Pochhammer symbol are given in terms of the `psi` function.

```
(%i1) diff (pochhammer (x, n), x);
(%o1)             (x)  (psi (x + n) - psi (x))
                     n     0             0
(%i2) diff (pochhammer (x, n), n);
(%o2)                  (x)  psi (x + n)
                          n    0
```

You need to be careful with the expression in (%o1); the difference of the `psi` functions has polynomials when x = `-1, -2, .., -`n. These polynomials cancel with factors in `pochhammer (`x`, `n`)` making the derivative a degree n `- 1` polynomial when n is a positive integer.

The Pochhammer symbol is defined for negative orders through its representation as a quotient of gamma functions. Consider

```
(%i1) q : makegamma (pochhammer (x, n));
                            gamma(x + n)
(%o1)                       ------------
                              gamma(x)
(%i2) sublis ([x=11/3, n= -6], q);
                                 729
(%o2)                          - ----
                                 2240
```

Alternatively, we can get this result directly.

```
(%i1) pochhammer (11/3, -6);
                                 729
(%o1)                          - ----
                                 2240
```

The unit step function is left-continuous; thus

```
(%i1) [unit_step (-1/10), unit_step (0), unit_step (1/10)];
(%o1)                          [0, 0, 1]
```

If you need a unit step function that is neither left or right continuous at zero, define your own using `signum`; for example,

```
(%i1) xunit_step (x) := (1 + signum (x))/2$

(%i2) [xunit_step (-1/10), xunit_step (0), xunit_step (1/10)];
                                  1
(%o2)                          [0, -, 1]
                                  2
```

Do not redefine `unit_step` itself; some code in `orthopoly` requires that the unit step function be left-continuous.

66.1.6 Algorithms

Generally, `orthopoly` does symbolic evaluation by using a hypergeometic representation of the orthogonal polynomials. The hypergeometic functions are evaluated using the (undocumented) functions `hypergeo11` and `hypergeo21`. The exceptions are the half-integer Bessel functions and the associated Legendre function of the second kind. The half-integer Bessel functions are evaluated using an explicit representation, and the associated Legendre function of the second kind is evaluated using recursion.

For floating point evaluation, we again convert most functions into a hypergeometic form; we evaluate the hypergeometic functions using forward recursion. Again, the exceptions are the half-integer Bessel functions and the associated Legendre function of the second kind. Numerically, the half-integer Bessel functions are evaluated using the SLATEC code.

66.2 Functions and Variables for orthogonal polynomials

assoc_legendre_p (*n, m, x*) Function

The associated Legendre function of the first kind of degree *n* and order *m*.

Reference: Abramowitz and Stegun, equations 22.5.37, page 779, 8.6.6 (second equation), page 334, and 8.2.5, page 333.

assoc_legendre_q (*n*, *m*, *x*) Function

The associated Legendre function of the second kind of degree *n* and order *m*.

Reference: Abramowitz and Stegun, equation 8.5.3 and 8.1.8.

chebyshev_t (*n*, *x*) Function

The Chebyshev function of the first kind.

Reference: Abramowitz and Stegun, equation 22.5.47, page 779.

chebyshev_u (*n*, *x*) Function

The Chebyshev function of the second kind.

Reference: Abramowitz and Stegun, equation 22.5.48, page 779.

gen_laguerre (*n*, *a*, *x*) Function

The generalized Laguerre polynomial of degree *n*.

Reference: Abramowitz and Stegun, equation 22.5.54, page 780.

hermite (*n*, *x*) Function

The Hermite polynomial.

Reference: Abramowitz and Stegun, equation 22.5.55, page 780.

intervalp (*e*) Function

Return `true` if the input is an interval and return false if it isn't.

jacobi_p (*n*, *a*, *b*, *x*) Function

The Jacobi polynomial.

The Jacobi polynomials are actually defined for all *a* and *b*; however, the Jacobi polynomial weight `(1 - x)^a (1 + x)^b` isn't integrable for *a* `<= -1` or *b* `<= -1`.

Reference: Abramowitz and Stegun, equation 22.5.42, page 779.

laguerre (*n*, *x*) Function

The Laguerre polynomial.

Reference: Abramowitz and Stegun, equations 22.5.16 and 22.5.54, page 780.

legendre_p (*n*, *x*) Function

The Legendre polynomial of the first kind.

Reference: Abramowitz and Stegun, equations 22.5.50 and 22.5.51, page 779.

legendre_q (*n*, *x*) Function

The Legendre polynomial of the first kind.

Reference: Abramowitz and Stegun, equations 8.5.3 and 8.1.8.

orthopoly_recur (*f, args*) Function

Returns a recursion relation for the orthogonal function family *f* with arguments *args*. The recursion is with respect to the polynomial degree.

```
(%i1) orthopoly_recur (legendre_p, [n, x]);
                (2 n - 1) P     (x) x + (1 - n) P     (x)
                          n - 1                 n - 2
(%o1)   P (x) = ----------------------------------------
         n                          n
```

The second argument to `orthopoly_recur` must be a list with the correct number of arguments for the function *f*; if it isn't, Maxima signals an error.

```
(%i1) orthopoly_recur (jacobi_p, [n, x]);

Function jacobi_p needs 4 arguments, instead it received 2
 -- an error.  Quitting.  To debug this try debugmode(true);
```

Additionally, when *f* isn't the name of one of the families of orthogonal polynomials, an error is signalled.

```
(%i1) orthopoly_recur (foo, [n, x]);

A recursion relation for foo isn't known to Maxima
 -- an error.  Quitting.  To debug this try debugmode(true);
```

orthopoly_returns_intervals Variable

Default value: `true`

When `orthopoly_returns_intervals` is `true`, floating point results are returned in the form `interval (`*c*`,` *r*`)`, where *c* is the center of an interval and *r* is its radius. The center can be a complex number; in that case, the interval is a disk in the complex plane.

orthopoly_weight (*f, args*) Function

Returns a three element list; the first element is the formula of the weight for the orthogonal polynomial family *f* with arguments given by the list *args*; the second and third elements give the lower and upper endpoints of the interval of orthogonality. For example,

```
(%i1) w : orthopoly_weight (hermite, [n, x]);
                              2
                           - x
(%o1)                  [%e    , - inf, inf]
(%i2) integrate(w[1]*hermite(3, x)*hermite(2, x), x, w[2], w[3]);
(%o2)                          0
```

The main variable of *f* must be a symbol; if it isn't, Maxima signals an error.

pochhammer (*n, x*) Function

The Pochhammer symbol. For nonnegative integers *n* with *n* `<= pochhammer_max_index`, the expression `pochhammer (`*x*`,` *n*`)` evaluates to the product *x* `(`*x* `+ 1) (`*x* `+ 2) ... (`*x* `+ n - 1)` when *n* `> 0` and to 1 when *n* `= 0`. For negative *n*, `pochhammer (`*x*`,` *n*`)` is defined as `(-1)^`*n* `/ pochhammer (1 -` *x*`, -`*n*`)`. Thus

```
(%i1) pochhammer (x, 3);
(%o1)                    x (x + 1) (x + 2)
(%i2) pochhammer (x, -3);
                                  1
(%o2)                - ------------------------
                       (1 - x) (2 - x) (3 - x)
```

To convert a Pochhammer symbol into a quotient of gamma functions, (see Abramowitz and Stegun, equation 6.1.22) use `makegamma`; for example

```
(%i1) makegamma (pochhammer (x, n));
                             gamma(x + n)
(%o1)                        ------------
                               gamma(x)
```

When n exceeds `pochhammer_max_index` or when n is symbolic, `pochhammer` returns a noun form.

```
(%i1) pochhammer (x, n);
(%o1)                           (x)
                                   n
```

pochhammer_max_index Variable

Default value: 100

`pochhammer (`n`, `x`)` expands to a product if and only if n `<= pochhammer_max_index`.

Examples:

```
(%i1) pochhammer (x, 3), pochhammer_max_index : 3;
(%o1)                    x (x + 1) (x + 2)
(%i2) pochhammer (x, 4), pochhammer_max_index : 3;
(%o2)                           (x)
                                   4
```

Reference: Abramowitz and Stegun, equation 6.1.16, page 256.

spherical_bessel_j (n, x) Function

The spherical Bessel function of the first kind.

Reference: Abramowitz and Stegun, equations 10.1.8, page 437 and 10.1.15, page 439.

spherical_bessel_y (n, x) Function

The spherical Bessel function of the second kind.

Reference: Abramowitz and Stegun, equations 10.1.9, page 437 and 10.1.15, page 439.

spherical_hankel1 (n, x) Function

The spherical Hankel function of the first kind.

Reference: Abramowitz and Stegun, equation 10.1.36, page 439.

spherical_hankel2 (n, x) Function

The spherical Hankel function of the second kind.

Reference: Abramowitz and Stegun, equation 10.1.17, page 439.

spherical_harmonic (*n*, *m*, *x*, *y*) Function

The spherical harmonic function.

Reference: Merzbacher 9.64.

unit_step (*x*) Function

The left-continuous unit step function; thus `unit_step (x)` vanishes for *x* `<= 0` and equals 1 for *x* `> 0`.

If you want a unit step function that takes on the value 1/2 at zero, use `(1 + signum (x))/2`.

ultraspherical (*n*, *a*, *x*) Function

The ultraspherical polynomial (also known as the Gegenbauer polynomial).

Reference: Abramowitz and Stegun, equation 22.5.46, page 779.

67 plotdf

67.1 Introduction to plotdf

The function `plotdf` creates a plot of the direction field of a first-order Ordinary Differential Equation (ODE) or a system of two autonomous first-order ODE's.

Since this is an additional package, in order to use it you must first load it with `load("plotdf")`. Plotdf requires Openmath, which is provided by the package Xmaxima (Xmaxima is not only used as a graphical console for Maxima but also to plot graphs in the Openmath format).

To plot the direction field of a single ODE, the ODE must be written in the form:

$$\frac{dy}{dx} = F(x, y)$$

and the function F should be given as the argument for `plotdf`. If the independent and dependent variables are not x, and y, as in the equation above, then those two variables should be named explicitly in a list given as an argument to the plotdf command (see the examples).

To plot the direction field of a set of two autonomous ODE's, they must be written in the form

$$\frac{dx}{dt} = G(x, y) \qquad \frac{dy}{dt} = F(x, y)$$

and the argument for `plotdf` should be a list with the two functions G and F, in that order; namely, the first expression in the list will be taken to be the time derivative of the variable represented on the horizontal axis, and the second expression will be the time derivative of the variable represented on the vertical axis. Those two variables do not have to be x and y, but if they are not, then the second argument given to plotdf must be another list naming the two variables, first the one on the horizontal axis and then the one on the vertical axis.

If only one ODE is given, `plotdf` will implicitly admit `x=t`, and `G(x,y)=1`, transforming the non-autonomous equation into a system of two autonomous equations.

67.2 Functions and Variables for plotdf

plotdf (*dydx, ...options...*) Function
plotdf (*dvdu, [u,v], ...options...*) Function
plotdf (*[dxdt,dydt], ...options...*) Function
plotdf (*[dudt,dvdt], [u,v], ...options...*) Function

Displays a direction field in two dimensions x and y.

dydx, *dxdt* and *dydt* are expressions that depend on x and y. *dvdu*, *dudt* and *dvdt* are expressions that depend on u and v. In addition to those two variables, the expressions can also depend on a set of parameters, with numerical values given with

the `parameters` option (the option syntax is given below), or with a range of allowed values specified by a *sliders* option.

Several other options can be given within the command, or selected in the menu. Integral curves can be obtained by clicking on the plot, or with the option `trajectory_at`. The direction of the integration can be controlled with the `direction` option, which can have values of *forward*, *backward* or *both*. The number of integration steps is given by `nsteps` and the time interval between them is set up with the `tstep` option. The Adams Moulton method is used for the integration; it is also possible to switch to an adaptive Runge-Kutta 4th order method.

Plot window menu:

The menu in the plot window has the following options: *Zoom*, will change the behavior of the mouse so that it will allow you to zoom in on a region of the plot by clicking with the left button. Each click near a point magnifies the plot, keeping the center at the point where you clicked. Holding the ⟨Shift⟩ key while clicking, zooms out to the previous magnification. To resume computing trajectories when you click on a point, select *Integrate* from the menu.

The option *Config* in the menu can be used to change the ODE(s) in use and various other settings. After configuration changes are made, the menu option *Replot* should be selected, to activate the new settings. If a pair of coordinates are entered in the field *Trajectory at* in the *Config* dialog menu, and the ⟨enter⟩ key is pressed, a new integral curve will be shown, in addition to the ones already shown. When *Replot* is selected, only the last integral curve entered will be shown.

Holding the right mouse button down while the cursor is moved, can be used to drag the plot sideways or up and down. Additional parameters such as the number of steps, the initial value of t and the x and y centers and radii, may be set in the Config menu.

A copy of the plot can be saved as a postscript file, using the menu option *Save*.

Plot options:

The `plotdf` command may include several commands, each command is a list of two or more items. The first item is the name of the option, and the remainder comprises the value or values assigned to the option.

The options which are recognized by `plotdf` are the following:

- *tstep* defines the length of the increments on the independent variable t, used to compute an integral curve. If only one expression *dydx* is given to `plotdf`, the x variable will be directly proportional to t. The default value is 0.1.
- *nsteps* defines the number of steps of length `tstep` that will be used for the independent variable, to compute an integral curve. The default value is 100.
- *direction* defines the direction of the independent variable that will be followed to compute an integral curve. Possible values are `forward`, to make the independent variable increase `nsteps` times, with increments `tstep`, `backward`, to make the independent variable decrease, or `both` that will lead to an integral curve that extends `nsteps` forward, and `nsteps` backward. The keywords `right` and `left` can be used as synonyms for `forward` and `backward`. The default value is `both`.

- *tinitial* defines the initial value of variable t used to compute integral curves. Since the differential equations are autonomous, that setting will only appear in the plot of the curves as functions of t. The default value is 0.

- *versus_t* is used to create a second plot window, with a plot of an integral curve, as two functions x, y, of the independent variable t. If `versus_t` is given any value different from 0, the second plot window will be displayed. The second plot window includes another menu, similar to the menu of the main plot window. The default value is 0.

- *trajectory_at* defines the coordinates *xinitial* and *yinitial* for the starting point of an integral curve. The option is empty by default.

- *parameters* defines a list of parameters, and their numerical values, used in the definition of the differential equations. The name and values of the parameters must be given in a string with a comma-separated sequence of pairs `name=value`.

- *sliders* defines a list of parameters that will be changed interactively using slider buttons, and the range of variation of those parameters. The names and ranges of the parameters must be given in a string with a comma-separated sequence of elements `name=min:max`

- *xfun* defines a string with semi-colon-separated sequence of functions of x to be displayed, on top of the direction field. Those functions will be parsed by Tcl and not by Maxima.

- x should be followed by two numbers, which will set up the minimum and maximum values shown on the horizontal axis. If the variable on the horizontal axis is not x, then this option should have the name of the variable on the horizontal axis. The default horizontal range is from -10 to 10.

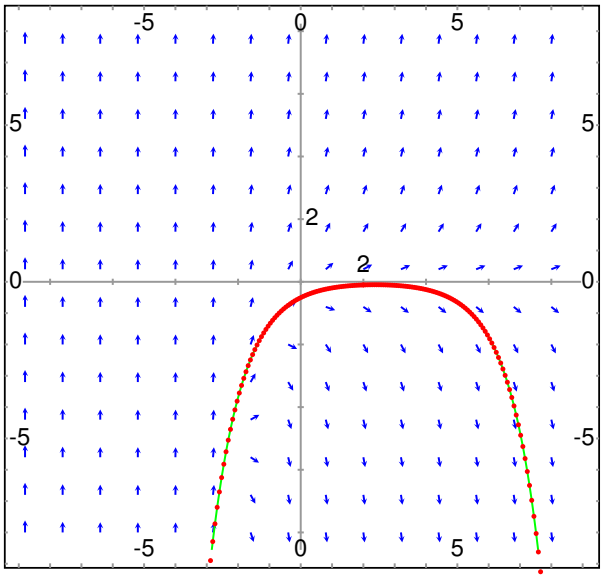
- y should be followed by two numbers, which will set up the minimum and maximum values shown on the vertical axis. If the variable on the vertical axis is not y, then this option should have the name of the variable on the vertical axis. The default vertical range is from -10 to 10.

Examples:

- To show the direction field of the differential equation $y' = exp(-x) + y$ and the solution that goes through $(2, -0.1)$:

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

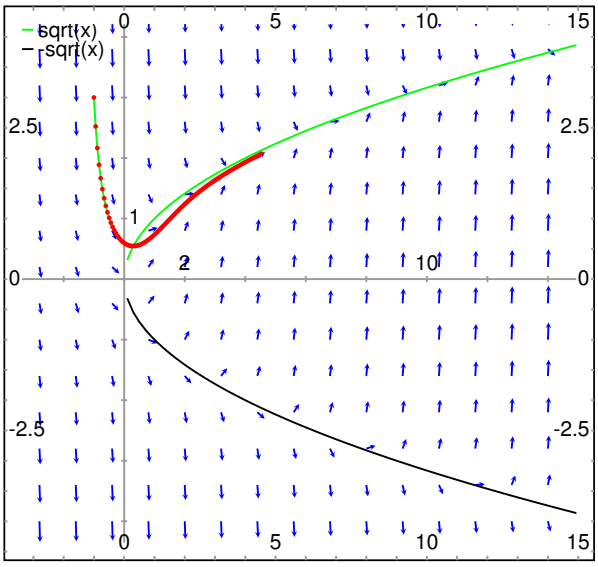
```
(%i2) plotdf(exp(-x)+y,[trajectory_at,2,-0.1])$
```



- To obtain the direction field for the equation $diff(y,x) = x - y^2$ and the solution with initial condition $y(-1) = 3$, we can use the command:

```
(%i3) plotdf(x-y^2,[xfun,"sqrt(x);-sqrt(x)"],
          [trajectory_at,-1,3], [direction,forward],
          [y,-5,5], [x,-4,16])$
```

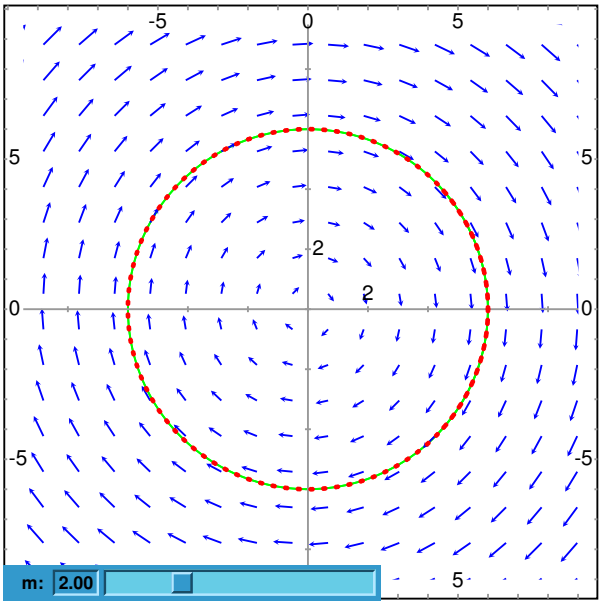
  The graph also shows the function $y = sqrt(x)$.



- The following example shows the direction field of a harmonic oscillator, defined by the two equations $dz/dt = v$ and $dv/dt = -k * z/m$, and the integral curve

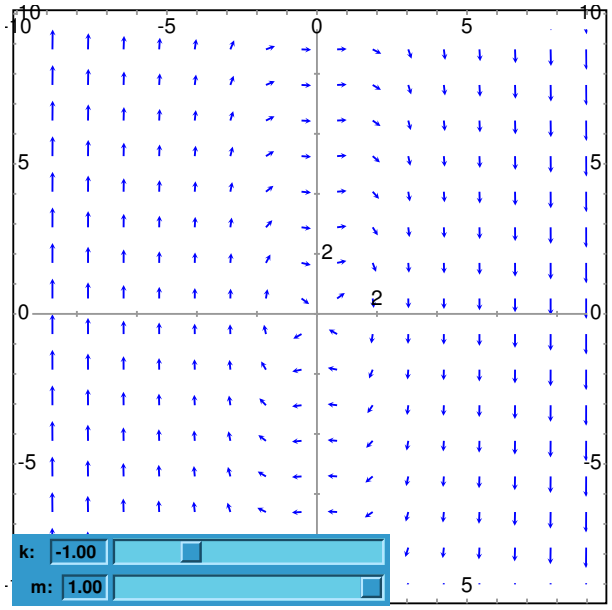
through $(z, v) = (6, 0)$, with a slider that will allow you to change the value of m interactively (k is fixed at 2):

```
(%i4) plotdf([v,-k*z/m], [z,v], [parameters,"m=2,k=2"],
              [sliders,"m=1:5"], [trajectory_at,6,0])$
```



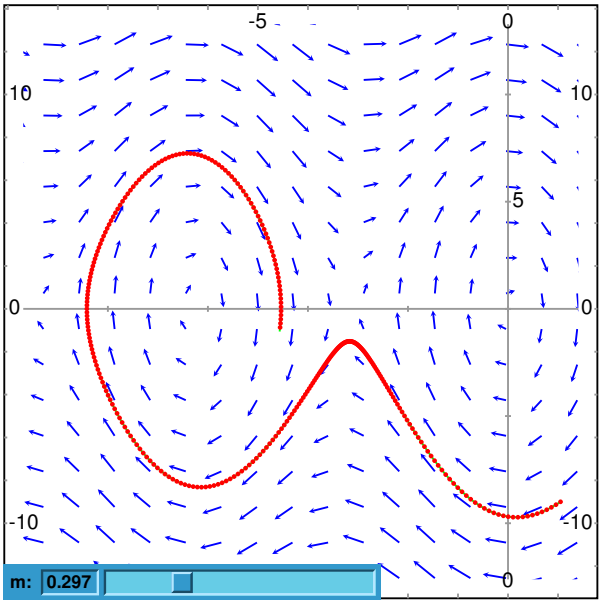
- To plot the direction field of the Duffing equation, $m*x''+c*x'+k*x+b*x^3 = 0$, we introduce the variable $y = x'$ and use:

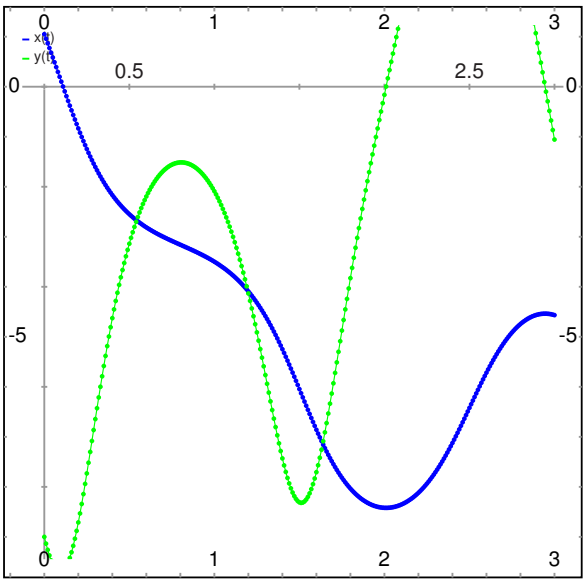
```
(%i5) plotdf([y,-(k*x + c*y + b*x^3)/m],
              [parameters,"k=-1,m=1.0,c=0,b=1"],
              [sliders,"k=-2:2,m=-1:1"],[tstep,0.1])$
```



- The direction field for a damped pendulum, including the solution for the given initial conditions, with a slider that can be used to change the value of the mass m, and with a plot of the two state variables as a function of time:

```
(%i6) 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])$
```





68 romberg

68.1 Functions and Variables for romberg

romberg (*expr*, *x*, *a*, *b*) Function
romberg (*F*, *a*, *b*) Function

Computes a numerical integration by Romberg's method.

`romberg`(*expr*, *x*, *a*, *b*) returns an estimate of the integral `integrate`(*expr*, *x*, *a*, *b*). *expr* must be an expression which evaluates to a floating point value when *x* is bound to a floating point value.

`romberg`(*F*, *a*, *b*) returns an estimate of the integral `integrate(F(x), x,` *a*, *b*) where `x` represents the unnamed, sole argument of *F*; the actual argument is not named `x`. *F* must be a Maxima or Lisp function which returns a floating point value when the argument is a floating point value. *F* may name a translated or compiled Maxima function.

The accuracy of `romberg` is governed by the global variables `rombergabs` and `rombergtol`. `romberg` terminates successfully when the absolute difference between successive approximations is less than `rombergabs`, or the relative difference in successive approximations is less than `rombergtol`. Thus when `rombergabs` is 0.0 (the default) only the relative error test has any effect on `romberg`.

`romberg` halves the stepsize at most `rombergit` times before it gives up; the maximum number of function evaluations is therefore `2^rombergit`. If the error criterion established by `rombergabs` and `rombergtol` is not satisfied, `romberg` prints an error message. `romberg` always makes at least `rombergmin` iterations; this is a heuristic intended to prevent spurious termination when the integrand is oscillatory.

`romberg` repeatedly evaluates the integrand after binding the variable of integration to a specific value (and not before). This evaluation policy makes it possible to nest calls to `romberg`, to compute multidimensional integrals. However, the error calculations do not take the errors of nested integrations into account, so errors may be underestimated. Also, methods devised especially for multidimensional problems may yield the same accuracy with fewer function evaluations.

`load(romberg)` loads this function.

See also `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)  + ---
```

```
                          100              1000              200
(%i3) rombergtol : 1e-6;
(%o3)                 9.9999999999999995E-7
(%i4) rombergit : 15;
(%o4)                           15
(%i5) estimate : romberg (f(x), x, -5, 5);
(%o5)                  173.6730736617464
(%i6) exact : integrate (f(x), x, -5, 5);
(%o6) 10 sqrt(10) atan(70 sqrt(10))
 + 10 sqrt(10) atan(30 sqrt(10)) + 10 sqrt(2) atan(80 sqrt(2))
 + 10 sqrt(2) atan(20 sqrt(2)) + 10 atan(60) + 10 atan(40)
(%i7) abs (estimate - exact) / exact, numer;
(%o7)                7.5527060865060088E-11
```

A 2-dimensional integration, implemented by nested calls to `romberg`.

```
(%i1) load (romberg);
(%o1)    /usr/share/maxima/5.11.0/share/numeric/romberg.lisp
(%i2) g(x, y) := x*y / (x + y);
                                    x y
(%o2)                  g(x, y) := -----
                                   x + y
(%i3) rombergtol : 1e-6;
(%o3)                 9.9999999999999995E-7
(%i4) estimate : romberg (romberg (g(x, y), y, 0, x/2), x, 1, 3);
(%o4)                  0.81930239628356
(%i5) assume (x > 0);
(%o5)                         [x > 0]
(%i6) integrate (integrate (g(x, y), y, 0, x/2), x, 1, 3);
                                      3
                                   2 log(-) - 1
                    9                  2        9
(%o6)       - 9 log(-) + 9 log(3) + ------------ + -
                    2                  6         2
(%i7) exact : radcan (%);
                    26 log(3) - 26 log(2) - 13
(%o7)             - --------------------------
                                 3
(%i8) abs (estimate - exact) / exact, numer;
(%o8)                1.3711979871851024E-10
```

rombergabs Option variable

Default value: 0.0

The accuracy of `romberg` is governed by the global variables `rombergabs` and `rombergtol`. `romberg` terminates successfully when the absolute difference between successive approximations is less than `rombergabs`, or the relative difference in successive approximations is less than `rombergtol`. Thus when `rombergabs` is 0.0 (the default) only the relative error test has any effect on `romberg`.

See also `rombergit` and `rombergmin`.

rombergit Option variable

Default value: 11

`romberg` halves the stepsize at most `rombergit` times before it gives up; the maximum number of function evaluations is therefore `2^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`.

69 simplex

69.1 Introduction to simplex

`simplex` is a package for linear optimization using the simplex algorithm.

Example:

```
(%i1) load("simplex")$
(%i2) minimize_lp(x+y, [3*x+2*y>2, x+4*y>3]);
                   9        7       1
(%o2)             [--, [y = --, x = -]]
                   10       10      5
```

69.2 Functions and Variables for simplex

epsilon_lp Option variable

Default value: `10^-8`

Epsilon used for numerical computations in `linear_program`.

See also: `linear_program`.

linear_program (*A*, *b*, *c*) Function

`linear_program` is an implementation of the simplex algorithm. `linear_program(A, b, c)` computes a vector *x* for which `c.x` is minimum possible among vectors for which `A.x = b` and `x >= 0`. Argument *A* is a matrix and arguments *b* and *c* are lists.

`linear_program` returns a list which contains the minimizing vector *x* and the minimum value `c.x`. If the problem is not bounded, it returns `"Problem not bounded!"` and if the problem is not feasible, it returns `"Problem not feasible!"`.

To use this function first load the `simplex` package with `load(simplex);`.

Example:

```
(%i2) A: matrix([1,1,-1,0], [2,-3,0,-1], [4,-5,0,0])$
(%i3) b: [1,1,6]$
(%i4) c: [1,-2,0,0]$
(%i5) linear_program(A, b, c);
                     13     19        3
(%o5)             [[--, 4, --, 0], - -]
                     2      2         2
```

See also: `minimize_lp`, `scale_lp`, and `epsilon_lp`.

maximize_lp (*obj*, *cond*, [*pos*]) Function

Maximizes linear objective function *obj* subject to some linear constraints *cond*. See `minimize_lp` for detailed description of arguments and return value.

See also: `minimize_lp`.

minimize_lp (*obj, cond,* [*pos*]) Function

Minimizes a linear objective function *obj* subject to some linear constraints *cond*. *cond* a list of linear equations or inequalities. In strict inequalities `>` is replaced by `>=` and `<` by `<=`. The optional argument *pos* is a list of decision variables which are assumed to be positive.

If the minimum exists, `minimize_lp` returns a list which contains the minimum value of the objective function and a list of decision variable values for which the minimum is attained. If the problem is not bounded, `minimize_lp` returns "Problem not bounded!" and if the problem is not feasible, it returns "Ploblem not feasible!".

The decision variables are not assumed to be nonegative by default. If all decision variables are nonegative, set `nonegative_lp` to `true`. If only some of decision variables are positive, list them in the optional argument *pos* (note that this is more efficient than adding constraints).

`minimize_lp` uses the simplex algorithm which is implemented in maxima `linear_program` function.

To use this function first load the `simplex` package with `load(simplex);`.

Examples:

```
(%i1) minimize_lp(x+y, [3*x+y=0, x+2*y>2]);
                       4       6        2
(%o1)                 [-, [y = -, x = - -]]
                       5       5        5
(%i2) minimize_lp(x+y, [3*x+y>0, x+2*y>2]), nonegative_lp=true;
(%o2)                 [1, [y = 1, x = 0]]
(%i3) minimize_lp(x+y, [3*x+y=0, x+2*y>2]), nonegative_lp=true;
(%o3)                Problem not feasible!
(%i4) minimize_lp(x+y, [3*x+y>0]);
(%o4)                Problem not bounded!
```

See also: `maximize_lp`, `nonegative_lp`, `epsilon_lp`.

nonegative_lp Option variable

Default value: `false`

If `nonegative_lp` is true all decision variables to `minimize_lp` and `maximize_lp` are assumed to be positive.

See also: `minimize_lp`.

70 simplification

70.1 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.

70.2 Package absimp

The `absimp` package contains pattern-matching rules that extend the built-in simplification rules for the `abs` and `signum` functions. `absimp` respects relations established with the built-in `assume` function and by declarations such as `modedeclare (m, even, n, odd)` for even or odd integers.

`absimp` defines `unitramp` and `unitstep` functions in terms of `abs` and `signum`.

`load (absimp)` loads this package. `demo (absimp)` shows a demonstration of this package.

Examples:

```
(%i1) load (absimp)$
(%i2) (abs (x))^2;
                                        2
(%o2)                                   x
(%i3) diff (abs (x), x);
                                        x
(%o3)                                 ------
                                      abs(x)
(%i4) cosh (abs (x));
(%o4)                                 cosh(x)
```

70.3 Package facexp

The `facexp` package contains several related functions that provide the user with the ability to structure expressions by controlled expansion. This capability is especially useful when the expression contains variables that have physical meaning, because it is often true that the most economical form of such an expression can be obtained by fully expanding the expression with respect to those variables, and then factoring their coefficients. While it is true that this procedure is not difficult to carry out using standard Maxima functions, additional fine-tuning may also be desirable, and these finishing touches can be more difficult to apply.

The function `facsum` and its related forms provide a convenient means for controlling the structure of expressions in this way. Another function, `collectterms`, can be used to add two or more expressions that have already been simplified to this form, without resimplifying the whole expression again. This function may be useful when the expressions are very large.

`load (facexp)` loads this package. `demo (facexp)` shows a demonstration of this package.

facsum (*expr, arg_1, ..., arg_n*) Function

Returns a form of *expr* which depends on the arguments *arg_1*, ..., *arg_n*. The arguments can be any form suitable for `ratvars`, or they can be lists of such forms. If the arguments are not lists, then the form returned is fully expanded with respect to the arguments, and the coefficients of the arguments are factored. These coefficients are free of the arguments, except perhaps in a non-rational sense.

If any of the arguments are lists, then all such lists are combined into a single list, and instead of calling `factor` on the coefficients of the arguments, `facsum` calls itself on these coefficients, using this newly constructed single list as the new argument list for this recursive call. This process can be repeated to arbitrary depth by nesting the desired elements in lists.

It is possible that one may wish to `facsum` with respect to more complicated subexpressions, such as `log (x + y)`. Such arguments are also permissible. With no variable specification, for example `facsum (`*expr*`)`, the result returned is the same as that returned by `ratsimp (`*expr*`)`.

Occasionally the user may wish to obtain any of the above forms for expressions which are specified only by their leading operators. For example, one may wish to `facsum` with respect to all `log`'s. In this situation, one may include among the arguments either the specific `log`'s which are to be treated in this way, or alternatively, either the expression `operator (log)` or `'operator (log)`. If one wished to `facsum` the expression *expr* with respect to the operators *op_1*, ..., *op_n*, one would evaluate `facsum (`*expr*`, operator (`*op_1*`, ..., `*op_n*`))`. The `operator` form may also appear inside list arguments.

In addition, the setting of the switches `facsum_combine` and `nextlayerfactor` may affect the result of `facsum`.

nextlayerfactor Global variable

Default value: `false`

When `nextlayerfactor` is `true`, recursive calls of `facsum` are applied to the factors of the factored form of the coefficients of the arguments.

When `false`, `facsum` is applied to each coefficient as a whole whenever recusive calls to `facsum` occur.

Inclusion of the atom `nextlayerfactor` in the argument list of `facsum` has the effect of `nextlayerfactor: true`, but for the next level of the expression *only*. Since `nextlayerfactor` is always bound to either `true` or `false`, it must be presented single-quoted whenever it appears in the argument list of `facsum`.

facsum_combine Global variable

Default value: `true`

`facsum_combine` controls the form of the final result returned by `facsum` when its argument is a quotient of polynomials. If `facsum_combine` is `false` then the form will be returned as a fully expanded sum as described above, but if `true`, then the expression returned is a ratio of polynomials, with each polynomial in the form described above.

The `true` setting of this switch is useful when one wants to `facsum` both the numerator and denominator of a rational expression, but does not want the denominator to be multiplied through the terms of the numerator.

factorfacsum (*expr*, *arg_1*, ... *arg_n*) Function

Returns a form of *expr* which is obtained by calling `facsum` on the factors of *expr* with *arg_1*, ... *arg_n* as arguments. If any of the factors of *expr* is raised to a power, both the factor and the exponent will be processed in this way.

collectterms (*expr*, *arg_1*, ..., *arg_n*) Function

If several expressions have been simplified with `facsum`, `factorfacsum`, `factenexpand`, `facexpten` or `factorfacexpten`, and they are to be added together, it may be desirable to combine them using the function `collecterms`. `collecterms` can take as arguments all of the arguments that can be given to these other associated functions with the exception of `nextlayerfactor`, which has no effect on `collectterms`. The advantage of `collectterms` is that it returns a form similar to `facsum`, but since it is adding forms that have already been processed by `facsum`, it does not need to repeat that effort. This capability is especially useful when the expressions to be summed are very large.

70.4 Package functs

rempart (*expr*, *n*) Function

Removes part *n* from the expression *expr*.

If *n* is a list of the form `[`*l*`,` *m*`]` then parts *l* thru *m* are removed.

To use this function write first `load(functs)`.

wronskian ([*f_1*, ..., *f_n*], *x*) Function

Returns the Wronskian matrix of the functions *f_1*, ..., *f_n* in the variable *x*.

f_1, ..., *f_n* may be the names of user-defined functions, or expressions in the variable *x*.

The determinant of the Wronskian matrix is the Wronskian determinant of the set of functions. The functions are linearly dependent if this determinant is zero.

To use this function write first `load(functs)`.

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*`*`*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* `+ 2*`*d*`, ...,` *a* `+ (`*n* `- 1)*`*d*.

To use this function write first `load(functs)`.

geometric (*a*, *r*, *n*) Function

Returns the *n*-th term of the geometric series *a*`,` *a*`*`*r*`,` *a*`*`*r*`^2, ...,` *a*`*`*r*`^(`*n* `- 1)`.

To use this function write first `load(functs)`.

harmonic (*a*, *b*, *c*, *n*) Function

Returns the *n*-th term of the harmonic series *a*/*b*`,` *a*`/(`*b* `+` *c*`),` *a*`/(`*b* `+ 2*`*c*`), ...,` *a*`/(`*b* `+ (`*n* `- 1)*`*c*`)`.

To use this function write first `load(functs)`.

arithsum (*a*, *d*, *n*) Function

Returns the sum of the arithmetic series from 1 to *n*.

To use this function write first `load(functs)`.

geosum (*a*, *r*, *n*) Function

Returns the sum of the geometric series from 1 to *n*. If *n* is infinity (`inf`) then a sum is finite only if the absolute value of *r* is less than 1.

To use this function write first `load(functs)`.

gaussprob (*x*) Function

Returns the Gaussian probability function `%e^(-x^2/2) / sqrt(2*%pi)`.

To use this function write first `load(functs)`.

gd (*x*) Function

Returns the Gudermannian function `2 * atan(%e^x - %pi/2)`.

To use this function write first `load(functs)`.

agd (*x*) Function

Returns the inverse Gudermannian function `log (tan (%pi/4 + x/2)))`.

To use this function write first `load(functs)`.

vers (*x*) Function

Returns the versed sine `1 - cos (x)`.

To use this function write first `load(functs)`.

covers (*x*) Function

Returns the coversed sine `1 - sin (x)`.

To use this function write first `load(functs)`.

exsec (*x*) Function

Returns the exsecant `sec (x) - 1`.

To use this function write first `load(functs)`.

hav (*x*) Function

Returns the haversine `(1 - cos(x))/2`.

To use this function write first `load(functs)`.

combination (*n*, *r*) Function

Returns the number of combinations of *n* objects taken *r* at a time.

To use this function write first `load(functs)`.

permutation (*n*, *r*) Function

Returns the number of permutations of *r* objects selected from a set of *n* objects.

To use this function write first `load(functs)`.

70.5 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.
(%i2) a>=4;  /* a sample inequality */
(%o2)                                a >= 4
(%i3) (b>c)+%; /* add a second, strict inequality */
(%o3)                            b + a > c + 4
(%i4) 7*(x<y); /* multiply by a positive number */
(%o4)                              7 x < 7 y
(%i5) -2*(x>=3*z); /* multiply by a negative number */
(%o5)                            - 2 x <= - 6 z
(%i6) (1+a^2)*(1/(1+a^2)<=1); /* Maxima knows that 1+a^2 > 0 */
                                           2
(%o6)                              1 <= a  + 1
(%i7) assume(x>0)$ x*(2<3); /* assuming x>0 */
(%o7)                               2 x < 3 x
(%i8) a>=b; /* another inequality */
(%o8)                                a >= b
(%i9) 3+%; /* add something */
(%o9)                            a + 3 >= b + 3
(%i10) %-3; /* subtract it out */
(%o10)                               a >= b
(%i11) a>=c-b; /* yet another inequality */
(%o11)                             a >= c - b
(%i12) b+%; /* add b to both sides */
(%o12)                             b + a >= c
(%i13) %-c; /* subtract c from both sides */
(%o13)                          - c + b + a >= 0
(%i14) -%;  /* multiply by -1 */
(%o14)                           c - b - a <= 0
(%i15) (z-1)^2>-2*z; /* determining truth of assertion */
                                     2
(%o15)                         (z - 1)  > - 2 z
(%i16) expand(%)+2*z; /* expand this and add 2*z to both sides */
                                    2
(%o16)                              z  + 1 > 0
(%i17) %,pred;
(%o17)                                 true
```

Be careful about using parentheses around the inequalities: when the user types in `(A > B) + (C = 5)` the result is `A + C > B + 5`, but `A > B + C = 5` is a syntax error, and `(A > B + C) = 5` is something else entirely.

Do `disprule (all)` to see a complete listing of the rule definitions.

The user will be queried if Maxima is unable to decide the sign of a quantity multiplying an inequality.

The most common mis-feature is illustrated by:

```
(%i1) eq: a > b;
(%o1)                                a > b
(%i2) 2*eq;
(%o2)                              2 (a > b)
(%i3) % - eq;
(%o3)                                a > b
```

Another problem is 0 times an inequality; the default to have this turn into 0 has been left alone. However, if you type `X*`*some_inequality* and Maxima asks about the sign of `X` and you respond `zero` (or `z`), the program returns `X*`*some_inequality* and not use the information that `X` is 0. You should do `ev (%, x: 0)` in such a case, as the database will only be used for comparison purposes in decisions, and not for the purpose of evaluating `X`.

The user may note a slower response when this package is loaded, as the simplifier is forced to examine more rules than without the package, so you might wish to remove the rules after making use of them. Do `kill (rules)` to eliminate all of the rules (including any that you might have defined); or you may be more selective by killing only some of them; or use `remrule` on a specific rule.

Note that if you load this package after defining your own rules you will clobber your rules that have the same name. The rules in this package are: `*rule1`, ..., `*rule8`, `+rule1`, ..., `+rule18`, and you must enclose the rulename in quotes to refer to it, as in `remrule ("+", "+rule1")` to specifically remove the first rule on `"+"` or `disprule ("*rule2")` to display the definition of the second multiplicative rule.

70.6 Package rducon

reduce_consts (*expr*) Function

Replaces constant subexpressions of *expr* with constructed constant atoms, saving the definition of all these constructed constants in the list of equations `const_eqns`, and returning the modified *expr*. Those parts of *expr* are constant which return `true` when operated on by the function `constantp`. Hence, before invoking `reduce_consts`, one should do

`declare ([`*objects to be given the constant property*`], constant)$`

to set up a database of the constant quantities occurring in your expressions.

If you are planning to generate Fortran output after these symbolic calculations, one of the first code sections should be the calculation of all constants. To generate this code segment, do

`map ('fortran, const_eqns)$`

Variables besides `const_eqns` which affect `reduce_consts` are:

`const_prefix` (default value: `xx`) is the string of characters used to prefix all symbols generated by `reduce_consts` to represent constant subexpressions.

`const_counter` (default value: 1) is the integer index used to generate unique symbols to represent each constant subexpression found by `reduce_consts`.

`load (rducon)` loads this function. `demo (rducon)` shows a demonstration of this function.

70.7 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.

70.8 Package sqdnst

sqrtdenest (*expr*) Function

Denests `sqrt` of simple, numerical, binomial surds, where possible. E.g.

```
(%i1) load (sqdnst)$
(%i2) sqrt(sqrt(3)/2+1)/sqrt(11*sqrt(2)-12);
                                      sqrt(3)
                                  sqrt(------- + 1)
                                         2
(%o2)                           ---------------------
                                sqrt(11 sqrt(2) - 12)
(%i3) sqrtdenest(%);
                                    sqrt(3)   1
                                    ------- + -
                                       2      2
(%o3)                               -------------
                                       1/4    3/4
                                    3 2    - 2
```

Sometimes it helps to apply `sqrtdenest` more than once, on such as `(19601-13860 sqrt(2))^(7/4)`.

`load (sqdnst)` loads this function.

71 solve_rec

71.1 Introduction to solve_rec

`solve_rec` is a package for solving linear recurrences with polynomial coefficients.

A demo is available with `demo(solve_rec);`.

Example:

```
(%i1) load("solve_rec")$
(%i2) solve_rec((n+4)*s[n+2] + s[n+1] - (n+1)*s[n], s[n]);
                                      n
                  %k  (2 n + 3) (- 1)          %k
                    1                           2
(%o2)       s  = -------------------- + ---------------
             n     (n + 1) (n + 2)      (n + 1) (n + 2)
```

71.2 Functions and Variables for solve_rec

reduce_order (*rec, sol, var*) Function

Reduces the order of linear recurrence *rec* when a particular solution *sol* is known. The reduced reccurence can be used to get other solutions.

Example:

```
(%i3) rec: x[n+2] = x[n+1] + x[n]/n;
                                          x
                                           n
(%o3)                x      = x      + --
                      n + 2    n + 1   n
(%i4) solve_rec(rec, x[n]);
WARNING: found some hypergeometrical solutions!
(%o4)                      x  = %k  n
                            n     1
(%i5) reduce_order(rec, n, x[n]);
(%t5)                      x  = n %z
                            n       n

                              n - 1
                              ====
                              \
(%t6)                 %z  =    >     %u
                        n     /        %j
                              ====
                              %j = 0

(%o6)             (- n - 2) %u     - %u
                                n + 1     n
(%i6) solve_rec((n+2)*%u[n+1] + %u[n], %u[n]);
```

```
                                                  n
                                      %k  (- 1)
                                        1
(%o6)                        %u  = ----------
                              n     (n + 1)!
```

So the general solution is

```
                n - 1
                ====         j
                \       (- 1)
          %k  n  >    -------- + %k  n
            2   /     (j + 1)!     1
                ====
                j = 0
```

simplify_products Option variable

Default value: `true`

If `simplify_products` is `true`, `solve_rec` will try to simplify products in result.

See also: `solve_rec`.

simplify_sum (*expr*) Function

Tries to simplify all sums appearing in *expr* to a closed form.

To use this function first load the `simplify_sum` package with `load(simplify_sum)`.

Example:

```
(%i1) load("simplify_sum")$
(%i2) sum(binom(n+k,k)/2^k, k, 0, n)
                                + sum(binom(2*n, 2*k), k, 0, n);
         n                                  n
        ====                               ====
        \      binomial(n + k, k)          \
(%o2)    >     ------------------ +         >    binomial(2 n, 2 k)
        /                 k                /
        ====             2                ====
        k = 0                              k = 0
(%i3) simplify_sum(%);
                                    n
                                   4     n
(%o3)                              -- + 2
                                   2
```

solve_rec (*eqn*, *var*, [*init*]) Function

Solves for hypergeometrical solutions to linear recurrence *eqn* with polynomials coefficient in variable *var*. Optional arguments *init* are initial conditions.

`solve_rec` can solve linear recurrences with constant coefficients, finds hypergeometrical solutions to homogeneous linear recurrences with polynomial coefficients, rational

solutions to linear recurrences with polynomial coefficients and can solve Ricatti type recurrences.

Note that the running time of the algorithm used to find hypergeometrical solutions is exponential in the degree of the leading and trailing coefficient.

To use this function first load the `solve_rec` package with `load(solve_rec);`.

Example of linear recurrence with constant coefficients:

```
(%i2) solve_rec(a[n]=a[n-1]+a[n-2]+n/2^n, a[n]);
                          n          n
              (sqrt(5) - 1)  %k  (- 1)
                             1           n
(%o2) a  = ------------------------- - ----
       n                n                  n
                       2                5 2
                                                 n
                                    (sqrt(5) + 1)  %k
                                                   2    2
                                 + ------------------ - ----
                                             n             n
                                            2            5 2
```

Example of linear recurrence with polynomial coefficients:

```
(%i7) 2*x*(x+1)*y[x] - (x^2+3*x-2)*y[x+1] + (x-1)*y[x+2];
                            2
(%o7) (x - 1) y      - (x  + 3 x - 2) y      + 2 x (x + 1) y
                x + 2                    x + 1                 x
(%i8) solve_rec(%, y[x], y[1]=1, y[3]=3);
                              x
                          3 2    x!
(%o9)                y  = ---- - --
                      x    4     2
```

Example of Ricatti type recurrence:

```
(%i2) x*y[x+1]*y[x] - y[x+1]/(x+2) + y[x]/(x-1) = 0;
                              y         y
                               x + 1     x
(%o2)         x y  y      - ------ + ----- = 0
                  x  x + 1   x + 2    x - 1
(%i3) solve_rec(%, y[x], y[3]=5)$
(%i4) ratsimp(minfactorial(factcomb(%)));
                                  3
                               30 x  - 30 x
(%o4) y  = - ------------------------------------------------
       x        6      5       4       3       2
              5 x  - 3 x  - 25 x  + 15 x  + 20 x  - 12 x - 1584
```

See also: `solve_rec_rat`, `simplify_products`, and `product_use_gamma`.

solve_rec_rat (*eqn, var,* [*init*]) Function

Solves for rational solutions to linear recurrences. See solve_rec for description of arguments.

To use this function first load the `solve_rec` package with `load(solve_rec);`.

Example:

```
(%i1) (x+4)*a[x+3] + (x+3)*a[x+2] - x*a[x+1] + (x^2-1)*a[x];
(%o1)  (x + 4) a      + (x + 3) a      - x a
                x + 3            x + 2      x + 1
                                                  2
                                             + (x  - 1) a
                                                        x
(%i2) solve_rec_rat(% = (x+2)/(x+1), a[x]);
                          1
(%o2)       a  = ---------------
             x   (x - 1) (x + 1)
```

See also: `solve_rec`.

product_use_gamma Option variable

Default value: `true`

When simplifying products, `solve_rec` introduces gamma function into the expression if `product_use_gamma` is `true`.

See also: `simplify_products`, `solve_rec`.

summand_to_rec (*summand, k, n*) Function
summand_to_rec (*summand, [k, lo, hi], n*) Function

Returns the recurrence sattisfied by the sum

```
 hi
====
\
 >     summand
/
====
k = lo
```

where summand is hypergeometrical in *k* and *n*. If *lo* and *hi* are omited, they are assumed to be `lo = -inf` and `hi = inf`.

To use this function first load the `simplify_sum` package with `load(simplify_sum)`.

Example:

```
(%i1) load("simplify_sum")$
(%i2) summand: binom(n,k);
(%o2)                          binomial(n, k)
(%i3) summand_to_rec(summand,k,n);
(%o3)                    2 sm  - sm      = 0
                            n     n + 1
(%i7) summand: binom(n, k)/(k+1);
                               binomial(n, k)
(%o7)                          --------------
                                   k + 1
(%i8) summand_to_rec(summand, [k, 0, n], n);
(%o8)             2 (n + 1) sm  - (n + 2) sm      = - 1
                               n             n + 1
```

72 stats

72.1 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'.

72.2 Functions and Variables for inference_result

inference_result (*title, values, numbers*) Function

Constructs an `inference_result` object of the type returned by the stats functions. Argument *title* is a string with the name of the procedure; *values* is a list with elements of the form `symbol = value` and *numbers* is a list with positive integer numbers ranging from one to `length(`*values*`)`, indicating which values will be shown by default.

Example:

This is a simple example showing results concerning a rectangle. The title of this object is the string `"Rectangle"`, it stores five results, named `'base`, `'height`, `'diagonal`, `'area`, and `'perimeter`, but only the first, second, fifth, and fourth will be displayed. The `'diagonal` is stored in this object, but it is not displayed; to access its value, make use of function `take_inference`.

```
(%i1) load(inference_result)$
(%i2) b: 3$ h: 2$
(%i3) inference_result("Rectangle",
                       ['base=b,
                        'height=h,
                        'diagonal=sqrt(b^2+h^2),
                        'area=b*h,
                        'perimeter=2*(b+h)],
                       [1,2,5,4] );
                       |   Rectangle
```

```
                              |
                              |    base = 3
                              |
(%o3)                         |   height = 2
                              |
                              | perimeter = 10
                              |
                              |    area = 6
(%i4) take_inference('diagonal,%);
(%o4)                          sqrt(13)
```

See also `take_inference`.

inferencep (*obj*) Function

Returns `true` or `false`, depending on whether *obj* is an `inference_result` object or not.

items_inference (*obj*) Function

Returns a list with the names of the items stored in *obj*, which must be an `inference_result` object.

Example:

The `inference_result` object stores two values, named `'pi` and `'e`, but only the second is displayed. The `items_inference` function returns the names of all items, no matter they are displayed or not.

```
(%i1) load(inference_result)$
(%i2) inference_result("Hi", ['pi=%pi,'e=%e],[2]);
                              |   Hi
(%o2)                         |
                              | e = %e
(%i3) items_inference(%);
(%o3)                          [pi, e]
```

take_inference (*n, obj*) Function
take_inference (*name, obj*) Function
take_inference (*list, obj*) Function

Returns the *n*-th value stored in *obj* if *n* is a positive integer, or the item named *name* if this is the name of an item. If the first argument is a list of numbers and/or symbols, function `take_inference` returns a list with the corresponding results.

Example:

Given an `inference_result` object, function `take_inference` is called in order to extract some information stored in it.

```
(%i1) load(inference_result)$
(%i2) b: 3$ h: 2$
(%i3) sol: inference_result("Rectangle",
                              ['base=b,
                               'height=h,
                               'diagonal=sqrt(b^2+h^2),
```

```
                                    'area=b*h,
                                    'perimeter=2*(b+h)],
                                   [1,2,5,4] );
                              |   Rectangle
                              |
                              |    base = 3
                              |
(%o3)                         |   height = 2
                              |
                              | perimeter = 10
                              |
                              |    area = 6
(%i4) take_inference('base,sol);
(%o4)                              3
(%i5) take_inference(5,sol);
(%o5)                             10
(%i6) take_inference([1,'diagonal],sol);
(%o6)                        [3, sqrt(13)]
(%i7) take_inference(items_inference(sol),sol);
(%o7)                 [3, 2, sqrt(13), 6, 10]
```

See also `inference_result` and `take_inference`.

72.3 Functions and Variables for stats

stats_numer Option variable

Default value: `true`

If `stats_numer` is `true`, inference statistical functions return their results in floating point numbers. If it is `false`, results are given in symbolic and rational format.

test_mean (*x*) Function

test_mean (*x, option_1, option_2, ...*) 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.
4. `'method`: inference procedure.
5. `'hypotheses`: null and alternative hypotheses to be tested.
6. `'statistic`: value of the sample statistic used for testing the null hypothesis.
7. `'distribution`: distribution of the sample statistic, together with its parameter(s).
8. `'p_value`: p-value of the test.

Examples:

Performs an exact t-test with unknown variance. The null hypothesis is $H_0 : mean = 50$ against the one sided alternative $H_1 : mean < 50$; according to the results, the p-value is too great, there are no evidence for rejecting H_0.

```
(%i1) load("stats")$
(%i2) data: [78,64,35,45,45,75,43,74,42,42]$
(%i3) test_mean(data,'conflevel=0.9,'alternative='less,'mean=50);
                 |                 MEAN TEST
                 |
                 |            mean_estimate = 54.3
                 |
                 |              conf_level = 0.9
                 |
                 | conf_interval = [minf, 61.51314273502712]
                 |
(%o3)            |  method = Exact t-test. Unknown variance.
                 |
                 | hypotheses = H0: mean = 50 , H1: mean < 50
                 |
                 |       statistic = .8244705235071678
                 |
                 |       distribution = [student_t, 9]
                 |
                 |        p_value = .7845100411786889
```

This time Maxima performs an asymptotic test, based on the *Central Limit Theorem*. The null hypothesis is $H_0 : equal(mean, 50)$ against the two sided alternative $H_1 : notequal(mean, 50)$; according to the results, the p-value is very small, H_0 should be rejected in favor of the alternative H_1. Note that, as indicated by the `Method` component, this procedure should be applied to large samples.

```
(%i1) load("stats")$
(%i2) test_mean([36,118,52,87,35,256,56,178,57,57,89,34,25,98,35,
              98,41,45,198,54,79,63,35,45,44,75,42,75,45,45,
              45,51,123,54,151],
              'asymptotic=true,'mean=50);
```

```
                    |                        MEAN TEST
                    |
                    |            mean_estimate = 74.88571428571429
                    |
                    |                     conf_level = 0.95
                    |
                    | conf_interval = [57.72848600856194, 92.04294256286663]
                    |
          (%o2)     |    method = Large sample z-test. Unknown variance.
                    |
                    |       hypotheses = H0: mean = 50 , H1: mean # 50
                    |
                    |             statistic = 2.842831192874313
                    |
                    |             distribution = [normal, 0, 1]
                    |
                    |             p_value = .004471474652002261
```

test_means_difference (*x1, x2*) Function
test_means_difference (*x1, x2, option_1, option_2, ...*) Function

This is the difference of means *t*-test for two samples. Arguments *x1* and *x2* are lists or column matrices containing two independent samples. In case of different unknown variances (see options `'dev1`, `'dev2` and `'varequal` bellow), the degrees of freedom are computed by means of the Welch approximation. It also performs an asymptotic test based on the *Central Limit Theorem* if option `'asymptotic` is set to `true`.

Options:

-
- `'alternative`, default `'twosided`, is the alternative hypothesis; valid values are: `'twosided`, `'greater` and `'less`.
- `'dev1`, default `'unknown`, this is the value of the standard deviation of the *x1* sample when it is known; valid values are: `'unknown` or a positive expression.
- `'dev2`, default `'unknown`, this is the value of the standard deviation of the *x2* sample when it is known; valid values are: `'unknown` or a positive expression.
- `'varequal`, default `false`, whether variances should be considered to be equal or not; this option takes effect only when `'dev1` and/or `'dev2` are `'unknown`.
- `'conflevel`, default `95/100`, confidence level for the confidence interval; it must be an expression which takes a value in (0,1).
- `'asymptotic`, default `false`, indicates whether it performs an exact *t*-test or an asymptotic one based on the *Central Limit Theorem*; valid values are `true` and `false`.

The output of function `test_means_difference` is an `inference_result` Maxima object showing the following results:

1. `'diff_estimate`: the difference of means estimate.
2. `'conf_level`: confidence level selected by the user.
3. `'conf_interval`: confidence interval for the difference of means.

4. 'method: inference procedure.
5. 'hypotheses: null and alternative hypotheses to be tested.
6. 'statistic: value of the sample statistic used for testing the null hypothesis.
7. 'distribution: distribution of the sample statistic, together with its parameter(s).
8. 'p_value: p-value of the test.

Examples:

The equality of means is tested with two small samples x and y, against the alternative $H_1 : m_1 > m_2$, being m_1 and m_2 the populations means; variances are unknown and supposed to be different.

```
(%i1) load("stats")$
(%i2) x: [20.4,62.5,61.3,44.2,11.1,23.7]$
(%i3) y: [1.2,6.9,38.7,20.4,17.2]$
(%i4) test_means_difference(x,y,'alternative='greater);
                |              DIFFERENCE OF MEANS TEST
                |
                |         diff_estimate = 20.31999999999999
                |
                |                 conf_level = 0.95
                |
                |    conf_interval = [- .04597417812882298, inf]
                |
(%o4)           |        method = Exact t-test. Welch approx.
                |
                | hypotheses = H0: mean1 = mean2 , H1: mean1 > mean2
                |
                |           statistic = 1.838004300728477
                |
                |    distribution = [student_t, 8.62758740184604]
                |
                |            p_value = .05032746527991905
```

The same test as before, but now variances are supposed to be equal.

```
(%i1) load("stats")$
(%i2) x: [20.4,62.5,61.3,44.2,11.1,23.7]$
(%i3) y: matrix([1.2],[6.9],[38.7],[20.4],[17.2])$
(%i4) test_means_difference(x,y,'alternative='greater,
                                                  'varequal=true);
                |              DIFFERENCE OF MEANS TEST
                |
                |         diff_estimate = 20.31999999999999
                |
                |                 conf_level = 0.95
                |
                |     conf_interval = [- .7722627696897568, inf]
                |
(%o4)           |   method = Exact t-test. Unknown equal variances
                |
```

```
| hypotheses = H0: mean1 = mean2 , H1: mean1 > mean2
|
|           statistic = 1.765996124515009
|
|           distribution = [student_t, 9]
|
|            p_value = .05560320992529344
```

test_variance (*x*) Function
test_variance (*x*, *option_1*, *option_2*, ...) 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]
                |
```

```
(%o3)         | method = Variance Chi-square test. Unknown mean.
              |
              |    hypotheses = H0: var = 200 , H1: var > 200
              |
              |                 statistic = 4.43
              |
              |             distribution = [chi2, 8]
              |
              |           p_value = .8163948512777689
```

test_variance_ratio (*x1, x2*) Function
test_variance_ratio (*x1, x2, option_1, option_2, ...*) Function

This is the variance ratio F-test for two normal populations. Arguments *x1* and *x2* are lists or column matrices containing two independent samples.

Options:

- `'alternative`, default `'twosided`, is the alternative hypothesis; valid values are: `'twosided`, `'greater` and `'less`.
- `'mean1`, default `'unknown`, when it is known, this is the mean of the population from which *x1* was taken.
- `'mean2`, default `'unknown`, when it is known, this is the mean of the population from which *x2* was taken.
- `'conflevel`, default `95/100`, confidence level for the confidence interval of the ratio; it must be an expression which takes a value in (0,1).

The output of function `test_variance_ratio` is an `inference_result` Maxima object showing the following results:

1. `'ratio_estimate`: the sample variance ratio.
2. `'conf_level`: confidence level selected by the user.
3. `'conf_interval`: confidence interval for the variance ratio.
4. `'method`: inference procedure.
5. `'hypotheses`: null and alternative hypotheses to be tested.
6. `'statistic`: value of the sample statistic used for testing the null hypothesis.
7. `'distribution`: distribution of the sample statistic, together with its parameters.
8. `'p_value`: p-value of the test.

Examples:

The equality of the variances of two normal populations is checked against the alternative that the first is greater than the second.

```
(%i1) load("stats")$
(%i2) x: [20.4,62.5,61.3,44.2,11.1,23.7]$
(%i3) y: [1.2,6.9,38.7,20.4,17.2]$
(%i4) test_variance_ratio(x,y,'alternative='greater);
                |              VARIANCE RATIO TEST
                |
```

```
                    |       ratio_estimate = 2.316933391522034
                    |
                    |               conf_level = 0.95
                    |
                    |    conf_interval = [.3703504689507268, inf]
                    |
        (%o4)       | method = Variance ratio F-test. Unknown means.
                    |
                    | hypotheses = H0: var1 = var2 , H1: var1 > var2
                    |
                    |         statistic = 2.316933391522034
                    |
                    |            distribution = [f, 5, 4]
                    |
                    |          p_value = .2179269692254457
```

test_sign (*x*) Function
test_sign (*x*, *option_1*, *option_2*, ...) Function

This is the non parametric sign test for the median of a continuous population. Argument *x* is a list or a column matrix containing a one dimensional sample.

Options:

- `'alternative`, default `'twosided`, is the alternative hypothesis; valid values are: `'twosided`, `'greater` and `'less`.
- `'median`, default `0`, is the median value to be checked.

The output of function `test_sign` is an `inference_result` Maxima object showing the following results:

1. `'med_estimate`: the sample median.
2. `'method`: inference procedure.
3. `'hypotheses`: null and alternative hypotheses to be tested.
4. `'statistic`: value of the sample statistic used for testing the null hypothesis.
5. `'distribution`: distribution of the sample statistic, together with its parameter(s).
6. `'p_value`: *p*-value of the test.

Examples:

Checks whether the population from which the sample was taken has median 6, against the alternative $H_1 : median > 6$.

```
(%i1) load("stats")$
(%i2) x: [2,0.1,7,1.8,4,2.3,5.6,7.4,5.1,6.1,6]$
(%i3) test_sign(x,'median=6,'alternative='greater);
                    |                   SIGN TEST
                    |
                    |              med_estimate = 5.1
                    |
                    |      method = Non parametric sign test.
                    |
```

```
(%o3)          | hypotheses = H0: median = 6 , H1: median > 6
               |
               |                 statistic = 7
               |
               |      distribution = [binomial, 10, 0.5]
               |
               |         p_value = .05468749999999989
```

test_signed_rank (*x*) Function
test_signed_rank (*x, option_1, option_2, ...*) Function

This is the Wilcoxon signed rank test to make inferences about the median of a continuous population. Argument *x* is a list or a column matrix containing a one dimensional sample. Performs normal approximation if the sample size is greater than 20, or if there are zeroes or ties.

See also `pdf_rank_test` and `cdf_rank_test`.

Options:

- `'median`, default `0`, is the median value to be checked.
- `'alternative`, default `'twosided`, is the alternative hypothesis; valid values are: `'twosided`, `'greater` and `'less`.

The output of function `test_signed_rank` is an `inference_result` Maxima object with the following results:

1. `'med_estimate`: the sample median.
2. `'method`: inference procedure.
3. `'hypotheses`: null and alternative hypotheses to be tested.
4. `'statistic`: value of the sample statistic used for testing the null hypothesis.
5. `'distribution`: distribution of the sample statistic, together with its parameter(s).
6. `'p_value`: *p*-value of the test.

Examples:

Checks the null hypothesis $H_0 : median = 15$ against the alternative $H_1 : median > 15$. This is an exact test, since there are no ties.

```
(%i1) load("stats")$
(%i2) x: [17.1,15.9,13.7,13.4,15.5,17.6]$
(%i3) test_signed_rank(x,median=15,alternative=greater);
                  |             SIGNED RANK TEST
                  |
                  |           med_estimate = 15.7
                  |
                  |           method = Exact test
                  |
(%o3)             | hypotheses = H0: med = 15 , H1: med > 15
                  |
                  |              statistic = 14
                  |
```

```
                        |     distribution = [signed_rank, 6]
                        |
                        |            p_value = 0.28125
```

Checks the null hypothesis H_0 : $equal(median, 2.5)$ against the alternative H_1 : $notequal(median, 2.5)$. This is an approximated test, since there are ties.

```
(%i1) load("stats")$
(%i2) y:[1.9,2.3,2.6,1.9,1.6,3.3,4.2,4,2.4,2.9,1.5,3,2.9,4.2,3.1]$
(%i3) test_signed_rank(y,median=2.5);
                |                  SIGNED RANK TEST
                |
                |                med_estimate = 2.9
                |
                |           method = Asymptotic test. Ties
                |
(%o3)           |    hypotheses = H0: med = 2.5 , H1: med # 2.5
                |
                |                 statistic = 76.5
                |
                | distribution = [normal, 60.5, 17.58195097251724]
                |
                |           p_value = .3628097734643669
```

test_rank_sum (*x1, x2*) Function
test_rank_sum (*x1, x2, option_1*) Function

This is the Wilcoxon-Mann-Whitney test for comparing the medians of two continuous populations. The first two arguments *x1* and *x2* are lists or column matrices with the data of two independent samples. Performs normal approximation if any of the sample sizes is greater than 10, or if there are ties.

Option:

- `'alternative`, default `'twosided`, is the alternative hypothesis; valid values are: `'twosided`, `'greater` and `'less`.

The output of function `test_rank_sum` is an `inference_result` Maxima object with the following results:

1. `'method`: inference procedure.
2. `'hypotheses`: null and alternative hypotheses to be tested.
3. `'statistic`: value of the sample statistic used for testing the null hypothesis.
4. `'distribution`: distribution of the sample statistic, together with its parameters.
5. `'p_value`: *p*-value of the test.

Examples:

Checks whether populations have similar medians. Samples sizes are small and an exact test is made.

```
(%i1) load("stats")$
(%i2) x:[12,15,17,38,42,10,23,35,28]$
```

```
(%i3) y:[21,18,25,14,52,65,40,43]$
(%i4) test_rank_sum(x,y);
                |                  RANK SUM TEST
                |
                |              method = Exact test
                |
                | hypotheses = H0: med1 = med2 , H1: med1 # med2
(%o4)           |
                |                  statistic = 22
                |
                |        distribution = [rank_sum, 9, 8]
                |
                |          p_value = .1995886466474702
```

Now, with greater samples and ties, the procedure makes normal approximation. The alternative hypothesis is $H_1 : median1 < median2$.

```
(%i1) load("stats")$
(%i2) x: [39,42,35,13,10,23,15,20,17,27]$
(%i3) y: [20,52,66,19,41,32,44,25,14,39,43,35,19,56,27,15]$
(%i4) test_rank_sum(x,y,'alternative='less);
                |                  RANK SUM TEST
                |
                |          method = Asymptotic test. Ties
                |
                |  hypotheses = H0: med1 = med2 , H1: med1 < med2
(%o4)           |
                |                 statistic = 48.5
                |
                | distribution = [normal, 79.5, 18.95419580097078]
                |
                |           p_value = .05096985666598441
```

test_normality (x) Function

Shapiro-Wilk test for normality. Argument x is a list of numbers, and sample size must be greater than 2 and less or equal than 5000, otherwise, function `test_normality` signals an error message.

Reference:

[1] Algorithm AS R94, Applied Statistics (1995), vol.44, no.4, 547-551

The output of function `test_normality` is an `inference_result` Maxima object with the following results:

1. `'statistic`: value of the W statistic.
2. `'p_value`: p-value under normal assumption.

Examples:

Checks for the normality of a population, based on a sample of size 9.

```
(%i1) load("stats")$
(%i2) x:[12,15,17,38,42,10,23,35,28]$
(%i3) test_normality(x);
```

```
                          |      SHAPIRO - WILK TEST
                          |
(%o3)                     | statistic = .9251055695162436
                          |
                          |  p_value = .4361763918860381
```

simple_linear_regression (x) Function
simple_linear_regression (x *option_1*) Function

Simple linear regression, $y_i = a + bx_i + e_i$, where e_i are $N(0, sigma)$ independent random variables. Argument x must be a two column matrix or a list of pairs.

Options:

- `'conflevel`, default `95/100`, confidence level for the confidence interval; it must be an expression which takes a value in (0,1).
- `'regressor`, default `'x`, name of the independent variable.

The output of function `simple_linear_regression` is an `inference_result` Maxima object with the following results:

1. `'model`: the fitted equation. Useful to make new predictions. See examples bellow.
2. `'means`: bivariate mean.
3. `'variances`: variances of both variables.
4. `'correlation`: correlation coefficient.
5. `'adc`: adjusted determination coefficient.
6. `'a_estimation`: estimation of parameter a.
7. `'a_conf_int`: confidence interval of parameter a.
8. `'b_estimation`: estimation of parameter b.
9. `'b_conf_int`: confidence interval of parameter b.
10. `'hypotheses`: null and alternative hypotheses about parameter b.
11. `'statistic`: value of the sample statistic used for testing the null hypothesis.
12. `'distribution`: distribution of the sample statistic, together with its parameter.
13. `'p_value`: p-value of the test about b.
14. `'v_estimation`: unbiased variance estimation, or residual variance.
15. `'v_conf_int`: variance confidence interval.
16. `'cond_mean_conf_int`: confidence interval for the conditioned mean. See examples bellow.
17. `'new_pred_conf_int`: confidence interval for a new prediction. See examples bellow.
18. `'residuals`: list of pairs (prediction, residual), ordered with respect to predictions. This is useful for goodness of fit analysis. See examples bellow.

Only items 1, 4, 14, 9, 10, 11, 12, and 13 above, in this order, are shown by default. The rest remain hidden until the user makes use of functions `items_inference` and `take_inference`.

Example:

Fitting a linear model to a bivariate sample. Input `%i4` plots the sample together with the regression line; input `%i5` computes `y` given `x=113`; the means and the confidence interval for a new prediction when `x=113` are also calculated.

```
(%i1) load("stats")$
(%i2) s:[[125,140.7], [130,155.1], [135,160.3], [140,167.2],
                                                 [145,169.8]]$
(%i3) z:simple_linear_regression(s,conflevel=0.99);
           |               SIMPLE LINEAR REGRESSION
           |
           |   model = 1.405999999999985 x - 31.18999999999804
           |
           |           correlation = .9611685255255155
           |
           |           v_estimation = 13.57966666666665
           |
(%o3)      | b_conf_int = [.04469633662525263, 2.767303663374718]
           |
           |          hypotheses = H0: b = 0 ,H1: b # 0
           |
           |            statistic = 6.032686683658114
           |
           |            distribution = [student_t, 3]
           |
           |             p_value = 0.0038059549413203
(%i4) plot2d([[discrete, s], take_inference(model,z)],
        [x,120,150],
        [gnuplot_curve_styles, ["with points","with lines"]] )$
(%i5) take_inference(model,z), x=133;
(%o5)                        155.808
(%i6) take_inference(means,z);
(%o6)                     [135.0, 158.62]
(%i7) take_inference(new_pred_conf_int,z), x=133;
(%o7)              [132.0728595995113, 179.5431404004887]
```

72.4 Functions and Variables for special distributions

pdf_signed_rank (*x*, *n*) Function

Probability density function of the exact distribution of the signed rank statistic. Argument *x* is a real number and *n* a positive integer.

See also `test_signed_rank`.

cdf_signed_rank (*x*, *n*) Function

Cumulative density function of the exact distribution of the signed rank statistic. Argument *x* is a real number and *n* a positive integer.

See also `test_signed_rank`.

pdf_rank_sum (*x*, *n*, *m*) Function

Probability density function of the exact distribution of the rank sum statistic. Argument *x* is a real number and *n* and *m* are both positive integers.

See also `test_rank_sum`.

cdf_rank_sum (*x*, *n*, *m*) Function

Cumulative density function of the exact distribution of the rank sum statistic. Argument *x* is a real number and *n* and *m* are both positive integers.

See also `test_rank_sum`.

73 stirling

73.1 Functions and Variables for stirling

stirling (*z*,*n*) Function

Replace `gamma(x)` with the $O(1/x^(2n-1))$ Stirling formula. when n isn't a nonnegative integer, signal an error.

Reference: Abramowitz & Stegun, " Handbook of mathematical functions", 6.1.40.

Examples:

```
(%i1) load (stirling)$

(%i2) stirling(gamma(%alpha+x)/gamma(x),1);
       1/2 - x             x + %alpha - 1/2
(%o2) x        (x + %alpha)
                                    1           1
                                 --------------- - ---- - %alpha
                                 12 (x + %alpha)   12 x
                               %e
(%i3) taylor(%,x,inf,1);
                     %alpha       2    %alpha
          %alpha   x       %alpha  - x       %alpha
(%o3)/T/ x       + -------------------------------- + . . .
                                 2 x
(%i4) map('factor,%);
                                          %alpha - 1
         %alpha   (%alpha - 1) %alpha x
(%o4)   x       + -------------------------------
                                  2
```

The function `stirling` knows the difference between the variable 'gamma' and the function gamma:

```
(%i5) stirling(gamma + gamma(x),0);
                                      x - 1/2   - x
(%o5)    gamma + sqrt(2) sqrt(%pi) x        %e
(%i6) stirling(gamma(y) + gamma(x),0);
                            y - 1/2   - y
(%o6) sqrt(2) sqrt(%pi) y        %e
                                                x - 1/2   - x
                           + sqrt(2) sqrt(%pi) x        %e
```

To use this function write first `load("stirling")`.

74 stringproc

74.1 Introduction to string processing

`stringproc.lisp` enlarges Maximas capabilities of working with strings and adds some useful functions for file in/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.

```
(%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`).

```
(%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.

```
(%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);
(%o6)                     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, sqrt(2), %pi, 0.01, 0.01, 1.0b-2]
(%i10) float(%);
(%o10) [42.0, 1.234, 1.414213562373095, 3.141592653589793, 0.01,
                                                        0.01, 0.01]
(%i11) readline(s);
(%o11)                                false
(%i12) close(s)$
```

`readline` returns `false` when the end of file occurs.

74.2 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~%"$
(%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*) Function

Closes *stream* and returns `true` if *stream* had been open.

flength (*stream*) Function

Returns the number of elements in *stream*.

fposition (*stream*) Function
fposition (*stream*, *pos*) Function

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*) Function

Writes a new line (to *stream*), if the position is not at the beginning of a line. See also `newline`.

newline () Function
newline (*stream*) Function

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*) Function

Makes the Common Lisp function FORMAT available in Maxima. (From gcl.info: "format produces formatted output by outputting the characters of control-string string and observing that a tilde introduces a directive. The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies what kind of formatting is desired. Most directives use one or more elements of 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.

```
~%      new line
~&      fresh line
~t      tab
~$      monetary
~d      decimal integer
~b      binary integer
~o      octal integer
~x      hexadecimal integer
~br     base-b integer
```

```
~r       spell an integer
~p       plural
~f       floating point
~e       scientific notation
~g       ~f or ~e, depending upon magnitude
~h       bigfloat
~a       uses Maxima function string
~s       like ~a, but output enclosed in "double quotes"
~~       ~
~<       justification, ~> terminates
~(       case conversion, ~) terminates
~[       selection, ~] terminates
~{       iteration, ~} terminates
```

Note that the selection directive `~[` is zero-indexed. Also note that the directive `~*` is not supported.

```
(%i1) printf( false, "~a ~a ~4f ~a ~@r",
              "String",sym,bound,sqrt(12),144), bound = 1.234;
(%o1)                 String sym 1.23 2*sqrt(3) CXLIV
(%i2) printf( false,"~{~a ~}",["one",2,"THREE"] );
(%o2)                          one 2 THREE
(%i3) printf(true,"~{~{~9,1f ~}~%~}",mat ),
          mat = args(matrix([1.1,2,3.33],[4,5,6],[7,8.88,9]))$
     1.1       2.0       3.3
     4.0       5.0       6.0
     7.0       8.9       9.0
(%i4) control: "~:(~r~) bird~p ~[is~;are~] singing."$
(%i5) printf( false,control, n,n,if n=1 then 0 else 1 ), n=2;
(%o5)                     Two birds are singing.
```

If *dest* is a stream or `true`, then `printf` returns `false`. Otherwise, `printf` returns a string containing the output.

readline (*stream*) Function

Returns a string containing the characters from the current position in *stream* up to the end of the line or *false* if the end of the file is encountered.

sprint (*expr_1, ..., expr_n*) Function

Evaluates and displays its arguments one after the other 'on a line' starting at the leftmost position. The numbers are printed with the '-' right next to the number, and it disregards line length. `newline()`, which will be autoloaded from `stringproc.lisp` might be useful, if you whish to place intermediate line breaking.

```
(%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
```

74.3 Functions and Variables for characters

alphacharp (*char*) Function
Returns `true` if *char* is an alphabetic character.

alphanumericp (*char*) Function
Returns `true` if *char* is an alphabetic character or a digit.

ascii (*int*) Function
Returns the character corresponding to the ASCII number *int*. (-1 < int < 256)

```
(%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*) Function
Returns `true` if *char_1* and *char_2* are the same.

cequalignore (*char_1*, *char_2*) Function
Like `cequal` but ignores case.

cgreaterp (*char_1*, *char_2*) Function
Returns `true` if the ASCII number of *char_1* is greater than the number of *char_2*.

cgreaterpignore (*char_1*, *char_2*) Function
Like `cgreaterp` but ignores case.

charp (*obj*) Function
Returns `true` if *obj* is a Maxima-character. See introduction for example.

cint (*char*) Function
Returns the ASCII number of *char*.

clessp (*char_1*, *char_2*) Function
Returns `true` if the ASCII number of *char_1* is less than the number of *char_2*.

clesspignore (*char_1*, *char_2*) Function
Like `clessp` but ignores case.

constituent (*char*) Function
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.)

```
(%i1) for n from 0 thru 255 do (
tmp: ascii(n), if constituent(tmp) then sprint(tmp) )$
!  "  #  %  '  (  )  *  +  ,  -  .  /  0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?  @  A  B
C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _  `  a  b  c
d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~
```

cunlisp (*lisp_char*) Function

Converts a Lisp-character into a Maxima-character. (You won't need it.)

digitcharp (*char*) Function

Returns `true` if *char* is a digit.

lcharp (*obj*) Function

Returns `true` if *obj* is a Lisp-character. (You won't need it.)

lowercasep (*char*) Function

Returns `true` if *char* is a lowercase character.

newline Variable

The newline character.

space Variable

The space character.

tab Variable

The tab character.

uppercasep (*char*) Function

Returns `true` if *char* is an uppercase character.

74.4 Functions and Variables for strings

stringp (*obj*) Function

Returns `true` if *obj* is a string. See introduction for example.

charat (*string*, *n*) Function

Returns the *n*-th character of *string*. The first character in *string* is returned with *n* = 1.

```
(%i1) charat("Lisp",1);
(%o1)                          L
```

charlist (*string*) Function

Returns the list of all characters in *string*.

```
(%i1) charlist("Lisp");
(%o1)                     [L, i, s, p]
(%i2) %[1];
(%o2)                          L
```

eval_string (*str*) Function

Parse the string *str* as a Maxima expression and evaluate it. The string *str* may or may not have a terminator (dollar sign `$` or semicolon `;`). Only the first expression is parsed and evaluated, if there is more than one.

Complain if *str* is not a string.

Examples:

```
(%i1) eval_string ("foo: 42; bar: foo^2 + baz");
(%o1)                         42
(%i2) eval_string ("(foo: 42, bar: foo^2 + baz)");
(%o2)                     baz + 1764
```

See also `parse_string`.

parse_string (*str*) Function

Parse the string *str* as a Maxima expression (do not evaluate it). The string *str* may or may not have a terminator (dollar sign `$` or semicolon `;`). Only the first expression is parsed, if there is more than one.

Complain if *str* is not a string.

Examples:

```
(%i1) parse_string ("foo: 42; bar: foo^2 + baz");
(%o1)                       foo : 42
(%i2) parse_string ("(foo: 42, bar: foo^2 + baz)");
                                      2
(%o2)          (foo : 42, bar : foo  + baz)
```

See also `eval_string`.

scopy (*string*) Function

Returns a copy of *string* as a new string.

sdowncase (*string*) Function
sdowncase (*string*, *start*) Function
sdowncase (*string*, *start*, *end*) Function

Like `supcase`, but uppercase characters are converted to lowercase.

sequal (*string_1*, *string_2*) Function

Returns `true` if *string_1* and *string_2* are the same length and contain the same characters.

sequalignore (*string_1*, *string_2*) Function

Like `sequal` but ignores case.

sexplode (*string*) Function

`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)]);
(%o1)             xx[3]:y^3+3*x*y^2+3*x^2*y+x^3
(%i2) simplode( sexplode("stars")," * " );
(%o2)                   s * t * a * r * s
(%i3) simplode( ["One","more","coffee."]," " );
(%o3)                   One more coffee.
```

sinsert (*seq, string, pos*) Function

Returns a string that is a concatenation of `substring (`*string*`, 1, `*pos*` - 1)`, the string *seq* and `substring (`*string*`, `*pos*`)`. Note that the first character in *string* is in position 1.

```
(%i1) s: "A submarine."$
(%i2) concat( substring(s,1,3),"yellow ",substring(s,3) );
(%o2)                  A yellow submarine.
(%i3) sinsert("hollow ",s,3);
(%o3)                  A hollow submarine.
```

sinvertcase (*string*) Function
sinvertcase (*string, start*) Function
sinvertcase (*string, start, end*) Function

Returns *string* except that each character from position *start* to *end* is inverted. If *end* is not given, all characters from *start* to the end of *string* are replaced.

```
(%i1) sinvertcase("sInvertCase");
(%o1)                      SiNVERTcASE
```

slength (*string*) Function

Returns the number of characters in *string*.

smake (*num, char*) Function

Returns a new string with a number of *num* characters *char*.

```
(%i1) smake(3,"w");
(%o1)                          www
```

smismatch (*string_1, string_2*) Function
smismatch (*string_1, string_2, test*) Function

Returns the position of the first character of *string_1* at which *string_1* and *string_2* differ or `false`. Default test function for matching is `sequal`. If `smismatch` should ignore case, use `sequalignore` as test.

```
(%i1) smismatch("seven","seventh");
(%o1)                           6
```

split (*string*) Function
split (*string*, *delim*) Function
split (*string*, *delim*, *multiple*) Function

Returns the list of all tokens in *string*. Each token is an unparsed string. `split` uses *delim* as delimiter. If *delim* is not given, the space character is the default delimiter. *multiple* is a boolean variable with `true` by default. Multiple delimiters are read as one. This is useful if tabs are saved as multiple space characters. If *multiple* is set to `false`, each delimiter is noted.

```
(%i1) split("1.2   2.3   3.4   4.5");
(%o1)                  [1.2, 2.3, 3.4, 4.5]
(%i2) split("first;;third;fourth",";",false);
(%o2)               [first, , third, fourth]
```

sposition (*char*, *string*) Function

Returns the position of the first character in *string* which matches *char*. The first character in *string* is in position 1. For matching characters ignoring case see `ssearch`.

sremove (*seq*, *string*) Function
sremove (*seq*, *string*, *test*) Function
sremove (*seq*, *string*, *test*, *start*) Function
sremove (*seq*, *string*, *test*, *start*, *end*) Function

Returns a string like *string* but without all substrings matching *seq*. Default test function for matching is `sequal`. 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.

```
(%i1) sremove("n't","I don't like coffee.");
(%o1)                    I do like coffee.
(%i2) sremove ("DO ",%,'sequalignore);
(%o2)                     I like coffee.
```

sremovefirst (*seq*, *string*) Function
sremovefirst (*seq*, *string*, *test*) Function
sremovefirst (*seq*, *string*, *test*, *start*) Function
sremovefirst (*seq*, *string*, *test*, *start*, *end*) Function

Like `sremove` except that only the first substring that matches `seq` is removed.

sreverse (*string*) Function

Returns a string with all the characters of *string* in reverse order.

ssearch (*seq*, *string*) Function
ssearch (*seq*, *string*, *test*) Function
ssearch (*seq*, *string*, *test*, *start*) Function
ssearch (*seq*, *string*, *test*, *start*, *end*) Function

Returns the position of the first substring of *string* that matches the string *seq*. Default test function for matching is `sequal`. 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.

```
(%i1) ssearch("~s","~{~S ~}~%",'sequalignore);
(%o1)                                4
```

ssort (*string*) Function
ssort (*string, test*) Function

Returns a string that contains all characters from *string* in an order such there are no two successive characters *c* and *d* such that `test (`*c*`, `*d*`)` is `false` and `test (`*d*`, `*c*`)` is `true`. Default test function for sorting is *clessp*. 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)                 ytsoonnMlkIiedda.'
```

ssubst (*new, old, string*) Function
ssubst (*new, old, string, test*) Function
ssubst (*new, old, string, test, start*) Function
ssubst (*new, old, string, test, start, end*) Function

Returns a string like *string* except that all substrings matching *old* are replaced by *new*. *old* and *new* need not to be of the same length. Default test function for matching is `sequal`. If `ssubst` should ignore case while searching for old, use `sequalignore` as test. Use *start* and *end* to limit searching. Note that the first character in *string* is in position 1.

```
(%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*) Function

Like `strim` except that only the right end of string is trimmed.

substring (*string, start*) Function
substring (*string, start, end*) Function

Returns the substring of *string* beginning at position *start* and ending at position *end*. The character at position *end* is not included. If *end* is not given, the substring contains the rest of the string. Note that the first character in *string* is in position 1.

```
(%i1) substring("substring",4);
(%o1)                         string
(%i2) substring(%,4,6);
(%o2)                           in
```

supcase (*string*) Function
supcase (*string, start*) Function
supcase (*string, start, end*) Function

Returns *string* except that lowercase characters from position *start* to *end* are replaced by the corresponding uppercase ones. If *end* is not given, all lowercase characters from *start* to the end of *string* are replaced.

```
(%i1) supcase("english",1,2);
(%o1)                         English
```

tokens (*string*) Function
tokens (*string, test*) Function

Returns a list of tokens, which have been extracted from *string*. The tokens are substrings whose characters satisfy a certain test function. If test is not given, *constituent* is used as the default test. `{constituent, alphacharp, digitcharp, lowercasep, uppercasep, charp, characterp, alphanumericp}` is the set of test functions. (The Lisp-version of `tokens` is written by Paul Graham. ANSI Common Lisp, 1996, page 67.)

```
(%i1) tokens("24 October 2005");
(%o1)                    [24, October, 2005]
(%i2) tokens("05-10-24",'digitcharp);
(%o2)                       [05, 10, 24]
(%i3) map(parse_string,%);
(%o3)                        [5, 10, 24]
```

75 unit

75.1 Introduction to Units

The *unit* package enables the user to convert between arbitrary units and work with dimensions in equations. The functioning of this package is radically different from the original Maxima units package - whereas the original was a basic list of definitions, this package uses rulesets to allow the user to chose, on a per dimension basis, what unit final answers should be rendered in. It will separate units instead of intermixing them in the display, allowing the user to readily identify the units associated with a particular answer. It will allow a user to simplify an expression to its fundamental Base Units, as well as providing fine control over simplifying to derived units. Dimensional analysis is possible, and a variety of tools are available to manage conversion and simplification options. In addition to customizable automatic conversion, *units* also provides a traditional manual conversion option.

Note - when unit conversions are inexact Maxima will make approximations resulting in fractions. This is a consequence of the techniques used to simplify units. The messages warning of this type of substitution are disabled by default in the case of units (normally they are on) since this situation occurs frequently and the warnings clutter the output. (The existing state of ratprint is restored after unit conversions, so user changes to that setting will be preserved otherwise.) If the user needs this information for units, they can set *unitverbose:on* to reactivate the printing of warnings from the unit conversion process.

unit is included in Maxima in the share/contrib/unit directory. It obeys normal Maxima package loading conventions:

```
(%i1) load("unit")$
************************************************************************
*                        Units version 0.50                            *
*          Definitions based on the NIST Reference on                  *
*              Constants, Units, and Uncertainty                       *
*       Conversion factors from various sources including              *
*                   NIST and the GNU units package                     *
************************************************************************

Redefining necessary functions...
WARNING: DEFUN/DEFMACRO: redefining function TOPLEVEL-MACSYMA-EVAL ...
WARNING: DEFUN/DEFMACRO: redefining function MSETCHK ...
WARNING: DEFUN/DEFMACRO: redefining function KILL1 ...
WARNING: DEFUN/DEFMACRO: redefining function NFORMAT ...
Initializing unit arrays...
Done.
```

The WARNING messages are expected and not a cause for concern - they indicate the *unit* package is redefining functions already defined in Maxima proper. This is necessary in order to properly handle units. The user should be aware that if other changes have been made to these functions by other packages those changes will be overwritten by this loading process.

The *unit.mac* file also loads a lisp file *unit-functions.lisp* which contains the lisp functions needed for the package.

Clifford Yapp is the primary author. He has received valuable assistance from Barton Willis of the University of Nebraska at Kearney (UNK), Robert Dodier, and other intrepid folk of the Maxima mailing list.

There are probably lots of bugs. Let me know. `float` and `numer` don't do what is expected.

TODO : dimension functionality, handling of temperature, showabbr and friends. Show examples with addition of quantities containing units.

75.2 Functions and Variables for Units

setunits (*list*) Function

By default, the *unit* package does not use any derived dimensions, but will convert all units to the seven fundamental dimensions using MKS units.

```
(%i2) N;
                                    kg m
(%o2)                               ----
                                      2
                                     s
(%i3) dyn;
                                  1      kg m
(%o3)                           (------) (----)
                                 100000     2
                                           s
(%i4) g;
                                    1
(%o4)                              (----) (kg)
                                    1000
(%i5) centigram*inch/minutes^2;
                                   127        kg m
(%o5)                        (-------------) (----)
                              1800000000000     2
                                               s
```

In some cases this is the desired behavior. If the user wishes to use other units, this is achieved with the `setunits` command:

```
(%i6) setunits([centigram,inch,minute]);
(%o6)                                done
(%i7) N;
                              1800000000000   %in cg
(%o7)                        (-------------) (------)
                                   127            2
                                                %min
(%i8) dyn;
                                 18000000   %in cg
(%o8)                           (--------) (------)
```

```
                                      127          2
                                              %min
(%i9) g;
(%o9)                                (100) (cg)
(%i10) centigram*inch/minutes^2;
                                       %in cg
(%o10)                                 ------
                                           2
                                       %min
```

The setting of units is quite flexible. For example, if we want to get back to kilograms, meters, and seconds as defaults for those dimensions we can do:

```
(%i11) setunits([kg,m,s]);
(%o11)                                done
(%i12) centigram*inch/minutes^2;
                                     127        kg m
(%o12)                         (-------------) (----)
                                1800000000000     2
                                                  s
```

Derived units are also handled by this command:

```
(%i17) setunits(N);
(%o17)                                done
(%i18) N;
(%o18)                                  N
(%i19) dyn;
                                       1
(%o19)                               (------) (N)
                                     100000
(%i20) kg*m/s^2;
(%o20)                                  N
(%i21) centigram*inch/minutes^2;
                                      127
(%o21)                          (-------------) (N)
                                 1800000000000
```

Notice that the *unit* package recognized the non MKS combination of mass, length, and inverse time squared as a force, and converted it to Newtons. This is how Maxima works in general. If, for example, we prefer dyne to Newtons, we simply do the following:

```
(%i22) setunits(dyn);
(%o22)                                done
(%i23) kg*m/s^2;
(%o23)                           (100000) (dyn)
(%i24) centigram*inch/minutes^2;
                                      127
(%o24)                           (--------) (dyn)
                                  18000000
```

To discontinue simplifying to any force, we use the uforget command:

```
(%i26) uforget(dyn);
```

```
(%o26)                                  false
(%i27) kg*m/s^2;
                                        kg m
(%o27)                                  ----
                                          2
                                         s
(%i28) centigram*inch/minutes^2;
                                    127        kg m
(%o28)                         (-------------) (----)
                                1800000000000     2
                                                 s
```

This would have worked equally well with `uforget(N)` or `uforget(%force)`.

See also `uforget`. To use this function write first `load("unit")`.

uforget (*list*) Function

By default, the *unit* package converts all units to the seven fundamental dimensions using MKS units. This behavior can be changed with the `setunits` command. After that, the user can restore the default behavior for a particular dimension by means of the `uforget` command:

```
(%i13) setunits([centigram,inch,minute]);
(%o13)                                done
(%i14) centigram*inch/minutes^2;
                                      %in cg
(%o14)                                ------
                                          2
                                      %min
(%i15) uforget([cg,%in,%min]);
(%o15)                     [false, false, false]
(%i16) centigram*inch/minutes^2;
                                    127        kg m
(%o16)                         (-------------) (----)
                                1800000000000     2
                                                 s
```

`uforget` operates on dimensions, not units, so any unit of a particular dimension will work. The dimension itself is also a legal argument.

See also `setunits`. To use this function write first `load("unit")`.

convert (*expr, list*) Function

When resetting the global environment is overkill, there is the `convert` command, which allows one time conversions. It can accept either a single argument or a list of units to use in conversion. When a convert operation is done, the normal global evaluation system is bypassed, in order to avoid the desired result being converted again. As a consequence, for inexact calculations "rat" warnings will be visible if the global environment controlling this behavior (`ratprint`) is true. This is also useful for spot-checking the accuracy of a global conversion. Another feature is `convert` will allow a user to do Base Dimension conversions even if the global environment is set to simplify to a Derived Dimension.

```
(%i2) kg*m/s^2;
                                    kg m
(%o2)                               ----
                                      2
                                     s
(%i3) convert(kg*m/s^2,[g,km,s]);
                                    g km
(%o3)                               ----
                                      2
                                     s
(%i4) convert(kg*m/s^2,[g,inch,minute]);

'rat' replaced 39.37007874015748 by 5000/127 = 39.37007874015748
                              18000000000   %in g
(%o4)                        (-----------) (-----)
                                  127           2
                                             %min
(%i5) convert(kg*m/s^2,[N]);
(%o5)                                N
(%i6) convert(kg*m^2/s^2,[N]);
(%o6)                               m N
(%i7) setunits([N,J]);
(%o7)                               done
(%i8) convert(kg*m^2/s^2,[N]);
(%o8)                               m N
(%i9) convert(kg*m^2/s^2,[N,inch]);

'rat' replaced 39.37007874015748 by 5000/127 = 39.37007874015748
                               5000
(%o9)                         (----) (%in N)
                               127
(%i10) convert(kg*m^2/s^2,[J]);
(%o10)                               J
(%i11) kg*m^2/s^2;
(%o11)                               J
(%i12) setunits([g,inch,s]);
(%o12)                              done
(%i13) kg*m/s^2;
(%o13)                               N
(%i14) uforget(N);
(%o14)                              false
(%i15) kg*m/s^2;
                              5000000   %in g
(%o15)                       (-------) (-----)
                               127       2
                                        s
(%i16) convert(kg*m/s^2,[g,inch,s]);

'rat' replaced 39.37007874015748 by 5000/127 = 39.37007874015748
```

```
                                      5000000   %in g
(%o16)                               (-------) (-----)
                                        127       2
                                                  s
```

See also `setunits` and `uforget`. To use this function write first `load("unit")`.

usersetunits Optional variable

Default value: none

If a user wishes to have a default unit behavior other than that described, they can make use of *maxima-init.mac* and the *usersetunits* variable. The *unit* package will check on startup to see if this variable has been assigned a list. If it has, it will use setunits on that list and take the units from that list to be defaults. `uforget` will revert to the behavior defined by usersetunits over its own defaults. For example, if we have a *maxima-init.mac* file containing:

```
usersetunits : [N,J];
```

we would see the following behavior:

```
(%i1) load("unit")$
*************************************************************
*                      Units version 0.50                   *
*          Definitions based on the NIST Reference on       *
*              Constants, Units, and Uncertainty            *
*       Conversion factors from various sources including   *
*                   NIST and the GNU units package          *
*************************************************************

Redefining necessary functions...
WARNING: DEFUN/DEFMACRO: redefining function
 TOPLEVEL-MACSYMA-EVAL ...
WARNING: DEFUN/DEFMACRO: redefining function MSETCHK ...
WARNING: DEFUN/DEFMACRO: redefining function KILL1 ...
WARNING: DEFUN/DEFMACRO: redefining function NFORMAT ...
Initializing unit arrays...
Done.
User defaults found...
User defaults initialized.
(%i2) kg*m/s^2;
(%o2)                                  N
(%i3) kg*m^2/s^2;
(%o3)                                  J
(%i4) kg*m^3/s^2;
(%o4)                                 J m
(%i5) kg*m*km/s^2;
(%o5)                            (1000) (J)
(%i6) setunits([dyn,eV]);
(%o6)                               done
(%i7) kg*m/s^2;
(%o7)                          (100000) (dyn)
(%i8) kg*m^2/s^2;
```

```
(%o8)                      (6241509596477042688) (eV)
(%i9) kg*m^3/s^2;
(%o9)                    (6241509596477042688) (eV m)
(%i10) kg*m*km/s^2;
(%o10)                   (62415095964770426880000) (eV)
(%i11) uforget([dyn,eV]);
(%o11)                           [false, false]
(%i12) kg*m/s^2;
                                       N
(%o12)
(%i13) kg*m^2/s^2;
                                       J
(%o13)
(%i14) kg*m^3/s^2;
                                     J m
(%o14)
(%i15) kg*m*km/s^2;
                                (1000) (J)
(%o15)
```

Without `usersetunits`, the initial inputs would have been converted to MKS, and uforget would have resulted in a return to MKS rules. Instead, the user preferences are respected in both cases. Notice these can still be overridden if desired. To completely eliminate this simplification - i.e. to have the user defaults reset to factory defaults - the `dontusedimension` command can be used. `uforget` can restore user settings again, but only if `usedimension` frees it for use. Alternately, `kill(usersetunits)` will completely remove all knowledge of the user defaults from the session. Here are some examples of how these various options work.

```
(%i2) kg*m/s^2;
                                       N
(%o2)
(%i3) kg*m^2/s^2;
                                       J
(%o3)
(%i4) setunits([dyn,eV]);
                                    done
(%o4)
(%i5) kg*m/s^2;
                                (100000) (dyn)
(%o5)
(%i6) kg*m^2/s^2;
                           (6241509596477042688) (eV)
(%o6)
(%i7) uforget([dyn,eV]);
                                [false, false]
(%o7)
(%i8) kg*m/s^2;
                                       N
(%o8)
(%i9) kg*m^2/s^2;
                                       J
(%o9)
(%i10) dontusedimension(N);
                                   [%force]
(%o10)
(%i11) dontusedimension(J);
                                [%energy, %force]
(%o11)
(%i12) kg*m/s^2;
                                     kg m
                                     ----
(%o12)                                 2
                                       s
```

```
(%i13) kg*m^2/s^2;
                                        2
                                     kg m
(%o13)                               -----
                                       2
                                      s
(%i14) setunits([dyn,eV]);
(%o14)                               done
(%i15) kg*m/s^2;
                                     kg m
(%o15)                               ----
                                       2
                                      s
(%i16) kg*m^2/s^2;
                                        2
                                     kg m
(%o16)                               -----
                                       2
                                      s
(%i17) uforget([dyn,eV]);
(%o17)                       [false, false]
(%i18) kg*m/s^2;
                                     kg m
(%o18)                               ----
                                       2
                                      s
(%i19) kg*m^2/s^2;
                                        2
                                     kg m
(%o19)                               -----
                                       2
                                      s
(%i20) usedimension(N);
Done.  To have Maxima simplify to this dimension, use
setunits([unit]) to select a unit.
(%o20)                               true
(%i21) usedimension(J);
Done.  To have Maxima simplify to this dimension, use
setunits([unit]) to select a unit.
(%o21)                               true
(%i22) kg*m/s^2;
                                     kg m
(%o22)                               ----
                                       2
                                      s
(%i23) kg*m^2/s^2;
                                        2
                                     kg m
(%o23)                               -----
```

```
                                        2
                                       s
(%i24) setunits([dyn,eV]);
(%o24)                                done
(%i25) kg*m/s^2;
(%o25)                           (100000) (dyn)
(%i26) kg*m^2/s^2;
(%o26)                  (6241509596477042688) (eV)
(%i27) uforget([dyn,eV]);
(%o27)                            [false, false]
(%i28) kg*m/s^2;
(%o28)                                 N
(%i29) kg*m^2/s^2;
(%o29)                                 J
(%i30) kill(usersetunits);
(%o30)                                done
(%i31) uforget([dyn,eV]);
(%o31)                           [false, false]
(%i32) kg*m/s^2;
                                      kg m
(%o32)                                ----
                                        2
                                       s
(%i33) kg*m^2/s^2;
                                         2
                                      kg m
(%o33)                                -----
                                        2
                                       s
```

Unfortunately this wide variety of options is a little confusing at first, but once the user grows used to them they should find they have very full control over their working environment.

metricexpandall (x) Function

Rebuilds global unit lists automatically creating all desired metric units. x is a numerical argument which is used to specify how many metric prefixes the user wishes defined. The arguments are as follows, with each higher number defining all lower numbers' units:

```
            0 - none. Only base units
            1 - kilo, centi, milli
(default)   2 - giga, mega, kilo, hecto, deka, deci, centi, milli,
                micro, nano
            3 - peta, tera, giga, mega, kilo, hecto, deka, deci,
                centi, milli, micro, nano, pico, femto
            4 - all
```

Normally, Maxima will not define the full expansion since this results in a very large number of units, but `metricexpandall` can be used to rebuild the list in a more or less complete fashion. The relevant variable in the *unit.mac* file is *%unitexpand*.

%unitexpand Variable

Default value: `2`

This is the value supplied to `metricexpandall` during the initial loading of *unit*.

76 zeilberger

76.1 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.

76.1.0.1 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 $F_k = f_(k+1) - f_k$.

76.1.0.2 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

$a_0 F_(n,k) + ... + a_d F_(n+d), k = Delta_K(R(n,k)F_(n,k))$

where $Delta_k$ is the k-forward difference operator, i.e., $Delta_k(t_k) := t_(k+1) - t_k$.

76.1.1 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 intermidiate steps

`VeryVerbose`
More information

`Extra` Even more information including information on the linear system in Zeilberger's algorithm

For example: `GosperVerbose`, `parGosperVeryVerbose`, `ZeilbergerExtra`, `AntiDifferenceSummary`.

76.2 Functions and Variables for zeilberger

AntiDifference (*F_k, k*) Function

Returns the hypergeometric anti-difference of *F_k*, if it exists. Otherwise `AntiDifference` returns `no_hyp_antidifference`.

Gosper (*F_k, k*) Function

Returns the rational certificate *R(k)* for *F_k*, that is, a rational function such that

$F_k = R(k+1)F_(k+1) - R(k)F_k$

if it exists. Otherwise, `Gosper` returns `no_hyp_sol`.

GosperSum (*F_k, k, a, b*) Function

Returns the summmation of *F_k* from $k = a$ to $k = b$ if *F_k* has a hypergeometric anti-difference. Otherwise, `GosperSum` returns `nongosper_summable`.

Examples:

```
(%i1) load (zeilberger);
(%o1)  /usr/share/maxima/share/contrib/Zeilberger/zeilberger.mac
(%i2) GosperSum ((-1)^k*k / (4*k^2 - 1), k, 1, n);

Dependent equations eliminated:  (1)
                            3       n + 1
                      (n + -) (- 1)
                            2               1
(%o2)               - ------------------ - -
                                  2         4
                      2 (4 (n + 1)  - 1)
(%i3) GosperSum (1 / (4*k^2 - 1), k, 1, n);
                                3
                           - n - -
                                2       1
(%o3)                   -------------- + -
                                2       2
                       4 (n + 1)  - 1
(%i4) GosperSum (x^k, k, 1, n);
                           n + 1
                          x          x
(%o4)                     ------ - -----
                          x - 1    x - 1
(%i5) GosperSum ((-1)^k*a! / (k!*(a - k)!), k, 1, n);
                                n + 1
                 a! (n + 1) (- 1)                a!
(%o5)       - ------------------------ - ----------
              a (- n + a - 1)! (n + 1)!   a (a - 1)!
(%i6) GosperSum (k*k!, k, 1, n);

Dependent equations eliminated:  (1)
(%o6)                     (n + 1)! - 1
```

```
(%i7) GosperSum ((k + 1)*k! / (k + 1)!, k, 1, n);
                    (n + 1) (n + 2) (n + 1)!
(%o7)               ------------------------ - 1
                              (n + 2)!
(%i8) GosperSum (1 / ((a - k)!*k!), k, 1, n);
(%o8)                    nonGosper_summable
```

parGosper (*F_{n,k}*, *k*, *n*, *d*) Function

Attempts to find a *d*-th order recurrence for *F_{n,k}*.

The algorithm yields a sequence $[s_1, s_2, ..., s_m]$ of solutions. Each solution has the form

$[R(n,k), [a_0, a_1, ..., a_d]]$

`parGosper` returns `[]` if it fails to find a recurrence.

Zeilberger (*F_{n,k}*, *k*, *n*) Function

Attempts to compute the indefinite hypergeometric summation of *F_{n,k}*.

`Zeilberger` first invokes `Gosper`, and if that fails to find a solution, then invokes `parGosper` with order 1, 2, 3, ..., up to `MAX_ORD`. If Zeilberger finds a solution before reaching `MAX_ORD`, it stops and returns the solution.

The algorithms yields a sequence $[s_1, s_2, ..., s_m]$ of solutions. Each solution has the form

$[R(n,k), [a_0, a_1, ..., a_d]]$

`Zeilberger` returns `[]` if it fails to find a solution.

`Zeilberger` invokes `Gosper` only if `gosper_in_zeilberger` is `true`.

76.3 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.

76.4 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.

77 Indices

Appendix A Function and Variable Index

~

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