Manual do Maxima

Versão 5.47post

Tradução para português de Portugal

Maxima é um Sistema de Computação Algébrica, programado em Lisp.

Maxima derivou-se do sistema Macsyma, desenvolvido no MIT entre 1968 e 1982, como parte do Projecto MAC. O MIT transferiu uma cópia do código fonte do Macsyma para o Departamento de Energia em 1982, que ficou conhecida como Macsyma DOE e o Professor William F. Schelter, da Universidade do Texas, desenvolveu essa versão desde 1982 até a sua morte em 2001. Em 1998, Schelter obteve autorização do Departamento de Energia para publicar o código fonte do Macsyma DOE sob a Licença Pública GNU e em 2000 iniciou-se o projeto Maxima no sítio SourceForge, para manter e desenvolver o Macsyma DOE, agora chamado Maxima.

Esta tradução para português de Portugal foi feita com base na tradução de Jorge Barros de Abreu para português brasileiro e é mantida por Jaime Villate (`villate arroba fe.up.pt`).

Sumário

Conteúdo

1 Introdução ao Maxima

O Maxima inicia-se executando alguma das suas interfaces gráficas ou numa consola, com o comando "maxima". No início Maxima mostra informação do número de versão e uma linha de entrada de comando, identificada por `(%i1)` (entrada número 1). Cada comando deve terminar-se com ponto e vírgula "`;`", para que seja executado e apareça a resposta, ou com "`$`", para que seja executado sem aparecer a resposta. Para terminar a sessão usa-se o comando "`quit()`;". Normalmente, as teclas *`Ctrl-d`* (Ctrl e d em simultâneo) servem como atalho para terminar a sessão. Eis um exemplo de uma sessão no Maxima:

```
$ maxima
Maxima 5.34.1 http://maxima.sourceforge.net
using Lisp SBCL 1.1.14.debian
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.

(%i1) 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();
$
```

Maxima pode procurar informação nas páginas do manual. Usa-se o comando *`describe`* para mostrar todas as funções e variáveis relacionadas com o termo de pesquisa e opcionalmente a sua documentação. O símbolo de interrogação `?` é uma abreviatura para `describe`:

```
(%i1) ?? integer
 0: askinteger  (Functions and Variables for Facts)
 1: askinteger <1>  (Functions and Variables for Facts)
 2: askinteger <2>  (Functions and Variables for Facts)
 3: askinteger <3>  (Functions and Variables for Facts)
 4: beta_args_sum_to_integer  (Gamma and factorial Functions)
 5: integer  (Functions and Variables for Properties)
 6: integer_partitions  (Functions and Variables for Sets)
 7: integer_partitions <1>  (Functions and Variables for Sets)
 8: integerp  (Functions and Variables for Numbers)
 9: integervalued  (Functions and Variables for Properties)
 10: noninteger  (Functions and Variables for Properties)
 11: nonnegintegerp  (Functions and Variables for Numbers)
```

```
Enter space-separated numbers, 'all' or 'none': 0

 -- 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
```

Para usar um resultado em cálculos posteriores, pode atribuir-se esse valor a uma variável ou usar-se a etiqueta %o*n*, onde *n* é o número da saída. Adicionalmente, % refere-se sempre ao resultado mais recente:

```
(%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 tem conhecimento sobre números complexos e constantes numéricas:

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

Maxima pode fazer contas de cálculo diferencial e integral:

```
(%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 pode resolver sistemas lineares de equações e equações cúbicas:

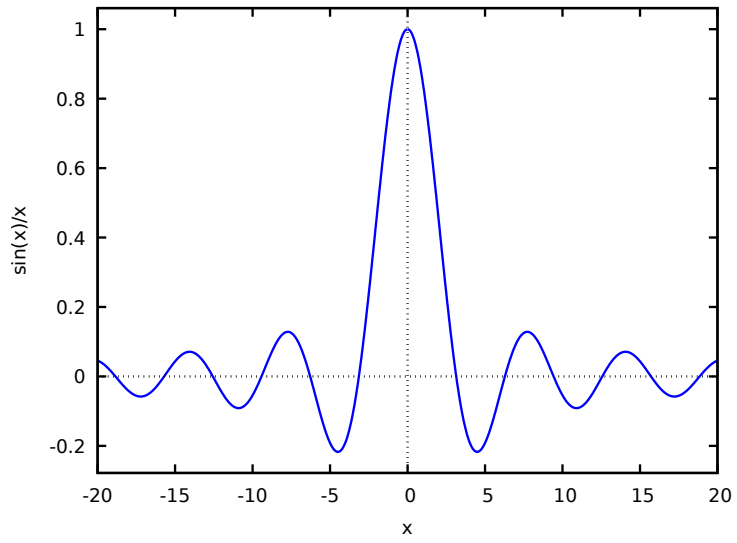
```
(%i1) linsolve ([3*x + 4*y = 7, 2*x + a*y = 13], [x, y]);
                          7 a - 52        25
(%o1)                [x = --------, y = -------]
                          3 a - 8       3 a - 8
(%i2) solve (x^3 - 3*x^2 + 5*x = 15, x);
(%o2)         [x = - sqrt(5) %i, x = sqrt(5) %i, x = 3]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

Maxima pode também resolver sistemas de equações não lineares. Note-se os comandos terminados com *$*, que não mostram o resultado obtido.

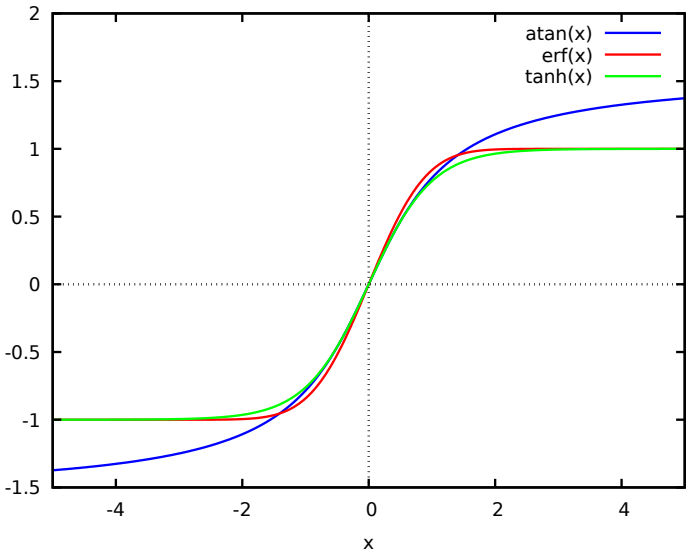
```
(%i1) eq_1: x^2 + 3*x*y + y^2 = 0$
(%i2) eq_2: 3*x + y = 1$
(%i3) solve ([eq_1, eq_2]);
                3 sqrt(5) + 7      sqrt(5) + 3
(%o3) [[y = - -------------, x = -----------],
                     2                2
                                    3 sqrt(5) - 7        sqrt(5) - 3
     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

Maxima pode gerar gráficos de uma ou mais funções:

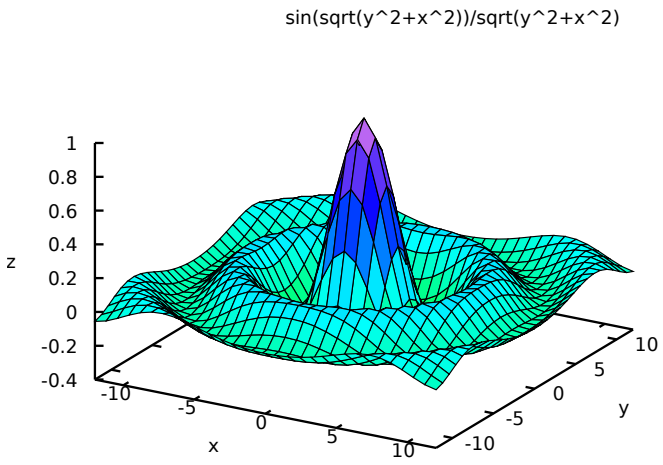
```
(%i1) plot2d (sin(x)/x, [x, -20, 20])$
```




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




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



2 Detecção e Relato de Erros

2.1 Definições para Detecção e Relato de Erros

`run_testsuite` () [Função]
`run_testsuite` (*boolean*) [Função]
`run_testsuite` (*boolean*, *boolean*) [Função]
`run_testsuite` (*boolean*, *boolean*, *list*) [Função]

Executa o conjunto de testes do Maxima. Testes que produzem a resposta desejada são considerados "aprovações" (em inglês, passes) e testes que não produzem a resposta desejada são marcados como erros.

`run_testsuite ()` mostra somente testes que não são aprovados.

`run_testsuite (true)` mostra somente testes que são marcados como erros, bem como as falhas.

`run_testsuite (true, true)` mostra todos os testes.

Se for usado o terceiro argumento opcional, que deve ser uma lista, executam-se unicamente os testes indicados nessa lista. Os nomes de todos os testes é especificado por `testsuite_files`.

`run_testsuite` altera a variável de ambiente Maxima. Tipicamente um script de teste executa `kill` para estabelecer uma variável de ambiente (uma a saber sem funções definidas pelo utilizador e variáveis) e então define funções e variáveis apropriadamente para o teste.

`run_testsuite` retorna `done`.

`testsuite_files` [Variável]

`testsuite_files` é o conjunto de testes a ser executado por `run_testsuite`. É uma lista de nomes de ficheiros contendo os testes a executar. Se alguns dos testes num ficheiro falha de forma conhecida, então em lugar de listar o nome do ficheiro mostra-se uma lista com o nome do ficheiro e o número dos testes que falharam.

Por exemplo, a linha seguinte é uma parte do conjunto de testes padrão:

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

Essa linha especifica o conjunto de testes contidos nos ficheiros "rtest13s" e "rtest14", em que os testes números 57 e 63 do ficheiro "rtest14" falharam de forma conhecida.

`bug_report` () [Função]

Mostra os números de versão do Maxima e do Lisp e o apontador para o sítio onde devem informar-se os erros encontrados no Maxima, para que possam ser solucionados. A informação das versões pode ser consultada também com `build_info`.

Quando se informa sobre um erro, é muito útil indicar a versão do Maxima e do Lisp usadas.

A saída do comando `bug_report` é uma sequência de caracteres vazia "".

`build_info` () [Função]

Mostra os números de versão do Maxima e do Lisp.

A saída do comando `build_info` é uma sequência de caracteres vazia "".

3 Ajuda

3.1 Lisp e Maxima

Maxima é escrito na liguagem de programação Lisp, e é fácil acessar funções Lisp e variáveis a partir do Maxima e vice-versa. Símbolos Lisp e Maxima são distinguidos através de uma convenção de nome. Um símbolo Lisp que começa com um sinal de dólar `$` corresponde a um símbolo Maxima sem o sinal de dólar. Um símbolo Maxima que começa com um ponto de interrogação `?` corresponde a um símbolo Lisp sem o ponto de interrogação. Por exemplo, o símbolo Maxima `foo` corresponde ao símbolo Lisp `$foo`, enquanto o símbolo Maxima `?foo` corresponde ao símbolo Lisp `foo`, Note que `?foo` é escrito sem um espaço entre `?` e `foo`; de outra forma pode ser uma chamada errônea para `describe ("foo")`.

Hífen `-`, asterisco `*`, ou outro caractere especial em símbolos Lisp deve ser precedido por uma barra invertida `\` onde ele aparecer no código Maxima. Por exemplo, o identificador Lisp `*foo-bar*` é escrito `?\*foo\-bar\*` no Maxima.

Código Lisp pode ser executado dentro de uma sessão Maxima. Uma linha simples de Lisp (contendo uma ou mais formas) pode ser executada através do comando especial `:lisp`. Por exemplo,

```
(%i1) :lisp (foo $x $y)
```

chama a função Lisp `foo` com variáveis Maxima `x` e `y` como argumentos. A constução `:lisp` pode aparecer na linha de comando interativa ou em um ficheiro processado por `batch` ou `demo`, mas não em um ficheiro processado por `load`, `batchload`, `translate_file`, ou `compile_file`.

A função `to_lisp()` abre uma sessão interativa Lisp. Digitando `(to-maxima)` fecha a sessão Lisp e retorna para o Maxima.

Funções Lisp e variáveis que são para serem visíveis no Maxima como funções e variáveis com nomes comuns (sem pontuação especial) devem ter nomes Lisp começando com o sinal de dólar `$`.

Maxima é sensível à caixa, distingue entre letras em caixa alta (maiúsculas) e letras em caixa baixa (minúsculas) em identificadores, enquanto Lisp não é sensível à caixa. Existem algumas regras governando a tradução de nomes entre o Lisp e o Maxima.

1. Um identificador Lisp não contido entre barras verticais corresponde a um identificador Maxima em caixa baixa. Se o identificador Lisp estiver em caixa alta, caixa baixa, ou caixa mista, é ignorado. E.g., Lisp `$foo`, `$FOO`, e `$Foo` todos correspondem a Maxima `foo`.
2. Um identificador Lisp que está todo em caixa alta ou todo em caixa baixa e contido em barras verticais corresponde a um identificador Maxima com caixa invertida. Isto é, caixa alta é alterada para caixa baixa e caixa baixa para caixa alta. E.g., Lisp `|$FOO|` e `|$foo|` corresponde a Maxima `foo` e `FOO`, respectivamente.
3. Um identificador Lisp que é misto de caixa alta e caixa baixa e contido entre barras verticais corresponde a um identificador Maxima com o mesma caixa. E.g., Lisp `|$Foo|` corresponde a Maxima `Foo`.

A macro Lisp `#$` permite o uso de expressões Maxima em código Lisp. `#$`*expr*`$` expande para uma expressão Lisp equivalente à expressão Maxima *expr*.

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

Isso tem o mesmo efeito que digitar

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

A função Lisp `displa` imprime uma expressão em formato Maxima.

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

Funções definidas em Maxima não são funções comuns em Lisp. A função Lisp `mfuncall` chama uma função Maxima. Por exemplo:

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

Algumas funções Lisp possuem o mesmo nome que no pacote Maxima, a saber as seguintes.

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

3.2 Descartando

Computação simbólica tende a criar um bom volume de ficheiros temporários, e o efectivo manuseio disso pode ser crucial para sucesso completo de alguns programas.

Sob GCL, nos sistemas UNIX onde a chamada de sistema mprotect (controle de acessso autorizado a uma região de memória) está disponível (incluindo SUN OS 4.0 e algumas variantes de BSD) uma organização de ficheiros temporários estratificada está disponível. Isso limita a organização para páginas que tenham sido recentemente escritas. Veja a documentação da GCL sob ALLOCATE e GBC. No ambiente Lisp fazendo (setq si::*notify-gbc* t) irá ajudá-lo a determinar quais áreas podem precisar de mais espaço.

3.3 Documentação

O manual on-line de utilizador do Maxima pode ser visto em diferentes formas. A partir da linha de comando interativa do Maxima, o manual de utilizador é visto em texto plano através do comando `?` (i.e., a função `describe`). O manual de utilizador é visto como hipertexto `info` através do programa visualizador `info` e como uma web page através de qualquer navegador web comum.

`example` mostra exemplos de muitas funções do Maxima. Por exemplo,

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

retorna

```
(%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
```

e saída adicional.

3.4 Definições para Ajuda

`demo` (*nomeficheiro*) [Função]

Avalia expressões Maxima em *nomeficheiro* e mostra os resultados. `demo` faz uma pausa após avaliar cada expressão e continua após a conclusão com um enter das entradas de utilizador. (Se executando em Xmaxima, `demo` pode precisar ver um ponto e vírgula `;` seguido por um enter.)

`demo` procura na lista de directórios `file_search_demo` para achar `nomeficheiro`. Se o ficheiro tiver o sufixo `dem`, o sufixo pode ser omitido. Veja também `file_search`.

`demo` avalia seus argumento. `demo` retorna o nome do ficheiro de demonstração.

Exemplo:

```
(%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`*) [Função]
`describe` (*`string`*, *exact*) [Função]
`describe` (*`string`*, *inexact*) [Função]

`describe(`*`string`*`)` é equivalente a `describe(`*`string`*`, exact)`.

`describe(`*`string`*`, exact)` encontra um item com título igual (case-insensitive) a *string*, se existir tal item.

`describe(`*`string`*`, inexact)` encontra todos os itens documentados que contiverem *string* em seus títulos. Se existe mais de um de tal item, Maxima solicita ao utilizador seleccionar um item ou ítens para mostrar.

Na linha de comando interativa, `? foo` (com um espaço entre `?` e `foo`) é equivalente a `describe("foo", exact)`. e `?? foo` é equivalente a `describe("foo", inexact)`.

`describe("", inexact)` retorna uma lista de todos os tópicos documentados no manual on-line.

`describe` não avalia seu argumento. `describe` retorna `true` se alguma documentação for encontrada, de outra forma retorna `false`.

Veja também *Documentação*.

Exemplo:

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

Info from file /use/local/maxima/doc/info/maxima.info:
 - Function: integrate (expr, var)
 - Function: integrate (expr, var, a, b)
     Attempts to symbolically compute the integral of 'expr' with
     respect to 'var'.  'integrate (expr, var)' is an indefinite
     integral, while 'integrate (expr, var, a, b)' is a definite
     integral, [...]
```

Nesse , ítens 7 e 8 foram seleccionados. Todos ou nenhum dos ítens poderia ter sido seleccionado através da inserção de `all` ou `none`, que podem ser abreviado para `a` ou para `n`, respectivamente.

`example (`*`tópico`*`)` [Função]
`example ()` [Função]

`example (`*`topic`*`)` mostra alguns exemplos de *tópico*, que é um símbolo (não uma sequência de caracteres). A maioria dos tópicos são nomes de função. `example ()` retorna a lista de todos os tópicos reconhecidos.

O nome do ficheiro contendo os exemplos é dado pela variável global `manual_demo`, cujo valor padrão é `"manual.demo"`.

`example` não avalia seu argumento. `example` retorna `done` a menos que ocorra um erro ou não exista o argumento fornecido pelo utilizador, nesse caso `example` retorna uma lista de todos os tópicos reconhecidos.

Exemplos:

```
(%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 Linha de Comandos

4.1 Introdução a Linha de Comandos

’ [Operador]

O operador apóstrofo ’ evita avaliação.

Aplicado a um símbolo, o apóstrofo evita avaliação do símbolo.

Aplicado a uma chamada de função, o apóstrofo evita avaliação da chamada de função, embora os argumentos da função sejam ainda avaliados (se a avaliação não for de outra forma evitada). O resultado é a forma substantiva da chamada de função.

Aplicada a uma espressão com parêntesis, o apóstrofo evita avaliação de todos os símbolos e chamadas de função na expressão. E.g., `’(f(x))` significa não avalie a expressão `f(x)`. `’f(x)` (com apóstrofo aplicado a `f` em lugar de `f(x)`) retorna a forma substantiva de `f` aplicada a `[x]`.

O apóstrofo nao evita simplificação.

Quando o sinalizador global `noundisp` for `true`, substantivos são mostrados com um apóstrofo. Esse comutador é sempre `true` quando mostrando definições de funções.

Veja também operador apóstrofo-apóstrofo `’’` e `nouns`.

Exemplos:

Aplicado a um símbolo, o apóstrofo evita avaliação do símbolo.

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

Aplicado a uma chamada de função, o apóstrofo evita avaliação da chamada de função. O resultado é a forma substantiva da chamada de função.

```
(%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
```

Aplicado a uma expressão com parêntesis, o apóstrofo evita avaliação de todos os símbolos e chamadas de função na expressão.

```
(%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
```

O apóstrofo não evita simplificação.

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

'' [Operador]

O operador apóstrofo-apóstrofo '' (dois apóstrofost) modifica avaliação em expressões de entrada.

Aplicado a uma expressão geral *expr*, apóstrofo-apóstrofo faz com que o valor de *expr* seja substituído por *expr* na expressão de entrada.

Aplicado ao operadro de uma expressão, apóstrofo-apóstrofo modifica o operadro de um susbstantivo para um verbo (se esse operador não for já um verbo).

O operador apóstrofo-apóstrofo é aplicado através do passador de entrada; o apóstrofo-apóstrofo não é armazenado como parte de uma expressão de entrada passada. O operador apóstrofo-apóstrofo é sempre aplicado tão rapidamente quanto for passado, e não pode receber um terceiro apóstrofo. Dessa forma faz com que ocorra avaliação quando essa avaliação for de outra forma suprimida, da mesma forma que em definições de função, definições de expressãoes lambda, e expressões que recebem um apóstrofo simples '.

Apóstrofo-apóstrofo é reconhecido por `batch` e `load`.

Veja também o operador apóstrofo ' e `nouns`.

Exemplos:

Aplicado a uma expressão geral *expr*, apóstrofo-apóstrofo fazem com que o valor de *expr* seja substituido por *expr* na expressão de entrada.

```
(%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
```

Aplicado ao operador de uma expressão, apóstrofo-apóstrofo muda o operadro de um substantivo para um verbo (se esse operadro não for já um verbo).

```
(%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
```

O operador apóstrofo-apóstrofo é aplicado por meio de um passador de entrada; operador-apóstrofo não é armazenado como parte da expressão de entrada.

```
(%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
```

Apóstrofo apóstrofo faz com que ocorra avaliação quando a avaliação tiver sido de outra forma suprimida, da mesma forma que em definições de função, da mesma forma que em definições de função lambda expressions, E expressões que recebem o apóstrofo simples '.

```
(%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 Definições para Linha de Comandos

`alias` (*new_name_1*, *old_name_1*, ..., *new_name_n*, *old_name_n*) [Função]
provê um nome alternativo para uma função (de utilizador ou de sistema), variável, array, etc. Qualquer número de argumentos pode ser usado.

`debugmode` [Variável de opção]
Valor por omissão: `false`

Quando um erro do Maxima ocorre, Maxima iniciará o depurador se `debugmode` for `true`. O utilizador pode informar comandos para examinar o histórico de chamadas, marcar pontos de parada, percorrer uma linha por vez o código do Maxima, e assim por diante. Veja `debugging` para uma lista de opções do depurador.

Habilitando `debugmode` por meio da alteração de seu valor para `true`, não serão capturados erros do Lisp.

`ev` (*expr*, *arg_1*, ..., *arg_n*) [Função]
Avalia a expressão *expr* no ambiente especificado pelos argumentos *arg_1*, ..., *arg_n*. Os argumentos são comutadores (sinalizadores Booleanos), atribuições, equações, e funções. `ev` retorna o resultado (outra expressão) da avaliação.

A avaliação é realizada em passos, como segue.

1. Primeiro o ambiente é preparado examinando os argumentos que podem ser quaisquer ou todos os seguintes.
   - `simp` faz com que *expr* seja simplificado independentemente da posição do comutador `simp` que inibe simplificação se `false`.
   - `noeval` suprime a fase de avaliação de `ev` (veja passo (4) adiante). Isso é útil juntamente com outros comutadores e faz com que *expr* seja simplificado novamente sem ser reavaliado.
   - `nouns` causa a avaliação de formas substantivas (tipicamente funções não avaliadas tais como `'integrate` ou `'diff`) em *expr*.
   - `expand` causa expansão.

- `expand (m, n)` causa expansão, alterando os valores de `maxposex` e `maxnegex` para *m* e *n* respectivamente.
- `detout` faz com que qualquer matriz inversa calculada em *expr* tenha seu determinante mantido fora da inversa ao invés de dividindo a cada elemento.
- `diff` faz com que todas as diferenciações indicadas em *expr* sejam executadas.
- `derivlist (x, y, z, ...)` causa somente diferenciações referentes às variáveis indicadas.
- `float` faz com que números racionais não inteiros sejam convertidos para ponto flutuante.
- `numer` faz com que algumas funções matemáticas (incluindo a exponenciação) com argumentos sejam valiadas em ponto flutuante. Isso faz com que variávels em *expr* que tenham sido dados numervals (valores numéricos) sejam substituídas por seus valores. Isso também modifica o comutador `float` para activado.
- `pred` faz com que predicados (expressões que podem ser avaliados em `true` ou `false`) sejam avaliadas.
- `eval` faz com que uma avaliação posterior de *expr* ocorra. (Veja passo (5) adiante.) `eval` pode ocorrer múltiplas vezes. Para cada instância de `eval`, a expressão é avaliada novamente.
- `A` onde `A` é um átomo declarado seja um sinalizador de avaliação (veja `evflag`) faz com que `A` seja associado a `true` durante a avaliação de *expr*.
- `V: expresão` (ou alternativamente `V=expressão`) faz com que `V` seja associado ao valor de `expressão` durante a avaliação de *expr*. Note que se `V` é uma opção do Maxima, então `expression` é usada para seu valor durante a avaliação de *expr*. Se mais que um argumento para `ev` é desse tipo então a associação termina em paralelo. Se `V` é uma expressão não atômica então a substituição, ao invés de uma associação, é executada.
- `F` onde `F`, um nome de função, tenha sido declarado para ser uma função de avaliação (veja `evfun`) faz com que `F` seja aplicado a *expr*.
- Qualquer outro nome de função (e.g., `sum`) causa a avaliação de ocorrências desses nomes em *expr* mesmo que eles tenham sido verbos.
- De forma adicional uma função ocorrendo em *expr* (digamos `F(x)`) pode ser definida localmente para o propósito dessa avaliação de *expr* dando `F(x) := expressão` como um argumento para `ev`.
- Se um átomo não mensionado acima ou uma variável subscrita ou expressão subscrita for dada como um argumento, isso é avaliado e se o resultado for uma equação ou uma atribuição então a associação indicada ou substituição é executada. Se o resultado for uma lista então os membros da lista serão tratados como se eles fossem argumentos adicionais dados para `ev`. Isso permite que uma lista de equações seja dada (e.g. `[X=1, Y=A**2]`) ou que seja dado uma lista de nomes de equações (e.g., `[%t1, %t2]` onde `%t1` e `%t2` são equações) tais como aquelas listas retornadas por `solve`.

Os argumentos de `ev` podem ser dados em qualquer ordem com exceção de substituições de equações que são manuseadas em sequência, da esquerda para a direita, e funções de avaliação que são compostas, e.g., `ev (`*`expr`*`, ratsimp, realpart)` são manuseadas como `realpart (ratsimp (`*`expr`*`))`.

Os comutadores `simp`, `numer`, `float`, e `pred` podem também ser alterados localmente em um bloco, ou globalmente no Maxima dessa forma eles irã permanecer em efeito até serem resetados ao término da execução do bloco.

Se *expr* for uma expressão racional canónica (CRE), então a expressão retornada por `ev` é também uma CRE, contanto que os comutadores `numer` e `float` não sejam ambos `true`.

2. Durante o passo (1), é feito uma lista de variáveis não subscritas aparecendo do lado esquerdo das equações nos argumentos ou nos valores de alguns argumentos se o valor for uma equação. As variáveis (variáveis subscritas que não possuem funções array associadas bem como variáveis não subscritas) na expressão *expr* são substituídas por seus valores globais, excepto para esse aparecendo nessa lista. Usualmente, *expr* é apenas um rótulo ou `%` (como em `%i2` no exemplo adiante), então esse passo simplesmente repete a expressão nomeada pelo rótulo, de modo que `ev` possa trabalhar sobre isso.
3. Se quaisquer substituições tiveem sido indicadas pelos argumentos, elas serão realizadas agora.
4. A expressão resultante é então reavaliada (a menos que um dos argumentos seja `noeval`) e simplificada conforme os argumentos. Note que qualquer chamada de função em *expr* será completada depois das variáveis nela serem avalidas e que `ev(F(x))` dessa forma possa comportar-se como `F(ev(x))`.
5. Para cada instância de `eval` nos argumentos, os passos (3) e (4) são repetidos.

Exemplos

```
(%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
```

Uma sintaxe alternativa de alto nível tem sido provida por `ev`, por meio da qual se pode apenas digitar seus argumentos, sem o `ev()`. Isto é, se pode escrever simplesmente

`expr, arg_1, ..., arg_n`

Isso não é permitido como parte de outra expressão, e.g., em funções, blocos, etc.

Observe o processo de associação paralela no seguinte exemplo.

```
(%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` [Propriedade]

Quando um símbolo x tem a propriedade `evflag`, as expressões `ev(`*`expr`*`, `*`x`*`)` e *`expr`*`, `*`x`* (na linha de comando interativa) são equivalentes a `ev(`*`expr`*`, `*`x`*` = true)`. Isto é, x está associada a `true` enquanto *expr* for avaliada.

A expressão `declare(x, evflag)` fornece a propriedade `evflag` para a variável x.

Os sinalizadores que possuem a propriedade `evflag` por padrão são os seguintes: `algebraic`, `cauchysum`, `demoivre`, `dotscrules`, `%emode`, `%enumer`, `exponentialize`, `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`, e `trigexpand`.

Exemplos:

```
(%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 SIM else NÃO;
(%o11)                          NÃO
(%i12) if aa = true then SIM else NÃO, aa;
(%o12)                          SIM
```

`evfun` [Propriedade]

Quando uma função F tem a propriedade `evfun`, as expressões `ev(`*`expr`*`, `*`F`*`)` e *`expr`*`, `*`F`* (na linha de comando interativa) são equivalentes a *`F`*`(ev(`*`expr`*`))`.

Se duas ou mais funções F, G, etc., que possuem a propriedade `evfun` forem especificadas, as funções serão aplicadas na ordem em que forem especificadas.

A expressão `declare(`*`F`*`, evfun)` fornece a propriedade `evfun` para a função F.

As funções que possuem a propriedade `evfun` por padrão são as seguintes: `bfloat`, `factor`, `fullratsimp`, `logcontract`, `polarform`, `radcan`, `ratexpand`, `ratsimp`, `rectform`, `rootscontract`, `trigexpand`, e `trigreduce`.

Exemplos:

```
(%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` [Variável de opção]

Habilita o modo "avaliação infinita". `ev` repetidamente avalia uma expressão até que ela permaneça invariante. Para prevenir uma variável, digamos `X`, seja demoradamente avaliada nesso modo, simplesmente inclua `X='X` como um argumento para `ev`. Certamente expressões tais como `ev (X, X=X+1, infeval)` irão gerar um ciclo infinito.

`kill` (`a_`*1*, ..., `a_`*n*) [Função]

`kill` (*labels*) [Função]

`kill` (*inlabels, outlabels, linelabels*) [Função]

`kill` (*n*) [Função]
`kill` ([*m*, *n*]) [Função]
`kill` (*values, functions, arrays, ...*) [Função]
`kill` (*all*) [Função]
`kill` (*allbut* (`a_1`, ..., `a_n`)) [Função]

Remove todas as associações (valor, funções, array, ou regra) dos argumentos *a_1*, ..., *a_n*. Um argumento *a_k* pode ser um símbolo ou um elemento de array simples. Quando *a_k* for um elemento de array simples, `kill` remove a associação daquele elemento sem afectar qualquer outro elemento do array.

Muitos argumentos especiais são reconhecidos. Diferentes famílias de argumentos podem ser combinadas, e.g., `kill (inlabels, functions, allbut (foo, bar))`

todos os rótulos de entrada, de saída, e de expressões intermédias criados até então. `kill (inlabels)` libera somente rótudos de entrada que começam com o valor corrente de `inchar`. De forma semelhante, `kill (outlabels)` libera somente rótulos de saída que começam com o valor corrente de `outchar`, e `kill (linelabels)` libera somente rótulos de expressões intermédias que começam com o valor corrente de `linechar`.

`kill (n)`, onde *n* é um inteiro, libera os *n* mais recentes rótulos de entrada e saída.

`kill ([m, n])` libera rótulos de entrada e saída de *m* até *n*.

`kill (infolist)`, onde *infolist* é um item em `infolists` (tais como `values`, `functions`, ou `arrays`) libera todos os ítens em *infolist*. Veja também `infolists`.

`kill (all)` liberar todos os ítens em todas as infolists. `kill (all)` não retorna variáveis globais para seus valores padrões; Veja `reset` sobre esse ponto.

`kill (allbut (a_1, ..., a_n))` remove a associação de todos os itens sobre todas as infolistas excepto para *a_1*, ..., *a_n*. `kill (allbut (infolist))` libera todos os ítens excepto para si próprio em *infolist*, onde *infolist* é `values`, `functions`, `arrays`, etc.

A memória usada por uma propriedade de associação não será liberada até que todos os símbolos sejam liberados disso. Em particular, para liberar a memória usada pelo valor de um símbolo, deve-se liberar o rótulo de saída que mosta o valor associado, bem como liberando o próprio símbolo.

`kill` coloca um apóstro em seus argumentos (não os avalia). O operador apóstrofo-apóstrofo, `''`, faz com que ocorra avaliação.

`kill (símbolo)` libera todas as propriedades de *símbolo*. Em oposição, `remvalue`, `remfunction`, `remarray`, e `remrule` liberam uma propriedade específica.

`kill` sempre retorna `done`, igualmente se um argumento não tem associações.

`labels` (*símbolo*) [Função]
`labels` [Variável de sistema]

Retorna a lista de rótulos de entradas, de saída, de expressões intermédias que começam com *símbolo*. Tipicamente *símbolo* é o valor de `inchar`, `outchar`, ou `linechar`. O caracter rótulo pode ser dado com ou sem o sinal de porcentagem, então, por exemplo, `i` e `%i` retornam o mesmo resultado.

Se nenhum rótulo começa com *símbolo*, `labels` retorna uma lista vazia.

A função `labels` não avalia seu argumento. O operador apóstrofo-apóstrofo `''` faz com que ocorra avaliação. Por exemplo, `labels (''inchar)` retorna os rótulos de entrada que começam com o caractere corrente do rótulo de entrada.

A variável `labels` é uma lista de rótulos de entrada, saída, e de expressões intermédias, incluindo todos os rótulos anteriores se `inchar`, `outchar`, ou `linechar` que tiverem sido redefinidos.

Por padrão, Maxima mostra o resultado de cada expressão de entrada do utilizador, dando ao resultado um rótulo de saída. A exibição da saída é suprimida pelo encerramento da entrada com `$` (sinal de dolar) em lugar de `;` (ponto e vírgula). Um rótulo de saída é construido e associado ao resultado, mas não é mostrado, e o rótulo pode ser referenciado da mesma forma que rótulos de saída mostrados. Veja também `%`, `%%`, e `%th`.

Rótulos de expressões intermédias podem ser gerados por algumas funções. O sinalizador `programmode` controla se `solve` e algumas outras funções geram rótulos de expressões intermédias em lugar de retornar uma lista de expressões. Algumas outras funções, tais como `ldisplay`, sempre geram rótulos de expressões intermédias.

Veja também `inchar`, `outchar`, `linechar`, e `infolists`.

`linenum` [Variável de sistema]

Retorna o número da linha do par corrente de expressões de entrada e saída.

`myoptions` [Variável de sistema]

Valor por omissão: `[]`

`myoptions` é a lista de todas as opções alguma vez alteradas pelo utilizador, tenha ou não ele retornado a alteração para o seu valor padrão.

`nolabels` [Variável de opção]

Valor por omissão: `false`

Quando `nolabels` for `true`, rótulos de entrada e saída (`%i` e `%o`, respectivamente) são mostrados, mas os rótulos não são associados aos resultados, e os rótulos não são anexados ao final da lista `labels`. Uma vez que rótulos não são associados aos resultados, a reciclagem pode recuperar a memória tomada pelos resultados.

De outra forma rótulos de entrada e saída são associados aos resultados, e os rótulos são anexados ao final da lista `labels`.

Veja também `batch`, `batchload`, e `labels`.

`optionset` [Variável de opção]

Valor por omissão: `false`

Quando `optionset` for `true`, Maxima mostrará uma mensagem sempre que uma opção do Maxima for alterada. Isso é útil se o utilizador está incerto sobre a ortografia de alguma opção e quer ter certeza que a variável por ele atribuído um valor foi realmente uma variável de opção.

`playback ()` [Função]
`playback (`*n*`)` [Função]
`playback ([`*m*`,` *n*`])` [Função]

`playback` ([*m*]) [Função]
`playback` (*input*) [Função]
`playback` (*slow*) [Função]
`playback` (*time*) [Função]
`playback` (*grind*) [Função]

Mostra expressões de entrada, de saída, e expressões intermédias, sem refazer os cálculos. `playback` somente mostra as expressões associadas a rótulos; qualquer outra saída (tais como textos impressos por `print` ou `describe`, ou messagens de erro) não é mostrada. Veja também `labels`.

`playback` não avalia seus argumentos. O operador apóstrofo-apóstrofo, `''`, sobrepõe-se às aspas. `playback` sempre retorna `done`.

`playback ()` (sem argumentos) mostra todas as entradas, saídas e expressões intermédias geradas até então. Uma expressão de saída é mostrada mesmo se for suprimida pelo terminador `$` quando ela tiver sido originalmente calculada.

`playback (`*n*`)` mostra as mais recentes *n* expressões. Cada entrada, saída e expressão intermédia conta como um.

`playback ([`*m*`, `*n*`])` mostra entradas, saídas e expressões intermédias com os números de *m* até *n*, inclusive.

`playback ([`*m*`])` é equivalente a `playback ([`*m*`, `*m*`])`; isso usualmente imprime um par de expressões de entrada e saída.

`playback (input)` mostra todas as expressões de entrada geradas até então.

`playback (slow)` insere pausas entre expressões e espera que o utilizador pressione `enter`. Esse comportamento é similar a `demo`. `playback (slow)` é útil juntamente com `save` ou `stringout` quando criamos um ficheiro secundário de armazenagem com a finalidade de capturar expressões úteis.

`playback (time)` mostra o tempo de computação de cada expressão.

`playback (grind)` mostra expressões de entrada no mesmo formato da função `grind`. Expressões de saída não são afectadas pela opção `grind`. Veja `grind`.

Argumentos podem ser combinados, e.g., `playback ([5, 10], grind, time, slow)`.

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

Mostra a propriedade como o indicador *i* associada com o átomo *a*. *a* pode também ser uma lista de átomos ou o átomo `all` nesse caso todos os átomos com a propriedade dada serão usados. Por exemplo, `printprops ([f, g], atvalue)`. `printprops` é para propriedades que não podem ser mostradas de outra forma, i.e. para `atvalue`, `atomgrad`, `gradef`, e `matchdeclare`.

`prompt` [Variável de opção]

Valor por omissão: `_`

`prompt` é o símbolo de linha de comando da função `demo`, modo `playback (slow)`, e da interrupção de ciclos do Maxima (como invocado por `break`).

`quit` () [Função]

Encerra a sessão do Maxima. Note que a função pode ser invocada como `quit();` ou `quit()$`, não por sí mesma `quit`.

Para parar um cálculo muito longo, digite `control-C`. A ação padrão é retornar à linha de comando do Maxima. Se `*debugger-hook*` é `nil`, `control-C` abre o depurador Lisp. Veja também `debugging`.

`remfunction` (*f_1*, ..., *f_n*) [Função]
`remfunction` (*all*) [Função]

Desassocia as definições de função dos síbolos *f_1*, ..., *f_n*. Os argumentos podem ser os nomes de funções comuns (criadas por meio de `:=` ou `define`) ou funções macro (criadas por meio de `::=`).

`remfunction (all)` desassocia todas as definições de funcção.

`remfunction` coloca um ap'ostrofo em seus argumentos (não os avalia).

`remfunction` retorna uma lista de símbolos para a qual a definição de função foi desassociada. `false` é retornado em lugar de qualquer símbolo para o qual não exista definição de função.

`reset` () [Função]

Retorna muitas variáveis globais e opções, e algumas outras variáveis, para seus valores padrões.

`reset` processa as variáveis na lista Lisp `*variable-initial-values*`. A macro Lisp `defmvar` coloca variáveis nessa lista (entre outras ações). Muitas, mas não todas, variáveis globais e opções são definidas por `defmvar`, e algumas variáveis definidas por `defmvar` não são variáveis globais ou variáveis de opção.

`showtime` [Variável de opção]

Valor por omissão: `false`

Quando `showtime` for `true`, o tempo de computação e o tempo decorrido são impressos na tela com cada expressão de saída.

O tempo de cálculo é sempre gravado, então `time` e `playback` podem mostrar o tempo de cálculo mesmo quando `showtime` for `false`.

Veja também `timer`.

`sstatus` (*recurso*, *pacote*) [Função]

Altera o status de *recurso* em *pacote*. Após `sstatus (recurso, pacote)` ser executado, `status (recurso, pacote)` retorna `true`. Isso pode ser útil para quem escreve pacotes, para manter um registro de quais recursos os pacotes usam.

`to_lisp` () [Função]

Insere o sistema Lisp dentro do Maxima. `(to-maxima)` retorna para o Maxima.

`values` [Variável de sistema]

Valor inicial: `[]`

`values` é uma lista de todas as varáveis de utilizador associadas (não opções Maxima ou comutadores). A lista compreende símbolos associados por `:` , `::`, ou `:=`.

5 Operadores

5.1 N-Argumentos

Um operador `nary` é usado para denotar uma função com qualquer número de argumentos, cada um dos quais é separado por uma ocorrência do operador, e.g. A+B ou A+B+C. A função `nary("x")` é uma função de extensão sintática para declarar x como sendo um operador `nary`. Funções podem ser declaradas para serem `nary`. Se `declare(j,nary);` é concluída, diz ao simplicador para simplificar, e.g. `j(j(a,b),j(c,d))` para `j(a, b, c, d)`.

Veja também `syntax`.

5.2 Operador não fixado

Operadores `nofix` são usados para denotar funções sem argumentos. A mera presença de tal operador em um comando fará com que a função correspondente seja avaliada. Por exemplo, quando se digita "exit;" para sair de uma parada do Maxima, "exit" tem comportamento similar a um operador `nofix`. A função `nofix("x")` é uma função de extensão sintática que declara x como sendo um operador `nofix`.

Veja também `syntax`.

5.3 Operador Pósfixado

Operadores `postfix` como a variedade `prefix` denotam funções de um argumento simples, mas nesse caso o argumento sucede imediatamente uma ocorrência do operador na sequência de caracteres de entrada, e.g. 3! . Uma função `postfix("x")` é uma função de extensão sintática que declara x como sendo um operador `postfix`.

Veja também `syntax`.

5.4 Operador Préfixado

Um operador `prefix` é um que significa uma função de um argumento, o qual imediatamente segue uma ocorrência do operador. `prefix("x")` é uma função de extensão sintática que declara x como sendo um operador `prefix`.

Veja também `syntax`.

5.5 Operadores Aritméticos

`+` [Operador]
`-` [Operador]
`*` [Operador]
`/` [Operador]
`^` [Operador]

Os símbolos `+ * /` e `^` representam adição, multiplicação, divisão, e exponenciação, respectivamente. O nome desses operadores são `"+" "*" "/"` e `"^"`, os quais podem aparecer em lugares onde o nome da função ou operador é requerido.

Os símbolos `+` e `-` representam a adição unária e a negação unária, respectivamente, e os nomes desses operadores são `"+"` e `"-"`, respectivamente.

A subtração `a - b` é representada dentro do Maxima como a adição, `a + (- b)`. Expressões tais como `a + (- b)` são mostradas como subtração. Maxima reconhece `"-"` somente como o nome do operador unário de negação, e não como o nome do operador binário de subração.

A divisão `a / b` é representada dentro do Maxima como multiplicação, `a * b^(- 1)`. Expressões tais como `a * b^(- 1)` são mostradas como divisão. Maxima reconhece `"/"` como o nome do operador de divisão.

A adição e a multiplicação são operadores enários e comutativos. a divisão e a exponenciação são operadores binários e não comutativos.

Maxima ordena os operandos de operadores não comutativos para construir uma representação canónica. Para armazenamento interno, a ordem é determinada por `orderlessp`. Para mostrar na tela, a ordem para adição é determinada por `ordergreatp`, e para a multiplicação, a ordem é a mesma da ordenação para armazenamento interno.

Computações aritiméticas são realizadas sobre números literais (inteiro, racionais, números comuns em ponto flutuante, e grandes números em ponto flutuante de dupla precisão). Execto a exponenciação, todas as operações aritméticas sobre números são simplificadas para números. A exponenciação é simplificada para um número se ou o operando é um número comum em ponto flutuante ou um grande número em ponto flutuante de dupla precisão ou se o resultado for um inteiro exato ou um racional exato; de outra forma uma exponenciação pode ser simplificada para `sqrt` ou outra exponenciação ou permanecer inalterada.

A propagação de números em ponto flutuante aplica-se a computações aritiméticas: Se qualquer operando for um grande número em ponto flutuante, o resultado é um grande número em ponto flutuante; de outra forma, se qualquer operando for um número em ponto flutuante comum, o resultado é um número comum em ponto flutuante; de outra forma, se os operandos forem racioanis ou inteiros e o resultado será um racional ou inteiro.

Computaçãoes aritiméticas são uma simplificação, não uma avaliação. Dessa forma a aritmética é realizada em expressões com apóstrofo (mas simplificadas).

Operações aritméticas são aplicadas elemento-por-elemento para listas quando a variável global `listarith` for `true`, e sempre aplicada elemento-por-elemento para matrizes. Quando um operando for uma lista ou uma matriz e outro for um operando de algum outro tipo, o outro operando é combinado com cada um dos elementos da lista ou matriz.

Exemplos:

Adição e multiplicação são opeadores enários comutativos. Maxima ordena os operandos para construir uma representação canónica. Os nomes desses operadores são `"+"` e `"*"`.

```
(%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
```

Divisão e exponenciação são operadores binários e não comutativos. Os nomes desses operadores são `"/"` e `"^"`.

```
(%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
```

Subtração e divisão são representados internamente em termos de adição e multiplicação, respectivamente.

```
(%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
```

Cálculos são realizados sobre números lterais. A propagação de números em poto flutuante aplica-se.

```
(%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]
```

Computações aritméticas são uma simplificação, não uma avaliação.

```
(%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
```

A aritmética é realizada elemento-por-elemento para listas lists (dependendo de `listarith`) e dependendo de matrizes.

```
(%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 ]
```

**** [Operador]

Operador de exponenciação. Maxima reconhece `**` como o mesmo operador que `^` em entrada, e `**` é mostrado como `^` em saída unidimensional, ou colocando o expoente como sobrescrito em saída bidimensional.

A função `fortran` mostra o operador de exponenciação com como `**`, independente de a entrada ter sido na forma `**` ou a forma `^`.

Exemplos:

```
(%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 Operadores Relacionais

`<` [Operador]

`<=` [Operador]

`>=` [Operador]

`>` [Operador]

5.7 Operadores Geral

`^^` [Operador]

`!` [Operador]

O operador factorial. Para qualquer número complexo `x` (incluíndo números inteiros, racionais, e reais) excepto para inteiros negativos, `x!` é definido como `gamma(x+1)`.

Para um inteiro `x`, `x!` simplifica para o produto de inteiros de 1 a `x` inclusive. `0!` simplifica para 1. Para um número em ponto flutuante `x`, `x!` simplifica para o valor de `gamma (x+1)`. Para `x` igual a `n/2` onde `n` é um inteiro ímpar, `x!` simplifica para um factor racional vezes `sqrt (%pi)` (uma vez que `gamma (1/2)` é igual a `sqrt (%pi)`). Se `x` for qualquer outra coisa, `x!` não é simplificado.

As variáveis `factlim`, `minfactorial`, e `factcomb` controlam a simplificação de expressões contendo factoriais.

As funções `gamma`, `bffac`, e `cbffac` são variedades da função `gamma`. `makegamma` substitui `gamma` para funções relacionadas a factoriais.

Veja também `binomial`.

O factorial de um inteiro, inteiro dividido por dois, ou argumento em ponto flutuante é simplificado a menos que o operando seja maior que `factlim`.

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

O factorial de um número complexo, constante conhecida, ou expressão geral não é simplificado. Ainda assim pode ser possível simplificar o factorial após avaliar o operando.

```
(%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]
```

O factorial de um símbolo não associado não é simplificado.

```
(%i1) kill (foo);
```

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

Factoriais são simplificados, não avaliados. Dessa forma `x!` pode ser substituído mesmo em uma expressão com apóstrofo.

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

`!!` [Operador]

O operador de duplo factorial.

Para um número inteiro, número em ponto flutuante, ou número racional `n`, `n!!` avalia para o produto `n (n-2) (n-4) (n-6) ... (n - 2 (k-1))` onde `k` é igual a `entier (n/2)`, que é, o maior inteiro menor que ou igual a `n/2`. Note que essa definição não coincide com outras definições publicadas para argumentos que não são inteiros.

Para um inteiro par (ou ímpar) `n`, `n!!` avalia para o produto de todos os inteiros consecutivos pares (ou ímpares) de 2 (ou 1) até `n` inclusive.

Para um argumento `n` que não é um número inteiro, um número em ponto flutuante, ou um número racional, `n!!` retorna uma forma substantiva `genfact (n, n/2, 2)`.

`#` [Operador]

Representa a negação da igualdade sintática `=`.

Note que pelo facto de as regras de avaliação de expressões predicadas (em particular pelo facto de `not` *`expr`* fazer com que ocorra a avaliação de *expr*), a forma `not a = b` não é equivalente à forma `a # b` em alguns casos.

Note que devido às regras para avaliação de expressões predicadas (em particular devido a `not` *`expr`* fazer com que a avaliação de *expr* ocorra), `not a = b` é equivalente a `is(a # b)`, em lugar de ser equivalente a `a # b`.

Exemplos:

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

`.` [Operador]

O operador ponto, para multiplicação (não comutativa) de matrizes. Quando "." é usado com essa finalidade, espaços devem ser colocados em ambos os lados desse

operador, e.g. A . B. Isso distingue o operador ponto plenamente de um ponto decimal em um número em ponto flutuante.

Veja também `dot`, `dot0nscsimp`, `dot0simp`, `dot1simp`, `dotassoc`, `dotconstrules`, `dotdistrib`, `dotexptsimp`, `dotident`, e `dotscrules`.

`:` [Operador]

O operador de atribuição. E.g. A:3 escolhe a variável A para 3.

`::` [Operador]

Operador de atribuição. :: atribui o valor da expressão em seu lado direito para o valor da quantidade na sua esquerda, que pode avaliar para uma variável atômica ou variável subscrita.

`::=` [Operador]

Operador de definição de função de macro. `::=` define uma função (chamada uma "macro" por razões históricas) que coloca um apóstrofo em seus argumentos (evitando avaliação), e a expressão que é retornada (chamada a "expansão de macro") é avaliada no contexto a partir do qual a macro foi chamada. Uma função de macro é de outra forma o mesmo que uma função comum.

`macroexpand` retorna uma expansão de macro (sem avaliar a expansão). `macroexpand (foo (x))` seguida por `''%` é equivalente a `foo (x)` quando `foo` for uma função de macro.

`::=` coloca o nome da nova função de macro dentro da lista global `macros`. `kill`, `remove`, e `remfunction` desassocia definições de função de macro e remove nomes de `macros`.

`fundef` e `dispfun` retornam respectivamente uma definição de função de macro e uma atribuição dessa definição a um rótulo, respectivamente.

Funções de macro comumente possuem expressões `buildq` e `splice` para construir uma expressão, que é então avaliada.

Exemplos

Uma função de macro coloca um apóstrofo em seus argumentos evitando então a avaliação, então mensagem (1) mostra `y - z`, não o valor de `y - z`. A expansão de macro (a expressão com apóstrofo `'(print ("(2) x is equal to", x))` é avaliada no contexto a partir do qual a macro for chamada, mostrando a mensagem (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 é igual a", x), '(print ("(2) x é igua
(%o4) printq1(x) ::= block(print("(1) x é igual a", x),
                                '(print("(2) x é igual a", x)))
(%i5) printq1 (y - z);
(1) x é igual a y - z
(2) x é igual a %pi
```

```
(%o5)                              %pi
```

Uma função comum avalia seus argumentos, então message (1) mostra o valor de `y - z`. O valor de retorno não é avaliado, então mensagem (2) não é mostrada até a avaliação explícita `''%`.

```
(%i1) x: %pi;
(%o1)                              %pi
(%i2) y: 1234;
(%o2)                             1234
(%i3) z: 1729 * w;
(%o3)                           1729 w
(%i4) printe1 (x) := block (print ("(1) x é igual a", x), '(print ("(2) x é igual
(%o4) printe1(x) := block(print("(1) x é igual a", x),
                                  '(print("(2) x é igual a", x)))
(%i5) printe1 (y - z);
(1) x é igual a 1234 - 1729 w
(%o5)               print((2) x é igual a, x)
(%i6) ''%;
(2) x é igual a %pi
(%o6)                              %pi
```

`macroexpand` retorna uma expansão de macro. `macroexpand (foo (x))` seguido por `''%` é equivalente a `foo (x)` quando `foo` for uma função de macro.

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

`:=` [Operador]

O operador de definição de função. E.g. `f(x):=sin(x)` define uma função `f`.

`=` [Operador]

O operador de equação.

Uma expressão `a = b`, por si mesma, representa uma equação não avaliada, a qual pode ou não se manter. Equações não avaliadas podem aparecer como argumentos para `solve` e `algsys` ou algumas outras funções.

A função `is` avalia `=` para um valor Booleano. `is(a = b)` avalia `a = b` para `true` quando *a* e *b* forem idênticos. Isto é, *a* e *b* forem átomos que são idênticos, ou se eles não forem átomos e seus operadores forem idênticos e seus argumentos forem idênticos. De outra forma, `is(a = b)` avalia para `false`; `is(a = b)` nunca avalia para `unknown`. Quando `is(a = b)` for `true`, *a* e *b* são ditos para serem sintaticamente iguais, em contraste para serem expressões equivalentes, para as quais `is(equal(a, b))` é `true`. Expressões podem ser equivalentes e não sintáticamente iguais.

A negação de `=` é representada por `#`. Da mesma forma que com `=`, uma expressão `a # b`, por si mesma, não é avaliada. `is(a # b)` avalia `a # b` para `true` ou `false`.

Complementando a função `is`, alguns outros operadores avaliam `=` e `#` para `true` ou `false`, a saber `if`, `and`, `or`, e `not`.

Note que pelo facto de as regras de avaliação de expressões predicadas (em particular pelo facto de `not` *expr* fazer com que ocorra a avaliação de *expr*), a forma `not a = b` é equivalente a `is(a # b)`, em lugar de ser equivalente a `a # b`.

`rhs` e `lhs` retornam o primeiro membro e o segundo membro de uma equação, respectivamente, de uma equação ou inequação.

Veja também `equal` e `notequal`.

Exemplos:

Uma expressão `a = b`, por si mesma, representa uma equação não avaliada, a qual pode ou não se manter.

```
(%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)` avalia `a = b` para `true` quando *a* e *b* são sintaticamente iguais (isto é, identicos). Expressões podem ser equivalentes e não sintaticamente iguais.

```
(%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]
```

Alguns operadores avaliam `=` e `#` para `true` ou `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]
```

Devido a `not` *expr* fazer com que a avaliação de *expr* ocorra, `not a = b` é equivalente a `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` [Operador]

O operador lógico de conjunção. `and` é um operador n-ário infixo; seus operandos são expressões Booleanas, e seu resultado é um valor Booleano.

`and` força avaliação (como `is`) de um ou mais operandos, e pode forçar a avaliação de todos os operandos.

Operandos são avaliados na ordem em que aparecerem. `and` avalia somente quantos de seus operandos forem necessários para determinar o resultado. Se qualquer operando for `false`, o resultado é `false` e os operandos restantes não são avaliados.

O sinalizador global `prederror` governa o comportamento de `and` quando um operando avaliado não pode ser determinado como sendo `true` ou `false`. `and` imprime uma mensagem de erro quando `prederror` for `true`. De outra forma, `and` retorna `unknown` (desconhecido).

`and` não é comutativo: `a and b` pode não ser igual a `b and a` devido ao tratamento de operandos indeterminados.

`or` [Operador]

O operador lógico de disjunção. `or` é um operador n-ário infixo; seus operandos são expressões Booleanas, e seu resultado é um valor Booleano.

`or` força avaliação (como `is`) de um ou mais operandos, e pode forçar a avaliação de todos os operandos.

Operandos são avaliados na ordem em que aparecem. `or` avalia somente quantos de seus operandos forem necessários para determinar o resultado. Se qualquer operando for `true`, o resultado é `true` e os operandos restantes não são avaliados.

O sinalizador global `prederror` governa o comportamento de `or` quando um operando avaliado não puder ser determinado como sendo `true` ou `false`. `or` imprime uma mensagem de erro quando `prederror` for `true`. De outra forma, `or` retorna `unknown`.

`or` não é comutativo: `a or b` pode não ser igual a `b or a` devido ao tratamento de operando indeterminados.

`not` [Operador]

O operador lógico de negação. `not` é operador prefixado; Seu operando é uma expressão Booleana, e seu resultado é um valor Booleano.

`not` força a avaliação (como `is`) de seu operando.

O sinalizador global `prederror` governa o comportamento de `not` quando seu operando não pode ser determinado em termos de `true` ou `false`. `not` imprime uma mensagem de erro quando `prederror` for `true`. De outra forma, `not` retorna `unknown`.

`abs` (*expr*) [Função]

Retorna o valor absoluto de *expr*. Se *expr* for um número complexo, retorna o módulo complexo de *expr*.

`additive` [Palavra chave]

Se `declare(f,additive)` tiver sido executado, então:

(1) Se `f` for uma função de uma única variável, sempre que o simplificador encontrar `f` aplicada a uma adição, `f` será distribuído sobre aquela adição. I.e. `f(y+x)` irá simplificar para `f(y)+f(x)`.

(2) Se `f` for uma função de 2 ou mais argumentos, a adição é definida como adição no primeiro argumento para `f`, como no caso de `sum` ou `integrate`, i.e. `f(h(x)+g(x),x)` irá simplificar para `f(h(x),x)+f(g(x),x)`. Essa simplificação não ocorre quando `f` é aplicada para expressões da forma `sum(x[i],i,lower-limit,upper-limit)`.

`allbut` [Palavra chave]

trabalha com os comandos `part` (i.e. `part`, `inpart`, `substpart`, `substinpart`, `dpart`, e `lpart`). Por exemplo,

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

enquanto

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

`allbut` é também reconhecido por `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, ...))` tem o mesmo efeito que `kill(all)` excepto que não elimina os símbolos *a_1*, *a_2*, ...

`antisymmetric` [Declaração]

Se `declare(h,antisymmetric)` é concluída, diz ao simplicador que `h` é uma função antisimétrica. E.g. `h(x,z,y)` simplificará para `- h(x, y, z)`. Isto é, dará (-1)^n vezes o resultado dado por `symmetric` ou `commutative`, quando n for o número de interescolhas de dois argumentos necessários para converter isso naquela forma.

`cabs` (*expr*) [Função]

Retorna o valor absoluto complexo (o módulo complexo) de *expr*.

`ceiling` (*x*) [Função]

Quando *x* for um número real, retorna o último inteiro que é maior que ou igual a *x*.

Se *x* for uma expressão constante (`10 * %pi`, por exemplo), `ceiling` avalia *x* usando grandes números em ponto flutuante, e aplica `ceiling` para o grande número em ponto flutuante resultante. Porque `ceiling` usa avaliação de ponto flutuante, é possível, embora improvável, que `ceiling` possa retornar uma valor errôneo para entradas constantes. Para prevenir erros, a avaliação de ponto flutuante é concluída usando três valores para `fpprec`.

Para entradas não constantes, `ceiling` tenta retornar um valor simplificado. Aqui está um exemplo de simplificações que `ceiling` conhece:

```
(%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
```

A função `ceiling` não mapeia automaticamente sobre listas ou matrizes. Finalmente, para todas as entradas que forem manifestamente complexas, `ceiling` retorna uma forma substantiva.

Se o intervalo de uma função é um subconjunto dos inteiros, o intervalo pode ser declarado `integervalued`. Ambas as funções `ceiling` e `floor` podem usar essa informação; por exemplo:

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

`charfun` (*p*) [Função]

Retorna 0 quando o predicado *p* avaliar para `false`; retorna 1 quando o predicado avaliar para `true`. Quando o predicado avaliar para alguma coisa que não `true` ou `false` (`unknown`), retorna uma forma substantiva.

Exemplos:

```
(%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` [Declaração]

Se `declare(h,commutative)` é concluída, diz ao simplicador que `h` é uma função comutativa. E.g. `h(x,z,y)` irá simplificar para `h(x, y, z)`. Isto é o mesmo que `symmetric`.

`compare` (*x*, *y*) [Função]

Retorna um operador de comparação *op* (`<`, `<=`, `>`, `>=`, `=`, ou `#`) tal que `is (`*x op y*`)` avalia para `true`; quando ou *x* ou *y* dependendo de `%i` e *x* `#` *y*, retorna `notcomparable`; Quando não existir tal operador ou Maxima não estiver apto a determinar o operador, retorna `unknown`.

Exemplos:

```
(%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)                             <=
```

A função `compare` não tenta de terminar se o domínio real de seus argumentos é não vazio; dessa forma

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

O domínio real de `acos (x^2 + 1)` é vazio.

`entier` (*x*) [Função]

Retorna o último inteiro menor que ou igual a *x* onde *x* é numérico. `fix` (como em `fixnum`) é um sinônimo disso, então `fix(x)` é precisamente o mesmo.

`equal (a, b)` [Função]

Representa a equivalência, isto é, valor igual.

Por si mesma, `equal` não avalia ou simplifica. A função `is` tenta avaliar `equal` para um valor Booleano. `is(equal(a, b))` retorna `true` (ou `false`) se e somente se a e b forem iguais (ou não iguais) para todos os possíveis valores de suas variáveis, como determinado através da avaliação de `ratsimp(a - b)`; se `ratsimp` retornar 0, as duas expressões são consideradas equivalentes. Duas expressões podem ser equivalentes mesmo se mesmo se elas não forem sintaticamente iguais (i.e., identicas).

Quando `is` falhar em reduzir `equal` a `true` ou `false`, o resultado é governado através do sinalizador global `prederror`. Quando `prederror` for `true`, `is` reclama com uma mensagem de erro. De outra forma, `is` retorna `unknown`.

Complementando `is`, alguns outros operadores avaliam `equal` e `notequal` para `true` ou `false`, a saber `if`, `and`, `or`, e `not`.

A negação de `equal` é `notequal`. Note que devido às regras de avaliação de expressões predicadas (em particular pelo facto de `not` *expr* causar a avaliação de *expr*), `not equal(a, b)` é equivalente a `is(notequal(a, b))` em lugar de ser equivalente a `notequal(a, b)`.

Exemplos:

Por si mesmo, `equal` não avalia ou simplifica.

```
(%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)
```

A função `is` tenta avaliar `equal` para um valor Booleano. `is(equal(a, b))` retorna `true` quando `ratsimp(a - b)` retornar 0. Duas expressões podem ser equivalentes mesmo se não forem sintaticamente iguais (i.e., identicas).

```
(%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));
Maxima was unable to evaluate the predicate:
```

```
equal(x, y)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i9) is (x = y);
(%o9)                          false
```

Quando `is` falha em reduzir `equal` a `true` ou `false`, o resultado é governado através do sinalizador global `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
```

Alguns operadores avaliam `equal` e `notequal` para `true` ou `false`.

```
(%i1) if equal (a, b) then FOO else BAR;
Maxima was unable to evaluate the predicate:
equal(a, b)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%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]
```

Devido a `not` *expr* fazer com que ocorra a avaliação de *expr*, `not equal(a, b)` é equivalente a `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*) [Função]

Quando *x* for um número real, retorna o maior inteiro que é menor que ou igual a *x*.

Se *x* for uma expressão constante (`10 * %pi`, for exemplo), `floor` avalia *x* usando grandes números em ponto flutuante, e aplica `floor` ao grande número em ponto flutuante resultante. Porque `floor` usa avaliação em ponto flutuante, é possível,

embora improvável, que `floor` não possa retornar um valor errôneo para entradas constantes. Para prevenir erros, a avaliação de ponto flutuante é concluída usando três valores para `fpprec`.

Para entradas não constantes, `floor` tenta retornar um valor simplificado. Aqui está exemplos de simplificações que `floor` conhece:

```
(%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
```

A função `floor` não mapeia automaticamente sobre listas ou matrizes. Finalmente, para todas as entradas que forem manifestamente complexas, `floor` retorna uma forma substantiva.

Se o intervalo de uma função for um subconjunto dos inteiros, o intervalo pode ser declarado `integervalued`. Ambas as funções `ceiling` e `floor` podem usar essa informação; por exemplo:

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

`notequal` (*a*, *b*) [Função]

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

Note que pelo facto de as regras de avaliação de expressões predicadas (em particular pelo facto de `not` *expr* causar a avaliação de *expr*), `not equal(a, b)` é equivalente a `is(notequal(a, b))` em lugar de ser equivalente a `notequal(a, b)`.

Exemplos:

```
(%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);
Maxima was unable to evaluate the predicate:
equal(a, b)
```

```
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i5) maybe (notequal (a, b));
(%o5)                          unknown
(%i6) maybe (not equal (a, b));
(%o6)                          unknown
(%i7) assume (a > b);
(%o7)                          [a > b]
(%i8) equal (a, b);
(%o8)                        equal(a, b)
(%i9) maybe (equal (a, b));
(%o9)                           false
(%i10) notequal (a, b);
(%o10)                     notequal(a, b)
(%i11) not equal (a, b);
(%o11)                          true
(%i12) maybe (notequal (a, b));
(%o12)                          true
(%i13) maybe (not equal (a, b));
(%o13)                          true
```

`eval` [Operador]

Como um argumento em uma chamada a `ev (expr)`, `eval` causa uma avaliação extra de *expr*. Veja `ev`.

`evenp (expr)` [Função]

Retorna `true` se *expr* for um inteiro sempre. `false` é retornado em todos os outros casos.

`fix (x)` [Função]

Um sinônimo para `entier (x)`.

`fullmap (f, expr_1, ...)` [Função]

Similar a `map`, mas `fullmap` mantém mapeadas para baixo todas as subexpressões até que os operadores principais não mais sejam os mesmos.

`fullmap` é usada pelo simplificador do Maxima para certas manipulações de matrizes; dessa forma, Maxima algumas vezes gera uma mensagem de erro concernente a `fullmap` mesmo apesar de `fullmap` não ter sido explicitamente chamada pelo utilizador.

Exemplos:

```
(%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, ...)` [Função]

Similar a `fullmap`, mas `fullmapl` somente mapeia sobre listas e matrizes.

Exemplo:

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

is (*expr*) [Função]

Tenta determinar se a *expr* predicada (expressões que avaliam para `true` ou `false`) é dedutível de factos localizados na base de dados de `assume`.

Se a dedutibilidade do predicado for `true` ou `false`, `is` retorna `true` ou `false`, respectivamente. De outra forma, o valor de retorno é governado através do sinalizador global `prederror`. Quando `prederror` for `true`, `is` reclama com uma mensagem de erro. De outra forma, `is` retorna `unknown`.

`ev(`*expr*`, pred)` (que pode ser escrita da forma *expr*`, pred` na linha de comando interativa) é equivalente a `is(`*expr*`)`.

Veja também `assume`, `facts`, e `maybe`.

Exemplos:

`is` causa avaliação de predicados.

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

`is` tenta derivar predicados da base de dados do `assume`.

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

Se `is` não puder nem comprovar nem refutar uma forma predicada a partir da base de dados de `assume`, o sinalizador global `prederror` governa o comportamento de `is`.

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

maybe (*expr*) [Função]

Tenta determinar se a *expr* predicada é dedutível dos factos na base de dados de `assume`.

Se a dedutibilidade do predicado for `true` ou `false`, `maybe` retorna `true` ou `false`, respectivamente. De outra forma, `maybe` retorna `unknown`.

`maybe` é funcinalmente equivalente a `is` com `prederror: false`, mas o resultado é computado sem actualmente atribuir um valor a `prederror`.

Veja também `assume`, `facts`, e `is`.

Exemplos:

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

isqrt (*x*) [Função]

Retorna o "inteiro raíz quadrada" do valor absoluto de *x*, que é um inteiro.

lmax (*L*) [Função]

Quando *L* for uma lista ou um conjunto, retorna `apply ('max, args (L))`. Quando *L* não for uma lista ou também não for um conjunto, sinaliza um erro.

lmin (*L*) [Função]

Quando *L* for uma lista ou um conjunto, retorna `apply ('min, args (L))`. Quando *L* não for uma lista ou ou também não for um conjunto, sinaliza um erro.

max (*x_1*, ..., *x_n*) [Função]

Retorna um valor simplificado para o máximo entre as expressões *x_1* a *x_n*. Quando `get (trylevel, maxmin)`, for dois ou mais, `max` usa a simplificação `max (e, -e) --> |e|`. Quando `get (trylevel, maxmin)` for 3 ou mais, *max* tenta eliminar expressões que estiverem entre dois outros argumentos; por exemplo, `max (x, 2*x, 3*x) --> max (x, 3*x)`. Para escolher o valor de `trylevel` para 2, use `put (trylevel, 2, maxmin)`.

min (*x_1*, ..., *x_n*) [Função]

Retorna um valor simplificado para o mínimo entre as expressões `x_1` até `x_n`. Quando `get (trylevel, maxmin)`, for 2 ou mais, `min` usa a simplificação `min (e, -e) --> -|e|`. Quando `get (trylevel, maxmin)` for 3 ou mais, `min` tenta eliminar expressões que estiverem entre dois outros argumentos; por exemplo, `min (x, 2*x, 3*x) --> min (x, 3*x)`. Para escolher o valor de `trylevel` para 2, use `put (trylevel, 2, maxmin)`.

polymod (*p*) [Função]
polymod (*p*, *m*) [Função]

Converte o polinómio *p* para uma representação modular com relação ao módulo corrente que é o valor da variável `modulus`.

`polymod (p, m)` especifica um módulo m para ser usado em lugar do valor corrente de `modulus`.

Veja `modulus`.

`mod (x, y)` [Função]

Se x e y forem números reais e y for não nulo, retorna `x - y * floor(x / y)`. Adicionalmente para todo real x, nós temos `mod (x, 0) = x`. Para uma discursão da definição `mod (x, 0) = x`, veja a Seção 3.4, de "Concrete Mathematics," por Graham, Knuth, e Patashnik. A função `mod (x, 1)` é uma função dente de serra com período 1 e com `mod (1, 1) = 0` e `mod (0, 1) = 0`.

Para encontrar o argumento (um número no intervalo `(-%pi, %pi]`) de um número complexo, use a função `x |-> %pi - mod (%pi - x, 2*%pi)`, onde x é um argumento.

Quando x e y forem expressões constantes (`10 * %pi`, por exemplo), `mod` usa o mesmo esquema de avaliação em ponto flutuante que `floor` e `ceiling` usam. Novamente, é possível, embora improvável, que `mod` possa retornar um valor errôneo nesses casos.

Para argumentos não numéricos x ou y, `mod` conhece muitas regras de simplificação:

```
(%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)` [Função]

é `true` se *expr* for um inteiro ímpar. `false` é retornado em todos os outros casos.

`pred` [Operador]

Como um argumento em uma chamada a `ev (expr)`, `pred` faz com que predicados (expressões que avaliam para `true` ou `false`) sejam avaliados. Veja `ev`.

`make_random_state (n)` [Função]
`make_random_state (s)` [Função]
`make_random_state (true)` [Função]
`make_random_state (false)` [Função]

Um objecto de estado aleatório representa o estado do gerador de números aleatórios (aleatórios). O estado compreende 627 palavras de 32 bits.

`make_random_state (n)` retorna um novo objecto de estado aleatório criado de um valor inteiro semente igual a n modulo 2^32. n pode ser negativo.

`make_random_state (s)` retorna uma copia do estado aleatório s.

`make_random_state (true)` retorna um novo objecto de estado aleatório, usando a hora corrente do relógio do computador como semente.

`make_random_state (false)` retorna uma cópia do estado corrente do gerador de números aleatórios.

`set_random_state (s)` [Função]

Copia s para o estado do gerador de números aleatórios.

`set_random_state` sempre retorna `done`.

`random (x)` [Função]

Retorna um número pseudoaleatório. Se x é um inteiro, `random (x)` retorna um inteiro de 0 a `x - 1` inclusive. Se x for um número em ponto flutuante, `random (x)` retorna um número não negativo em ponto flutuante menor que x. `random` reclama com um erro se x não for nem um inteiro nem um número em ponto flutuante, ou se x não for positivo.

As funções `make_random_state` e `set_random_state` mantém o estado do gerador de números aleatórios.

O gerador de números aleatórios do Maxima é uma implementação do algoritmo de Mersenne twister MT 19937.

Exemplos:

```
(%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)` [Função]

Converte todos os números em ponto flutuante de precisão dupla e grandes números em ponto flutuante na expressão do Maxima *expr* para seus exatos equivalentes racionais. Se vnão estiver familiarizado com a representação binária dos números em ponto flutuante, pode ficar surpreendido em saber que `rationalize (0.1)` não é igual a 1/10. Esse comportamento não é especial do Maxima – o número 1/10 tem uma representação binária repetitiva e não terminada.

```
(%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
```

Exemplo de utilização:

```
(%i1) unitfrac(r) := block([uf : [], q],
    if not(ratnump(r)) then error("The input to 'unitfrac' must be a rational num
    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
```

`sign` (*expr*) [Função]

Tenta determinar o sinal de *expr* a partir dos factos na base de dados corrente. Retorna uma das seguintes respostar: `pos` (positivo), `neg` (negativo), `zero`, `pz` (positivo ou zero), `nz` (negativo ou zero), `pn` (positivo ou negativo), ou `pnz` (positivo, negativo, ou zero, i.e. nada se sabe sobre o sinal da epressão).

`signum` (*x*) [Função]

Para um *x* numérico retorna 0 se *x* for 0, de outra forma retorna -1 ou +1 à medida que *x* seja menor ou maior que 0, respectivamente.

Se *x* não for numérico então uma forma simplificada mas equivalente é retornada. Por exemplo, `signum(-x)` fornece `-signum(x)`.

`sort` (*L*, *P*) [Função]

`sort` (*L*) [Função]

Organiza uma lista *L* coforme o predicado `P` de dois argumentos, de forma que *P* (*L*`[k]`, *L*`[k + 1]`) seja `true` para qualquer dois elementos sucessivos. O predicado pode ser especificado como o nome de uma função ou operador binário infixo, ou como uma expressão `lambda`. Se especificado como o nome de um operador, o nome deve ser contido entre "aspas duplas".

A lista ordenada é retornada como novo objecto; o argumento *L* não é modificado. Para construir o valor de retorno, `sort` faz uma cópia superficial dos elementos de *L*. Se o predicado *P* não for uma ordem total sobre os elementos de *L*, então `sort` possivelvente pode executar para concluir sem error, mas os resultados são indefinidos. `sort` reclama se o predicado avaliar para alguma outra coisa que não seja `true` ou `false`.

`sort (`*L*`)` é equivalente a `sort (`*L*`, orderlessp)`. Isto é, a ordem padrão de organização é ascendente, como determinado por `orderlessp`. Todos os átomos do Maxima e expressões são comparáveis sob `orderlessp`, embora exista exemplos isolados de expressões para as quais `orderlessp` não é transitiva; isso é uma falha.

Exemplos:

```
(%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*) [Função]

A raíz quadrada de *x*. É representada internamente por `x^(1/2)`. Veja também `rootscontract`.

`radexpand` se `true` fará com que n-ésimas raízes de factores de um produto que forem potências de n sejam colocados fora do radical, e.g. `sqrt(16*x^2)` retonará `4*x` somente se `radexpand` for `true`.

`sqrtdispflag` [Variável de opção]

Valor por omissão: `true`

Quando `sqrtdispflag` for `false`, faz com que `sqrt` seja mostrado como expoente 1/2.

`sublis` (*lista*, *expr*) [Função]

Faz múltiplas substituições paralelas dentro de uma expressão.

A variável `sublis_apply_lambda` controla a simplificação após `sublis`.

Exemplo:

`sublist` (*lista*, *p*) [Função]

Retorna a lista de elementos da *lista* da qual o predicado `p` retornar `true`.

Exemplo:

```
(%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` [Variável de opção]

Valor por omissão: `true` - controla se os substitutos de `lambda` são aplicados na simplificação após as `sublis` serem usadas ou se tiver que fazer um `ev` para obter coisas para aplicar. `true` significa faça a aplicação.

`subst` (*a*, *b*, *c*) [Função]

Substitue *a* por *b* em *c*. *b* deve ser um átomo ou uma subexpressão completa de *c*. Por exemplo, `x+y+z` é uma subexpressão completa de `2*(x+y+z)/w` enquanto `x+y` não

é. Quando *b* não tem essas características, pode-se algumas vezes usar `substpart` ou `ratsubst` (veja abaixo). Alternativamente, se *b* for da forma de `e/f` então se poderá usar `subst (a*f, e, c)` enquanto se *b* for da forma `e^(1/f)` então se poderá usar `subst (a^f, e, c)`. O comando `subst` também discerne o `x^y` de `x^-y` de modo que `subst (a, sqrt(x), 1/sqrt(x))` retorna `1/a`. *a* e *b* podem também ser operadores de uma expressão contida entre aspas duplas `"` ou eles podem ser nomes de função. Se se desejar substituir por uma variável independente em formas derivadas então a função `at` (veja abaixo) poderá ser usada.

`subst` é um álias para `substitute`.

`subst (`*eq_1*`, `*expr*`)` ou `subst ([`*eq_1*`, ..., `*eq_k*`], `*expr*`)` são outras formas permitidas. As *eq_i* são equações indicando substituições a serem feitas. Para cada equação, o lado direito será substituído pelo lado esquerdo na expressão *expr*.

`exptsubst` se `true` permite que substituições como `y` por `%e^x` em `%e^(a*x)` ocorram.

Quando `opsubst` for `false`, `subst` tentará substituir dentro do operador de uma expressão. E.g. `(opsubst: false, subst (x^2, r, r+r[0]))` trabalhará.

Exemplos:

```
(%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
```

Para exemplos adicionais, faça `example (subst)`.

`substinpart (`*x*, *expr*, *n_1*, ..., *n_k*`)` [Função]

Similar a `substpart`, mas `substinpart` trabalha sobre a representação interna de *expr*.

Exemplos:

```
(%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)
```

Se o último argumento para a função `part` for uma lista de índices então muitas subexpressões são escolhidas, cada uma correspondendo a um índice da lista. Dessa forma

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

`piece` recebe o valor da última expressão seleccionada quando usando as funções `part`. `piece` é escolhida durante a execução da função e dessa forma pode ser referenciada

para a própria função como mostrado abaixo. Se `partswitch` for escolhida para `true` então `end` é retornado quando uma parte seleccionada de uma expressão não existir, de outra forma uma mensagem de erro é fornecida.

```
(%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);
(%o3)                         abs(y)
(%i4) substpart (factor (piece), expr, [1, 2, 3, 5]);
                                3
(%o4)               (3 y + 2 x)  + y + x + 1
(%i5) expr: 1/x + y/x - 1/z;
                              1   y   1
(%o5)                       - - + - + -
                              z   x   x
(%i6) substpart (xthru (piece), expr, [2, 3]);
                             y + 1   1
(%o6)                        ----- - -
                               x     z
```

Também, escolhendo a opção `inflag` para `true` e chamando `part` ou `substpart` é o mesmo que chamando `inpart` ou `substinpart`.

`substpart` (*x*, *expr*, *n_1*, ..., *n_k*) [Função]

Substitue *x* para a subexpressão seleccionada pelo resto dos argumentos como em `part`. Isso retorna o novo valor de *expr*. *x* pode ser algum operador a ser substituído por um operador de *expr*. Em alguns casos *x* precisa ser contido em aspas duplas `"` (e.g. `substpart ("+", a*b, 0)` retorna `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);
(%o3)                      a x + f(b, y)
(%i4) substpart ("+", %, 1, 0);
(%o4)                    x + f(b, y) + a
```

Também, escolhendo a opção `inflag` para `true` e chamando `part` ou `substpart` é o mesmo que chamando `inpart` ou `substinpart`.

`subvarp` (*expr*) [Função]

Retorna `true` se *expr* for uma variável subscrita (i.e. que possui índice ou subscrito em sua grafia), por exemplo `a[i]`.

`symbolp` (*expr*) [Função]

Retorna `true` se *expr* for um símbolo, de outra forma retorna `false`. com efeito, `symbolp(x)` é equivalente ao predicado `atom(x) and not numberp(x)`.

Veja também *Identificadores*

`unorder` () [Função]

Disabilita a ação de alias criada pelo último uso dos comandos de ordenação `ordergreat` e `orderless`. `ordergreat` e `orderless` não podem ser usados mais que uma vez cada sem chamar `unorder`. Veja também `ordergreat` e `orderless`.

Exemplos:

```
(%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*) [Função]

Retorna o potencial do vector de um dado vector de torção, no sistema de coordenadas corrente. `potentialzeroloc` tem um papel similar ao de `potential`, mas a ordem dos lados esquerdos das equações deve ser uma permutação cíclica das variáveis de coordenadas.

`xthru` (*expr*) [Função]

Combina todos os termos de *expr* (o qual pode ser uma adição) sobre um denominador comum sem produtos e somas exponenciadas como `ratsimp` faz. `xthru` cancela factores comuns no numerador e denominador de expressões racionais mas somente se os factores são explícitos.

Algumas vezes é melhor usar `xthru` antes de `ratsimp` em uma expressão com o objectivo de fazer com que factores explicitos do máximo divisor comum entre o numerador e o denominador seja cancelado simplificando dessa forma a expressão a ser aplicado o `ratsimp`.

```
(%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*) [Função]

Testa se a expressão *expr* na variável *v* é equivalente a zero, retornando `true`, `false`, ou `dontknow` (não sei).

`zeroequiv` Tem essas restrições:

1. Não use funções que o Maxima não sabe como diferenciar e avaliar.
2. Se a expressão tem postes sobre o eixo real, podem existir erros no resultado (mas isso é improvável ocorrer).
3. Se a expressão contem funções que não são soluções para equações diferenciais de primeira ordem (e.g. funções de Bessel) pode ocorrer resultados incorrectos.
4. O algoritmo usa avaliação em pontos aleatóriamente escolhidos para subexpressões seleccionadas cuidadosamente. Isso é sempre negócio um tanto quanto perigoso, embora o algoritmo tente minimizar o potencial de erro.

Por exemplo `zeroequiv (sin(2*x) - 2*sin(x)*cos(x), x)` retorna `true` e `zeroequiv (%e^x + x, x)` retorna `false`. Por outro lado `zeroequiv (log(a*b) - log(a) - log(b), a)` retorna `dontknow` devido à presença de um parâmetro extra `b`.

6 Expressões

6.1 Introdução a Expressões

Existe um conjunto de palavras reservadas que não pode ser usado como nome de variável. Seu uso pode causar um possível erro crítico de sintaxe.

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

Muitas coisas em Maxima são expressões. Uma sequência de expressões pode ser feita dentro de uma expressão maior através da separação dessas através de vírgulas e colocando parêntesis em torno dela. Isso é similar ao **C** *expressão com vírgula*.

```
(%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
```

Mesmo ciclos em Maxima são expressões, embora o valor de retorno desses ciclos não seja muito útil (eles retornam sempre `done`).

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

enquanto que o que realmente queira seja provavelmente incluir um terceiro termo na *expressão com vírgula* que fornece de volta o valor actualizado.

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

6.2 Complexo

Uma expressão complexa é especificada no Maxima através da adição da parte real da expressão a `%i` vezes a parte imaginária. Dessa forma as raízes da equação `x^2 - 4*x + 13 = 0` são `2 + 3*%i` e `2 - 3*%i`. Note que produtos de simplificação de expressões complexas podem ser efetuadas através da expansão do produto. Simplificação de quocientes, raízes, e outras funções de expressões complexas podem usualmente serem realizadas através do uso das funções `realpart`, `imagpart`, `rectform`, `polarform`, `abs`, `carg`.

6.3 Substantivos e Verbos

Maxima distingue entre operadores que são "substantivos" e operadores que são "verbos". Um verbo é um operador que pode ser executado. Um substantivo é um operador que

aparece como um símbolo em uma expressão, sem ser executado. Por padrão, nomes de função são verbos. Um verbo pode ser mudado em um substantivo através da adição de um apóstrofo no início do nome da função ou aplicando a função `nounify`. Um substantivo pode ser mudado em um verbo através da aplicação da função `verbify`. O sinalizador de avaliação `nouns` faz com que `ev` avalie substantivos em uma expressão.

A forma verbal é distinguida através de um sinal de dólar `$` no início do símbolo Lisp correspondente. De forma oposta, a forma substantiva é distinguida através de um sinal de `%` no início do símbolo Lisp correspondente. Alguns substantivos possuem propriedades especiais de exibição, tais como `'integrate` e `'derivative` (retornado por `diff`), mas muitos não. Por padrão, as formas substantiva e verbal de uma função são idênticas quando mostradas. O sinalizador global `noundisp` faz com que Maxima mostre substantivos com um apóstrofo no início `'`.

Veja também `noun`, `nouns`, `nounify`, e `verbify`.

Exemplos:

```
(%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 Identificadores

Identificadores do Maxima podem compreender caracteres alfabéticos, mais os numerais de 0 a 9, mais qualquer caractere especial precedido por um caractere contra-barra `\`.

Um numeral pode ser o primeiro caractere de um identificador se esse numeral for precedido por uma contra-barra. Numerais que forem o segundo ou o último caractere não precisam ser precedidos por uma contra barra.

Caracteres podem ser declarados para serem alfabéticos por meio da função `declare`. Se então declarados alfabéticos, eles não precisam serem precedidos de uma contrabarra em um identificador. Os caracteres alfabéticos vão inicialmente de `A` a `Z`, de `a` a `z`, `%`, e `_`.

Maxima é sensível à caixa . Os identificadores `algumacoisa`, `ALGUMACOISA`, e `Algumacoisa` são distintos. Veja *Lisp e Maxima* para mais sobre esse ponto.

Um identificador Maxima é um símbolo Lisp que começa com um sinal de dólar `$`. Qualquer outro símbolo Lisp é precedido por um ponto de interrogação `?` quando aparecer no Maxima. Veja *Lisp e Maxima* para maiores detalhes sobre esse ponto.

Exemplos:

```
(%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 Sequências de caracteres

Strings (sequências de caracteres) são contidas entre aspas duplas " em entradas de dados usados pelo Maxima, e mostradas com ou sem as aspas duplas, dependendo do valor escolhido para a variável global `stringdisp`.

Sequências de caracteres podem conter quaisquer caracteres, incluindo tabulações (tab), nova linha (ou fim de linha), e caracteres de retorno da cabeça de impressão (carriage return). A sequência \" é reconhecida com uma aspa dupla literal, e \\ como uma contrabarra literal. Quando a contrabarra aparecer no final de uma linha, a contrabarra e a terminação de linha (ou nova linha ou retorno de carro e nova linha) são ignorados, de forma que a sequência de caracteres continue na próxima linha. Nenhuma outra combinação especial de contrabarra com outro caractere é reconhecida; quando a contrabarra aparecer antes de qualquer outro caractere que não seja ", \, ou um fim de linha, a contrabarra é ignorada. Não exite caminho para representar um caractere especial (tal como uma tabulação, nova linha, ou retorno da cabeça de impressão) excepto através de encaixar o caractere literal na sequência de caracteres.

Não existe tipo de caractere no Maxima; um caractere simples é representado como uma sequência de caracteres de um único caractere.

Sequências de caracteres no Maxima são implementadas como símbolos do Lisp, não como sequencias de caracteres do not Lisp; o que pode mudar em futuras versões do Maxima. Maxima pode mostrar sequências de caracteres do Lisp e caracteres do Lisp, embora algumas outras operações (por exemplo, testes de igualdade) possam falhar.

O pacote adicional `stringproc` contém muitas funções que trabalham com sequências de caracteres.

Exemplos:

```
(%i1) s_1 : "Isso é uma sequência de caracteres  do Maxima.";
(%o1)               Isso é uma sequência de caracteres  do Maxima.
(%i2) s_2 : "Caracteres \"aspas duplas\" e contrabarras \\ encaixados em uma sequência
(%o2) Caracteres "aspas duplas" e contrabarra \ encaixados em uma sequência de caracte
(%i3) s_3 : "Caractere de fim de linha encaixado
nessa sequência de caracteres.";
(%o3) Caractere de fim de linha encaixado
nessa sequência de caracteres.
(%i4) s_4 : "Ignore o \
caractere de \
fim de linha nessa \
sequência de caracteres.";
(%o4) Ignore o caractere de fim de linha nessa sequência de caracteres.
(%i5) stringdisp : false;
(%o5)                          false
(%i6) s_1;
(%o6)                Isso é uma sequência de caracteres  do Maxima.
(%i7) stringdisp : true;
(%o7)                          true
(%i8) s_1;
(%o8)              "Isso é uma sequência de caracteres  do Maxima."
```

6.6 Desigualdade

Maxima tem os operadores de desigualdade `<`, `<=`, `>=`, `>`, `#`, e `notequal`. Veja `if` para uma descrição de expressões condicionais.

6.7 Sintaxe

É possível definir novos operadores com precedência especificada, remover a definição de operadores existentes, ou redefinir a precedência de operadores existentes. Um operador pode ser unário prefixado ou unário pósfixado, binario infixado, n-ário infixado, matchfix, ou nofix. "Matchfix" significa um par de símbolos que abraçam seu argumento ou seus argumentos, e "nofix" significa um operador que não precisa de argumentos. Como exemplos dos diferentes tipos de operadores, existe o seguinte.

unário prefixado
: negação `- a`

unário posfixado
: factorial `a!`

binário infixado
: exponenciação `a^b`

n-ário infixado
: adição `a + b`

matchfix
: construção de lista `[a, b]`

(Não existe operadores internos nofix; para um exemplo de tal operador, veja `nofix`.)

O mecanismo para definir um novo operador é directo. Somente é necessário declarar uma função como um operador; a função operador pode ou não estar definida previamente.

Um exemplo de operadores definidos pelo utilizador é o seguinte. Note que a chamada explícita de função `"dd" (a)` é equivalente a `dd a`, da mesma forma `"<-" (a, b)` é equivalente a `a <- b`. Note também que as funções `"dd"` e `"<-"` são indefinidas nesse exemplo.

```
(%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
```

As funções máxima que definem novos operadores estão sumarizadas nessa tabela, equilibrando expoente associado esquerdo (padrão) e o expoente associado direito (`"eae"` e `"ead"`, respectivamente). (Associação de expoentes determina a precedência do operador. todavia, uma vez que os expoentes esquerdo e direito podem ser diferentes, associação de expoentes

é até certo ponto mais complicado que precedência.) Alguma das funções de definição de operações tomam argumentos adicionais; veja as descrições de função para maiores detalhes.

`prefixado`
: ead=180

`posfixado`
: eae=180

`infixado` eae=180, ead=180

`nário` eae=180, ead=180

`matchfix` (associação de expoentes não é aplicável)

`nofix` (associação de expoentes não é aplicável)

Para comparação, aqui está alguns operadores internos e seus expoentes associados esquerdo e direito.

Operador	eae	ead
:	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` e `kill` removem propriedades de operador de um átomo. `remove ("a", op)` remove somente as propriedades de operador de *a*. `kill ("a")` remove todas as propriedades de *a*, incluindo as propriedades de operador. Note que o nome do operador dever estar abraçado por aspas duplas.

```
(%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 Definições para Expressões

`at` (*expr*, [*eqn_1*, ..., *eqn_n*]) [Função]
`at` (*expr*, *eqn*) [Função]

Avalia a expressão *expr* com as variáveis assumindo os valores como especificado para elas na lista de equações [*eqn_1*, ..., *eqn_n*] ou a equação simples *eqn*.

Se uma subexpressão depender de qualquer das variáveis para a qual um valor foi especificado mas não existe `atvalue` especificado e essa subexpressão não pode ser avaliada de outra forma, então uma forma substantiva de `at` é retornada que mostra em uma forma bidimensional.

`at` realiza múltiplas substituições em série, não em paralelo.

Veja também `atvalue`. Para outras funções que realizam substituições, veja também `subst` e `ev`.

Exemplos:

```
(%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*) [Função]
`box` (*expr*, a) [Função]

Retorna *expr* dentro de uma caixa. O valor de retorno é uma expressão com `box` como o operador e *expr* como o argumento. Uma caixa é desenhada sobre a tela quando `display2d` for `true`.

`box` (*expr*, a) Empacota *expr* em uma caixa rotulada pelo símbolo a. O rótulo é truncado se for maior que a largura da caixa.

`box` avalia seu argumento. Todavia, uma expressão dentro de uma caixa não avalia para seu conteúdo, então expressões dentro de caixas são efectivamente excluídas de cálculos.

`boxchar` é o caractere usado para desenhar a caixa em `box` e nas funções `dpart` e `lpart`.

Exemplos:

```
(%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` [Variável de opção]

Valor por omissão: "

`boxchar` é o caractere usado para desenhar a caixa por `box` e nas funções `dpart` e `lpart`.

Todas as caixas em uma expressão são desenhadas com o valor actual de `boxchar`; o caractere de desenho não é armazenado com a expressão de caixa. Isso quer dizer que se desenhar uma caixa e em seguida mudar o caracter de desenho a caixa anteriormente desenhada será redesenhada com o caracter mudado caso isso seja solicitado.

`carg` (*z*) [Função]

Retorna o argumento complexo de *z*. O argumento complexo é um ângulo `theta` no intervalo de `(-%pi, %pi]` tal que `r exp (theta %i) = z` onde `r` é o módulo de *z*.

`carg` é uma função computacional, não uma função de simplificação.

`carg` ignora a declaração `declare (x, complex)`, e trata *x* como uma variável real. Isso é um erro.

Veja também `abs` (módulo de número complexo), `polarform`, `rectform`, `realpart`, e `imagpart`.

Exemplos:

```
(%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` [Opereador especial]

`declare (a, constant)` declara *a* para ser uma constante. Veja `declare`.

`constantp` (*expr*) [Função]

Retorna `true` se *expr* for uma expressão constante, de outra forma retorna `false`.

Uma expressão é considerada uma expressão constante se seus argumentos forem números (incluindo números racionais, como mostrado com `/R/`), constantes simbólicas como `%pi`, `%e`, e `%i`, variáveis associadas a uma constante ou constante declarada através de `declare`, ou funções cujos argumentos forem constantes.

`constantp` avalia seus argumentos.

Exemplos:

```
(%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*, ...) [Função]

Atribui aos átomos ou lista de átomos *a_i* a propriedade ou lista de propriedades *p_i*. Quando *a_i* e/ou *p_i* forem listas, cada um dos átomos recebe todas as propriedades.

`declare` não avalia seus argumentos. `declare` sempre retorna `done`.

Como colocado na descrição para cada sinalizador de declaração, para alguns sinalizadores `featurep(`*objecto*`, `*recurso*`)` retorna `true` se *objecto* tiver sido declarado para ter *recurso*. Todavia, `featurep` não reconhece alguns sinalizadores; isso é um erro.

Veja também `features`.

`declare` reconhece as seguintes propriedades:

`evfun` Torna *a_i* conhecido para `ev` de forma que a função nomeada por *a_i* é aplicada quando *a_i* aparece como um sinalizador argumento de `ev`. Veja `evfun`.

`evflag` Torna *a_i* conhecido para a função `ev` de forma que *a_i* é associado a `true` durante a execução de `ev` quando *a_i* aparece como um sinalizador argumento de `ev`. Veja `evflag`.

`bindtest` Diz ao Maxima para disparar um erro quando *a_i* for avaliado como sendo livre de associação.

`noun` Diz ao Maxima para passar *a_i* como um substantivo. O efeito disso é substituir intâncias de *a_i* com `'a_i` ou `nounify(a_i)`, ependendo do contexto.

`constant` Diz ao Maxima para considerar *a_i* uma constante simbólica.

`scalar` Diz ao Maxima para considerar *a_i* uma variável escalar.

`nonscalar`
Diz ao Maxima para considerar *a_i* uma variável não escalar. The usual application is to declare a variable as a symbolic vector or matrix.

`mainvar` Diz ao Maxima para considerar *a_i* uma "variável principal" (`mainvar`). `ordergreatp` determina a ordenação de átomos como segue:

(variáveis principais) `>` (outras variáveis) `>` (variáveis escalares) `>` (constantes) `>` (números)

`alphabetic`
Diz ao Maxima para reconhecer todos os caracteres em *a_i* (que deve ser uma sequência de caracteres) como caractere alfabético.

`feature` Diz ao Maxima para reconhecer *a_i* como nome de um recurso. Other atoms may then be declared to have the *a_i* property.

`rassociative`, `lassociative`
Diz ao Maxima para reconhecer *a_i* como uma funcão associativa a direita ou associativa a esquerda.

`nary` Diz ao Maxima para reconhecer *a_i* como uma função n-ária (com muitos argumentos).

A declaração `nary` não tem o mesmo objectivo que uma chamada à função `nary`. O único efeito de `declare(foo, nary)` é para instruir o simplificador do Maxima a melhorar as próximas expressões, por exemplo, para simplificar `foo(x, foo(y, z))` para `foo(x, y, z)`.

`symmetric`, `antisymmetric`, `commutative`
Diz ao Maxima para reconhecer *a_i* como uma função simétrica ou antisimétrica. `commutative` é o mesmo que `symmetric`.

`oddfun`, `evenfun`
Diz ao Maxima para reconhecer *a_i* como uma função par ou uma função ímpar.

`outative` Diz ao Maxima para simplificar expressões *a_i* colocando factores constantes em evidência no primeiro argumento.

Quando *a_i* tiver um argumento, um factor é onsiderado constante se for um literal ou se for declarado como sendo constante.

Quando *a_i* tiver dois ou mais argumentos, um factor é considerado constante se o segundo argumento for um símbolo e o factor estiver livre do segundo argumento.

`multiplicative`
Diz ao Maxima para simplificar expressões do tipo *a_i* através da substituição `a_i(x * y * z * ...) --> a_i(x) * a_i(y) * a_i(z) * ...`. A substituição é realizada no primeiro argumento somente.

`additive` Diz ao Maxima para simplificar expressões do tipo *a_i* através da substituição `a_i(x + y + z + ...) --> a_i(x) + a_i(y) + a_i(z) + ...`. A substituição é realizada no primeiro argumento somente.

`linear` Equivalente a declarar *a_i* ao mesmo tempo `outative` e `additive`.

`integer`, `noninteger`
Diz ao Maxima para reconhecer *a_i* como como uma variável inteira ou como uma variável não inteira.

Maxima reconhece os seguintes recursos de objectos:

`even`, `odd` Diz ao Maxima para reconhecer *a_i* como uma variável inteira par ou como uma variável inteira ímpar.

`rational`, `irrational`
Diz ao Maxima para reconhecer *a_i* como uma variável real e racional ou como uma variável real e irracional.

`real`, `imaginary`, `complex`
Dia ao Maxima para reconhecer *a_i* como uma variável real, imaginária pura ou complexa.

`increasing`, `decreasing`
Dia ao Maxima para reconhecer *a_i* como uma função de incremento ou decremento.

`posfun` Diz ao Maxima para reconhecer *a_i* como uma função positiva.

`integervalued`
Diz ao Maxima para reconhecer *a_i* como uma função de valores inteiros.

Exemplos:

Declarações `evfun` e `evflag`.

```
(%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))
```

Declaração `bindtest`.

```
(%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
```

Declaração `noun`.

```
(%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
```

Declarações `constant`, `scalar`, `nonscalar`, e `mainvar`.

Declaração `alphabetic`.

```
(%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~]
```

Declaração `feature`.

```
(%i1) declare (FOO, feature);
```

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

Declarações `rassociative` e `lassociative`.

Declaração `nary`.

```
(%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)
```

Declarações `symmetric` e `antisymmetric`.

```
(%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)
```

Declarações `oddfun` e `evenfun`.

```
(%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
```

Declaração `outative`.

```
(%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)
```

Declaração `multiplicative`.

```
(%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)
```

Declaração `additive`.

```
(%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)
```

Declaração `linear`.

```
(%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*) [Função]

é similar a `isolate (`*expr*`,` *x*`)` excepto que essa função habilita ao utilizador isolar mais que uma variável simultâneamente. Isso pode ser útil, por exemplo, se se

tiver tentado mudar variáveis em uma integração múltipla, e em mudança de variável envolvendo duas ou mais das variáveis de integração. Essa função é chamada automaticamente de `simplification/disol.mac`. Uma demostração está disponível através de `demo("disol")$`.

`dispform` (*expr*) [Função]

Retorna a representação externa de *expr* com relação a seu principal operador. Isso pode ser útil em conjunção com `part` que também lida com a representação externa. Suponha que *expr* seja -A . Então a representação interna de *expr* é `"*"`(-1,A), enquanto que a representação externa é `"-"`(A). `dispform (expr, all)` converte a expressão inteira (não apenas o nível mais alto) para o formato externo. Por exemplo, se `expr: sin (sqrt (x))`, então `freeof (sqrt, expr)` e `freeof (sqrt, dispform (expr))` fornece `true`, enquanto `freeof (sqrt, dispform (expr, all))` fornece `false`.

`distrib` (*expr*) [Função]

Distribue adições sobre produtos. `distrib` difere de `expand` no facto de que `distrib` trabalha em somente no nível mais alto de uma expressão, i.e., `distrib` não é recursiva e `distrib` é mais rápida que `expand`. `distrib` difere de `multthru` no que `distrib` expande todas as adições naquele nível.

Exemplos:

```
(%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*) [Função]

Selecciona a mesma subexpressão que `part`, mas em lugar de apenas retornar aquela subexpressão como seu valor, isso retorna a expressão completa com a subexpressão seleccionada mostrada dentro de uma caixa. A caixa é actualmente parte da expressão.

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

`exp (x)` [Função]

Representa função exponencial. Instâncias de `exp (x)` em uma entrada são simplificadas para `%e^x`; `exp` não aparece em expressões simplificadas.

`demoivre` se `true` faz com que `%e^(a + b %i)` simplificar para `%e^(a (cos(b) + %i sin(b)))` se `b` for livre de `%i`. veja `demoivre`.

`%emode`, quando `true`, faz com que `%e^(%pi %i x)` seja simplificado. Veja `%emode`.

`%enumer`, quando `true` faz com que `%e` seja substituído por 2.718... quando `numer` for `true`. Veja `%enumer`.

`%emode` [Variável de opção]

Valor por omissão: `true`

Quando `%emode` for `true`, `%e^(%pi %i x)` é simplificado como segue.

`%e^(%pi %i x)` simplifica para `cos (%pi x) + %i sin (%pi x)` se `x` for um inteiro ou um múltiplo de 1/2, 1/3, 1/4, ou 1/6, e então é adicionalmente simplificado.

Para outro `x` numérico, `%e^(%pi %i x)` simplifica para `%e^(%pi %i y)` onde `y` é `x - 2 k` para algum inteiro `k` tal que `abs(y) < 1`.

Quando `%emode` for `false`, nenhuma simplificação adicional de `%e^(%pi %i x)` é realizada.

`%enumer` [Variável de opção]

Valor por omissão: `false`

Quando `%enumer` for `true`, `%e` é substituido por seu valor numérico 2.718... mesmo que `numer` seja `true`.

Quando `%enumer` for `false`, essa substituição é realizada somente se o expoente em `%e^x` avaliar para um número.

Veja também `ev` e `numer`.

`exptisolate` [Variável de opção]

Valor por omissão: `false`

`exptisolate`, quando `true`, faz com que `isolate (expr, var)` examine expoentes de átomos (tais como `%e`) que contenham `var`.

`exptsubst` [Variável de opção]

Valor por omissão: `false`

`exptsubst`, quando `true`, permite substituições tais como `y` para `%e^x` em `%e^(a x)`.

`freeof (x_1, ..., x_n, expr)` [Função]

`freeof (x_1, expr)` Retorna `true` se nenhuma subexpressão de *expr* for igual a *x_1* ou se *x_1* ocorrer somente uma variável que não tenha associação fora da expressão *expr*, e retorna `false` de outra forma.

`freeof (x_1, ..., x_n, expr)` é equivalente a `freeof (x_1, expr) and ... and freeof (x_n, expr)`.

Os argumentos *x_1*, ..., *x_n* podem ser nomes de funções e variáveis, nomes subscritos, operadores (empacotados em aspas duplas), ou expressões gerais. `freeof` avalia seus argumentos.

`freeof` opera somente sobre *expr* como isso representa (após simplificação e avaliação) e não tenta determinar se alguma expressão equivalente pode fornecer um resultado diferente. Em particular, simplificação pode retornar uma expressão equivalente mas diferente que compreende alguns diferentes elementos da forma original de *expr*.

Uma variável é uma variável dummy em uma expressão se não tiver associação fora da expressão. Variáveis dummy recoreconhecidas através de `freeof` são o índice de um somatório ou produtório, o limite da variável em `limit`, a variável de integração na forma de integral definida de `integrate`, a variável original em `laplace`, variáveis formais em expressoes `at`, e argumentos em expressões `lambda`. Variáveis locais em `block` não são reconhecidas por `freeof` como variáveis dummy; isso é um bug.

A forma indefinida de `integrate` *not* é livre de suas variáveis de integração.

- Argumentos são nomes de funções, variáveis, nomes subscritos, operadores, e expressões. `freeof (a, b, expr)` é equivalente a `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` avalia seus argumentos.

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

- `freeof` não considera expressões equivalentes. Simplificação pode retornar uma expressão equivalente mas diferente.

```
(%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
```

- Um somatório ou uma integral definida está livre de uma variável dummy. Uma integral indefinida não é livre de suas variáveis de integração.

```
(%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*) [Função]

Retorna o factorial generalizado, definido como `x (x-z) (x - 2 z) ... (x - (y - 1) z)`. Dessa forma, para integral *x*, `genfact (x, x, 1) = x!` e `genfact (x, x/2, 2) = x!!`.

`imagpart` (*expr*) [Função]

Retorna a parte imaginária da expressão *expr*.

`imagpart` é uma função computacional, não uma função de simplificação.

Veja também `abs`, `carg`, `polarform`, `rectform`, e `realpart`.

`infix` (*op*) [Função]
`infix` (*op*, *lbp*, *rbp*) [Função]
`infix` (*op*, *lbp*, *rbp*, *lpos*, *rpos*, *pos*) [Função]

Declara *op* para ser um operador infixo. Um operador infixo é uma função de dois argumentos, com o nome da função escrito entre os argumentos. Por exemplo, o operador de subtração `-` é um operador infixo.

`infix (op)` declara *op* para ser um operador infixo com expoentes associados padrão (esquerdo e direito ambos iguais a 180) e podendo ser qualquer entre prefixado, infixado, posfixado, nário, matchfix e nofix (esquerdo e direito ambos iguais a `any`).

`infix (op, lbp, rbp)` declara *op* para ser um operador infixo com expoentes associados esquerdo e directio equilibrados e podendo ser qualquer entre prefixado, infixado, posfixado, nário, matchfix e nofix (esquerdo e direito ambos iguais a `any`).

`infix (op, lbp, rbp, lpos, rpos, pos)` declara *op* para ser um operdor infixo com expoentes associados padrão e podendo ser um entre prefixado, infixado, posfixado, nário, matchfix e nofix.

A precedência de *op* com relação a outros operadores derivam dos expoentes associados directiro e esquerdo dos operadores em questão. Se os expoentes associados esquerdo e direito de *op* forem ambos maiores que o expoente associado esquerdo e o direito de algum outro operador, então *op* tem prededência sobre o outro operador.

Se os expoentes associados não forem ambos maior ou menor, alguma relação mais complicada ocorre.

A associatividade de *op* depende de seus expoentes associados. Maior expoente associado esquerdo (*eae*) implica uma instância de *op* é avaliadas antes de outros operadores para sua esquerda em uma expressão, enquanto maior expoente associado direito (*ead*) implica uma instância de *op* é avaliada antes de outros operadores para sua direita em uma expressão. Dessa forma maior *eae* torna *op* associativo à direita, enquanto maior *ead* torna *op* associativa à esquerda. Se *eae* for igual a *ead*, *op* é associativa à esquerda.

Veja também `Syntax`.

Exemplos:

Se os expoentes associados esquerdo e direito de *op* forem ambos maiores que os expoentes associados à direita e à esquerda de algum outro operador, então *op* tem precedência sobre o outro operador.

```
(%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)
```

grande *eae* torna *op* associativa à direita, enquanto grande *ead* torna *op* associativa à esquerda.

```
(%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)
```

`inflag` [Variável de opção]

Velor padrão: `false`

Quando `inflag` for `true`, funções para extração de partes inspecionam a forma interna de `expr`.

Note que o simplificador re-organiza expressões. Dessa forma `first (x + y)` retorna `x` se `inflag` for `true` e `y` se `inflag` for `false`. (`first (y + x)` fornece os mesmos resultados.)

Também, escolhendo `inflag` para `true` e chamando `part` ou `substpart` é o mesmo que chamar `inpart` ou `substinpart`.

As funções afectadas pela posição do sinalizador `inflag` são: `part`, `substpart`, `first`, `rest`, `last`, `length`, a estrutura `for ... in`, `map`, `fullmap`, `maplist`, `reveal` e `pickapart`.

`inpart` (*expr*, *n_1*, ..., *n_k*) [Função]

É similar a `part` mas trabalha sobre a representação interna da expressão em lugar da forma de exibição e dessa forma pode ser mais rápida uma vez que nenhuma formatação é realizada. Cuidado deve ser tomado com relação à ordem de subexpressões em adições e produtos (uma vez que a ordem das variáveis na forma interna é muitas vezes diferente daquela na forma mostrada) e no manuseio com menos unário, subtração, e divisão (uma vez que esses operadores são removidos da expressão). `part (x+y, 0)` ou `inpart (x+y, 0)` retorna `+`, embora com o objectivo de referirse ao operador isso deva ser abraçado por aspas duplas. Por exemplo `... if inpart (%o9,0) = "+" then ....`

Exemplos:

```
(%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*) [Função]

Retorna *expr* com subexpressões que são adições e que não possuem *x* substituido por rótulos de expressão intermédia (esses sendo símbolos atômicos como `%t1`, `%t2`, ...). Isso é muitas vezes útil para evitar expansões desnecessárias de subexpressões que não possuam a variável de interesse. Uma vez que os rótulos intermédios são associados às subexpressões eles podem todos ser substituídos de volta por avaliação da expressão em que ocorrerem.

`exptisolate` (valor padrão: `false`) se `true` fará com que `isolate` examine expoentes de átomos (como `%e`) que contenham *x*.

`isolate_wrt_times` se `true`, então `isolate` irá também isolar com relação a produtos. Veja `isolate_wrt_times`.

Faça `example (isolate)` para exemplos.

`isolate_wrt_times` [Variável de opção]

Valor por omissão: `false`

Quando `isolate_wrt_times` for `true`, `isolate` irá também isolar com relação a produtos. E.g. compare ambas as escolhas do comutador em

```
(%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` [Variável de opção]

Valor por omissão: `false`

Quando `listconstvars` for `true`, isso fará com que `listofvars` inclua `%e`, `%pi`, `%i`, e quaisquer variáveis declaradas contantes na lista seja retornado se aparecer na expressão que chamar `listofvars`. O comportamento padrão é omitir isso.

`listdummyvars` [Variável de opção]

Valor por omissão: `true`

Quando `listdummyvars` for `false`, "variáveis dummy" na expressão não serão incluídas na lista retornada por `listofvars`. (O significado de "variável dummy" é o mesmo que em `freeof`. "Variáveis dummy" são conceitos matemáticos como o índice de um somatório ou produtório, a variável limite, e a variável da integral definida.) Exemplo:

```
(%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*) [Função]

Retorna uma lista de variáveis em *expr*.

`listconstvars` se `true` faz com que `listofvars` inclua `%e`, `%pi`, `%i`, e quaisquer variáveis declaradas constantes na lista é retornada se aparecer em *expr*. O comportamento padrão é omitir isso.

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

`lfreeof` (*lista*, *expr*) [Função]

Para cada um dos membros *m* de lista, chama `freeof (m, expr)`. Retorna `false` se qualquer chamada a `freeof` for feita e `true` de outra forma.

`lopow` (*expr*, *x*) [Função]

Retorna o menor expoente de *x* que explicitamente aparecer em *expr*. Dessa forma

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

`lpart` (*rótulo*, *expr*, *n_1*, ..., *n_k*) [Função]

é similar a `dpart` mas usa uma caixa rotulada. Uma caixa rotulada é similar à que é produzida por `dpart` mas a produzida por `lpart` tem o nome na linha do topo.

`multthru` (*expr*) [Função]

`multthru` (*expr_1*, *expr_2*) [Função]

Multiplica um factor (que pode ser uma adição) de *expr* pelos outros factores de *expr*. Isto é, *expr* é *f_1 f_2 ... f_n* onde ao menos um factor, digamos *f_i*, é uma soma de termos. Cada termo naquela soma é multiplicado por outros factores no produto. (A saber todos os factores excepto *f_i*). `multthru` não expande somas exponenciais. Essa função é o caminho mais rápido para distribuir produtos (comutativos ou não) sobre adições. Uma vez que quocientes são representados como produtos `multthru` podem ser usados para dividir adições por produtos também.

`multthru (expr_1, expr_2)` multiplica cada termo em *expr_2* (que pode ser uma adição ou uma equção) por *expr_1*. Se *expr_1* não for por si mesmo uma adição então essa forma é equivalente a `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 que isso não expande (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*) [Função]

Retorna a forma substantiva do nome da função *f*. Isso é necessário se se quer referir ao nome de uma função verbo como se esse nome fosse um substantivo. Note que algumas funções verbos irão retornar sua forma substantiva senão puderem ser avaliadas para certos argumentos. A forma substantiva é também a forma retornada se uma chamada de função é precedida por um apóstrofo.

nterms (*expr*) [Função]

Retorna o número de termos que *expr* pode ter se for completamente expandida e nenhum cancelamento ou combinação de termos acontecer. Note expressões como `sin (expr)`, `sqrt (expr)`, `exp (expr)`, etc. contam como apenas um termo independentemente de quantos termos *expr* tenha (se *expr* for uma adição).

op (*expr*) [Função]

Retorna o operador principal da expressão *expr*. `op (expr)` é equivalente a `part (expr, 0)`.

`op` retorna uma sequência de caracteres se o operador principal for uma operador interno ou definido pelo utilizador como prefixado, binário ou n-ário infixo, posfixado, matchfix ou nofix. De outra forma, se *expr* for uma expressão de função subscrita, `op` retorna uma função subscrita; nesse caso o valor de retorno não é um átomo. De outro modo, *expr* é uma função de array ou uma expressão de função comum, e `op` retorna um símbolo.

`op` observa o valor do sinalizador global `inflag`.

`op` avalia seus argumentos.

Veja também `args`.

Exemplos:

```
(%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*) [Função]
`operatorp` (*expr*, [*op_1*, ..., *op_n*]) [Função]

`operatorp (expr, op)` retorna `true` se *op* for igual ao operador de *expr*.

`operatorp (expr, [op_1, ..., op_n])` retorna `true` se algum elementos de *op_1*, ..., *op_n* for igual ao operador de *expr*.

`optimize` (*expr*) [Função]

Retorna uma expressão que produz o mesmo valor e efeito que *expr* mas faz de forma mais eficientemente por evitar a recomputação de subexpressões comuns. `optimize` também tem o mesmo efeito de "colapsar" seus argumentos de forma que todas as subexpressões comuns são compartilhadas. Faça `example (optimize)` para exemplos.

`optimprefix` [Variável de opção]

Valor por omissão: `%`

`optimprefix` é o prefixo usado para símbolos gerados pelo comando `optimize`.

`ordergreat` (*v_1*, ..., *v_n*) [Função]

Escolhe aliases para as variáveis *v_1*, ..., *v_n* tais que *v_1* > *v_2* > ... > *v_n*, e *v_n* > qualquer outra variável não mencionada como um argumento.

Veja também `orderless`.

`ordergreatp` (*expr_1*, *expr_2*) [Função]

Retorna `true` se *expr_2* precede *expr_1* na ordenação escolhida com a função `ordergreat`.

`orderless` (*v_1*, ..., *v_n*) [Função]

Escolhe aliases para as variáveis *v_1*, ..., *v_n* tais que *v_1* < *v_2* < ... < *v_n*, and *v_n* < qualquer outra variável não mencionada como um argumento.

Dessa forma a escala de ordenação completa é: constantes numéricas < constantes declaradas < escalares declarados < primeiro argumento para `orderless` < ... < último argumento para `orderless` < variáveis que começam com A < ... < variáveis que começam com Z < último argumento para `ordergreat` < ... < primeiro argumento para `ordergreat` < `mainvar`s - variáveis principais declaradas.

Veja também `ordergreat` e `mainvar`.

`orderlessp` (*expr_1*, *expr_2*) [Função]

Retorna `true` se *expr_1* precede *expr_2* na ordenação escolhida pelo comando `orderless`.

`part` (*expr*, *n_1*, ..., *n_k*) [Função]

Retorna partes da forma exibida de `expr`. Essa função obtém a parte de `expr` como especificado pelos índices *n_1*, ..., *n_k*. A primeira parte *n_1* de `expr` é obtida, então a parte *n_2* daquela é obtida, etc. O resultado é parte *n_k* de ... parte *n_2* da parte *n_1* da `expr`.

`part` pode ser usada para obter um elemento de uma lista, uma linha de uma matriz, etc.

Se o último argumento para uma função `part` for uma lista de índices então muitas subexpressões serão pinçadas, cada uma correspondendo a um índice da lista. Dessa forma `part (x + y + z, [1, 3])` é `z+x`.

`piece` mantém a última expressão seleccionada quando usando as funções `part`. Isso é escolhido durante a execução da função e dessa forma pode referir-se à função em si mesma como mostrado abaixo.

Se `partswitch` for escolhido para `true` então `end` é retornado quando uma parte seleccionada de uma expressão não existir, de outra forma uma mensagem de erro é forncecida.

Exemplo: `part (z+2*y, 2, 1)` retorna 2.

`example (part)` mostra exemplos adicionais.

`partition` (*expr*, *x*) [Função]

Retorna uma lista de duas expressões. Elas são (1) os factores de *expr* (se essa expressão for um produto), os termos de *expr* (se isso for uma adição), ou a lista (se isso for uma lsita) que não contiver `var` e, (2) os factores, termos, ou lista que faz.

```
(%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` [Variável de opção]

Valor por omissão: `false`

Quando `partswitch` for `true`, `end` é retornado quando uma parte seleccionada de uma expressão não existir, de outra forma uma mensagem de erro é fornecida.

`pickapart` (*expr*, *n*) [Função]

Atribui rótulos de expressão intermédia a subexpressões de *expr* de comprimento *n*, um inteiro. A subexpressões maiores ou menores não são atribuidos rótulos. `pickapart` retorna uma expressão em termos de expressões intermédias equivalentes à expressão original *expr*.

Veja também `part`, `dpart`, `lpart`, `inpart`, e `reveal`.

Exemplos:

```
(%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` [Variável de sistema]

Mantém a ultima expressão seleccionada quando usando funções `part`. Isso é escolhido durante a execução da função e dessa forma pode referir-se à função em si mesma.

`polarform` (*expr*) [Função]

Retorna uma expressão `r %e^(%i theta)` equivalente a *expr*, tal que `r` e `theta` sejam puramente reais.

`powers` (*expr*, *x*) [Função]

Fornece os expoentes de *x* que ocorrem em expressão *expr*.

`load ("powers")` chama essa função.

`product` (*expr*, *i*, *i_0*, *i_1*) [Função]

Representa um produto dos velores de `expr` com o índice *i* variando de *i_0* a *i_1*. A forma substantiva `'product` é mostrada como um pi maiísculo.

`product` avalia *expr* e os limites inferior e superior *i_0* e *i_1*, `product` coloca um apóstrofo (não avalia) o índice *i*.

Se os limites superiores e inferiores diferirem por um inteiro, *expr* é avaliada para cada valor do índice *i*, e o resultado um produto explícito.

de outra forma, o intervalo do índice é indefinido. Algumas regras são aplicads para simplificar o produto. Quando a variável global `simpproduct` for `true`, regras adicionais são aplicadas. Em alguns casos, simplificação um resultado que não é um produto; de outra forma, o resultado é uma forma substantiva `'product`.

Veja também `nouns` e `evflag`.

Exemplos:

```
(%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*) [Função]

Retorna a parte real de *expr*. `realpart` e `imagpart` irão trabalhar sobre expressões envolvendo funções trigonométricas e hiperbólicas, bem como raízes quadradas, logaritmos, e exponenciação.

rectform (*expr*) [Função]

Retorna uma expressão `a + b %i` equivalente a *expr*, tal que *a* e *b* sejam puramente reais.

rembox (*expr*, *unlabelled*) [Função]
rembox (*expr*, *rótulo*) [Função]
rembox (*expr*) [Função]

Remove caixas de *expr*.

`rembox (`*expr*`, unlabelled)` remove todas as caixas sem rótulos de *expr*.

`rembox (`*expr*`, `*rótulo*`)` remove somente caixas contendo *rótulo*.

`rembox (`*expr*`)` remove todas as caixas, rotuladas e nã rotuladas.

Caixas são desenhadas pelas funções `box`, `dpart`, e `lpart`.

Exemplos:

```
(%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*) [Função]

Representa um somatório dos valores de *expr* com o índice *i* variando de *i_0* a *i_1*. A forma substantiva `'sum` é mostrada com uma letra sigma maiúscula. `sum` avalia seu somando *expr* e limites inferior e superior *i_0* e *i_1*, `sum` coloca apóstrofo (não avalia) o índice *i*.

Se os limites superiores e inferiores diferirem de um número inteiro, o somatoriando *expr* é avaliado para cada valor do índice do somatório *i*, e o resultado é uma adição explícita.

De outra forma, o intervalo dos índices é indefinido. Algumas regras são aplicadas para simplificar o somatório. Quando a variável global `simpsum` for `true`, regras adicionais são aplicadas. Em alguns casos, simplificações retornam um resultado que não é um somatório; de outra forma, o resultado é uma forma substantiva `'sum`.

Quando o `evflag` (sinalizador de avaliação) `cauchysum` for `true`, um produto de somatórios é mostrado como um produto de Cauchy, no qual o índice do somatório mais interno é uma função de índice de um nível acima, em lugar de variar independentemente.

A variável global `genindex` é o prefixo alfabético usado para gerar o próximo índice do somatório, quando um índice automaticamente gerado for necessário.

`gensumnum` é o sufixo numérico usando para gerar o próximo índice do somatório, quando um índice gerado automaticamente for necessário. Quando `gensumnum` for `false`, um índice gerado automaticamente é somente `genindex` sem sufixo numérico.

Veja também `sumcontract`, `intosum`, `bashindices`, `niceindices`, `nouns`, `evflag`, e `zeilberger`.

Exemplos:

```
(%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));
Incorrect syntax: Too many )'s
else b^k, k, 1, 10))
                    ^
(%i12) linenum:11;
(%o11)                           11
(%i12) sum (integrate (x^k, x, 0, 1), k, 1, n);
                              n
                             ====
                             \       1
(%o12)                        >    -----
                             /     k + 1
                             ====
                             k = 1
(%i13) sum (if k <= 5 then a^k else b^k, k, 1, 10);
          10    9    8    7    6    5    4    3    2
(%o13)   b   + b  + b  + b  + b  + a  + a  + a  + a  + a
```

`lsum` (*expr*, *x*, *L*) [Função]

Representas a adição de *expr* a cada elemento *x* em *L*.

Uma forma substantiva `'lsum` é retornada se o argumento *L* não avaliar para uma lista.

Exemplos:

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

```
                          \      2
(%o2)                      >    i
                          /
                          ====
                                        3
                          i in rootsof(x  - 1, x)
```

`verbify` (*f*) [Função]

Retorna a forma verbal da função chamada *f*.

Veja também `verb`, `noun`, e `nounify`.

Exemplos:

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

7 Simplificação

7.1 Definições para Simplificação

`askexp` [Variável de sistema]

Quando `asksign` é chamada, `askexp` é a expressão que `asksign` está a testar.

Antigamente, era possível para um utilizador inspecionar `askexp` parando o Maxima com control-A.

`askinteger (`*expr*`, `*integer*`)` [Função]
`askinteger (`*expr*`)` [Função]
`askinteger (`*expr*`, even)` [Função]
`askinteger (`*expr*`, odd)` [Função]

`askinteger (`*expr*`, integer)` tenta determinar a partir da base de dados do `assume` se *expr* é um inteiro. Se não conseguir, `askinteger` perguntará ao utilizador, na linha de comandos, e inserirá essa informação na base de dados do `assume`, se for possível. `askinteger (`*expr*`)` é equivalente a `askinteger (`*expr*`, integer)`.

Da mesma forma, `askinteger (`*expr*`, even)` e `askinteger (`*expr*`, odd)` tentam determinar se *expr* é um inteiro par ou inteiro ímpar, respectivamente.

`asksign (`*expr*`)` [Função]

Primeiro tenta determinar se a expressão especificada é positiva, negativa, ou zero. Se isso não for possível, `asksign` perguntará ao utilizador as questões necessárias para completar a sua dedução. As respostas do utilizador serão guardadas na base de dados pelo tempo que durar a cálculo actual. O valor de retorno de `asksign` será `pos`, `neg`, ou `zero`.

`demoivre (`*expr*`)` [Função]
`demoivre` [Variável de opção]

A função `demoivre (expr)` transforma uma expressão sem modificar a variável global `demoivre`.

Quando a variável `demoivre` for `true`, as exponenciais complexas serão convertidas em expressões equivalentes em termos das funções circulares: `exp (a + b*%i)` simplifica para `%e^a * (cos(b) + %i*sin(b))` se `b` não incluir `%i`. `a` e `b` não serão expandidos.

O valor padrão de `demoivre` é `false`.

`exponentialize` converte funções circulares e hiperbólicas para a forma exponencial. `demoivre` e `exponentialize` não podem ambas serem `true` ao mesmo tempo.

`domain` [Variável de opção]

Valor por omissão: `real`

Quando a `domain` for dado o valor `complex`, `sqrt(x^2)` permanecerá `sqrt (x^2)` em lugar de retornar `abs(x)`.

`expand (`*expr*`)` [Função]
`expand (`*expr*`, `*p*`, `*n*`)` [Função]

Expande a expressão *expr*. Nos rodutos de somas e exponenciais de somas são expandidos os produtos, os numeradores de expressões racionais que incluirem somas

serão quebrados nas suas respectivas parcelas, e os produtos (comutativos e não comutativos) são distribuídos sobre as somas em todos os níveis de *expr*.

Para polinómios se pode usar frequêntemente `ratexpand` que possui um algoritmo mais eficiente.

`maxnegex` e `maxposex` controlam o máximo expoente negativo e o máximo expoente positivo, respectivamente, que irão expandir.

`expand (expr, p, n)` expande *expr*, usando *p* para `maxposex` e *n* para `maxnegex`. Isso é útil para expandir partes numa expressão mas não toda.

`expon` - o expoente da maior potência negativa que é automaticamente expandida (independente de chamadas a `expand`). Por Exemplo se `expon` for 4 então `(x+1)^(-5)` não será automaticamente expandido.

`expop` - o maior expoente positivo que é automaticamente expandido. Dessa forma `(x+1)^3`, quando digitado, será automaticamente expandido somente se `expop` for maior que ou igual a 3. Se quiser que `(x+1)^n` seja expandido onde `n` for maior que `expop`, então `expand ((x+1)^n)` funcionará unicamente se `maxposex` não for menor que `n`.

O sinalizador `expand` usado com `ev` causa expansão.

O ficheiro `simplification/facexp.mac` contém muitas funções relacionadas (em particular `facsum`, `factorfacsum` e `collectterms`, que são carregadas automaticamente) e as variáveis (`nextlayerfactor` e `facsum_combine`) que fornecem ao utilizador a possibilidade de estruturar expressões por expansão controlada. Uma descrição breve das função encontra-se no ficheiro `simplification/facexp.usg`. Há também uma demonstração disponível com o comando `demo("facexp")`.

`expandwrt (expr, x_1, ..., x_n)` [Função]

Expande a expressão `expr` com relação às variáveis *x_1*, ..., *x_n*. Todos os produtos que encvolvam as variáveis aparecerão explicitamente. O resultado estará livre de produtos de somas de expressões que não estejam livres das variáveis. *x_1*, ..., *x_n* podem ser variáveis, operadores, ou expressões.

Por omissão, os denominadores não são expandidos, mas isso pode ser controlado através da variável `expandwrt_denom`.

Esta função é carregada automaticamente a partir de `simplification/stopex.mac`.

`expandwrt_denom` [Variável de opção]

Valor por omissão: `false`

`expandwrt_denom` controla a simplificação de expressões racionais feita por `expandwrt`. Se tiver valor `true`, então tanto o numerador como o denominador da expressão serão expandidos conforme os argumentos de `expandwrt`, mas se `expandwrt_denom` for `false`, então somente o numerador será expandido.

`expandwrt_factored (expr, x_1, ..., x_n)` [Função]

é similar a `expandwrt`, mas trata os produtos numa forma diferente. `expandwrt_factored` expande somente sobre esses factores de `expr` que contiverem as variáveis *x_1*, ..., *x_n*.

Esta função é carregada automaticamente a partir de `simplification/stopex.mac`.

`expon` [Variável de opção]

Valor por omissão: 0

`expon` é o expoente da maior potência negativa que é automaticamente expandido (independente de chamadas a `expand`). Por exemplo, se `expon` for 4 então `(x+1)^(-5)` não será automaticamente expandido.

`exponentialize (`*expr*`)` [Função]
`exponentialize` [Variável de opção]

A função `exponentialize (expr)` converte as funções circulares e hiperbólicas em *expr* para exponenciais, sem modificar a variável global `exponentialize`.

Quando a variável `exponentialize` for `true`, todas as funções circulares e hiperbólicas são convertidas para a forma exponencial. O valor por omissão é `false`.

`demoivre` converte exponenciais complexas em funções circulares. `exponentialize` e `demoivre` não podem ambas serem `true` ao mesmo tempo.

`expop` [Variável de opção]

Valor por omissão: 0

`expop` - o maior expoente positivo que é automaticamente expandido. Dessa forma `(x+1)^3`, será automaticamente expandido somente se `expop` for maior que ou igual a 3. Se quiser que `(x+1)^n` seja expandido onde `n` for maior que `expop`, então `expand ((x+1)^n)` funcionará somente se `maxposex` não for menor que `n`.

`factlim` [Variável de opção]

Valor por omissão: -1

`factlim` especifica o maior factorial que é automaticamente expandido. Se for -1 então todos os inteiros são expandidos.

`intosum (`*expr*`)` [Função]

Move factores multiplicativos fora de um somatório para dentro. Se um índice for usado na expressão de fora, então a função tentará achar um índice razoável, o mesmo que é feito para `sumcontract`. Isto é essencialmente a ideia inversa da propriedade `outative` de somatórios, mas repare que não elimina essa propriedade, apenas faz com que seja ignorada.

Em alguns casos, poderá ser necessário um `scanmap(multthru,`*expr*`)` antes de `intosum`.

`lassociative` [Declaração]

`declare (g, lassociative)` diz ao simplificador do Maxima que `g` é associativa à esquerda. E.g., `g (g (a, b), g (c, d))` irá simplificar para `g (g (g (a, b), c), d)`.

`linear` [Declaração]

Uma das propriedades operativas do Maxima. As funções de uma única variável `f` assim declaradas fazem com que a expressão `f(x + y)` seja expandida em `f(x) + f(y)`, a expressão `f(a*x)` transforma-se em `a*f(x)` se `a` for uma constante. Para funções de dois ou mais argumentos, a linearidade define-se igual que no caso de `sum` ou `integrate`, isto é, `f (a*x + b, x)` retorna `a*f(x,x) + b*f(1,x)`, se `a` e `b` forem independentesx de `x`.

`linear` é equivalente a `additive` e `outative`. Veja também `opproperties`.

mainvar [Declaração]

Permite declarar variáveis do tipo mainvar (variável principal). A escala de ordenação para átomos é essencialmente: números < constantes (e.g., %e, %pi) < escalares < outras variáveis < mainvars. Por exemplo, compare expand ((X+Y)^4) com (declare (x, mainvar), expand ((x+y)^4)). (Nota: este recurso deverá ser usado com cautela. Por exemplo, se subtrair uma expressão, na qual x for uma mainvar, da mesma expressão, mas onde x não for mainvar, poderá precisar de resimplificação, por exemplo, com ev (expr, simp), para que sejam canceladas. Também, se grava uma expressão na qual x for uma mainvar, provavelmente deverá também gravar x.)

maxapplydepth [Variável de opção]

Valor por omissão: 10000

maxapplydepth é a profundidade máxima ate a qual apply1 e apply2 deverão descer.

maxapplyheight [Variável de opção]

Valor por omissão: 10000

maxapplyheight é nível máximo a ser atingido por applyb1 antes de abandonar.

maxnegex [Variável de opção]

Valor por omissão: 1000

maxnegex é o maior expoente negativo que será expandido pelo comando expand (veja também maxposex).

maxposex [Variável de opção]

Valor por omissão: 1000

maxposex é o maior expoente que será expandido com o comando expand (veja também maxnegex).

multiplicative [Declaração]

declare (f, multiplicative) diz ao simplificador do Maxima que f é multiplicativa.

1. Se f for uma função de uma única variável, sempre que o simplificador encontrar f aplicada a um produto, f será distribuida nesse produto. Por exemplo, f(x*y) simplifica para f(x)*f(y).
2. Se f for uma função de 2 ou mais argumentos, a multiplicatividade entende-se como multiplicatividade no primeiro argumento de f. Por exemplo, f (g(x) * h(x), x) simplifica para f (g(x) ,x) * f (h(x), x).

Esta simplificação não é feita quando f for aplicada a expressões da forma product (x[i], i, m, n).

negdistrib [Variável de opção]

Valor por omissão: true

Quando negdistrib for true, -1 distribue sobre uma expressão. Por exemplo, -(x + y) transforma-se em - y - x. Mudando o valor de negdistrib para false permitirá que - (x + y) seja mostrado como foi escrito. Embora isso possa ser útil, tenha muito cuidado: esta variável e a variável simp não deveriam ser escolhidas sempre como false, excepto em forma local no seu Maxima.

`negsumdispflag` [Variável de opção]

Valor por omissão: `true`

Quando `negsumdispflag` for `true`, `x - y` é mostrado como `x - y` em lugar de como `- y + x`. Mudando para `false` faz com que não seja feita a verificação especial para a apresentação da diferença entre duas expressões. Uma aplicação é para que `a + %i*b` e `a - %i*b` sejam mostrados na mesma forma.

`noeval` [Símbolo especial]

`noeval` suprime a fase de avaliação de `ev`. Isso é útil conjuntamente com outras condições e para fazer com que expressões sejam simplificadas sem serem reavaliadas.

`noun` [Declaração]

`noun` é uma das opções do comando `declare`. Faz com que as funções assim declaradas sejam substantivos (noun), implicando que não sejam avaliadas automaticamente.

`noundisp` [Variável de opção]

Valor por omissão: `false`

Quando `noundisp` for `true`, os substantivos (nouns) são mostrados com um apóstrofo. Sempre que se mostra a definição de uma função, essa variável é igual a `true`.

`nouns` [Símbolo especial]

`nouns` é um `evflag` (sinalizador de avaliação). Quando usado como uma opção para o comando `ev`, `nouns` converte todas as formas substantivas (noun), na expressão a ser avaliada, para verbos (`"verbs"`), isto é, avalia essas expressões. Veja também `noun`, `nounify`, `verb`, e `verbify`.

`numer` [Símbolo especial]

`numer` faz com que algumas funções matemáticas (incluindo exponenciação) com argumentos numéricos sejam avaliadas em ponto flutuante. Isto faz com que variáveis em `expr` que tiverem valores numéricos sejam substituídas pelos seus valores correspondentes. `numer` também activa a opção `float`.

`numerval` (*x_1, expr_1, ..., var_n, expr_n*) [Função]

Declara as variáveis `x_1`, ..., *x_n* com valores numéricos iguais a `expr_1`, ..., `expr_n`. O valor numérico é avaliado e substituido para a variável em quaisquer expressões em que a variável aparecer, se o sinalizador `numer` for igual a `true`. Veja também `ev`.

As expressões `expr_1`, ..., `expr_n` podem ser quaisquer, não necessariamente numéricas.

`opproperties` [Variável de sistema]

`opproperties` é a lista de propriedades de operadores especiais reconhecidas pelo simplificador do Maxima: `linear`, `additive`, `multiplicative`, `outative`, `evenfun`, `oddfun`, `commutative`, `symmetric`, `antisymmetric`, `nary`, `lassociative`, `rassociative`.

`opsubst` [Variável de opção]

Valor por omissão: `true`

Quando `opsubst` for `false`, `subst` não tenta substituir dentro de um operador de uma expressão. Por exemplo, `(opsubst: false, subst (x^2, r, r+r[0]))`.

outative [Declaração]

`declare (f, outative)` diz ao simplificador do Maxima que factores constantes no argumento de `f` podem ser puxados para fora.

1. Se `f` for uma função de uma única variável, sempre que o simplificador encontrar `f` aplicada a um produto, os factores que forem constantes nesse produto serão puxados para fora. Por exemplo, `f(a*x)` simplificará para `a*f(x)` se `a` for uma constante. Factores de constantes não atômicas não serão puxados para fora.
2. Se `f` for uma função de 2 ou mais argumentos, a colocação para fora é definida como no caso de `sum` ou `integrate`, isto é, `f (a*g(x), x)` irá simplificar para `a * f(g(x), x)` se `a` não depender de `x`.

`sum`, `integrate`, e `limit` são todas do tipo `outative`.

posfun [Declaração]

`declare (f, posfun)` declara `f` como função positiva. `is (f(x) > 0)` retorna `true`.

radcan (*expr*) [Função]

Simplifica *expr*, que pode conter logaritmos, exponenciais, e radicais, convertendo essa expressão numa forma canónica sobre uma ampla classe de expressões e com uma dada ordenação de variáveis; isto é, todas as formas funcionalmente equivalentes são mapeadas numa única forma. Para uma classe ampla de expressões, `radcan` produz uma forma regular. Duas expressões equivalentes nessa classe não possuem necessáriamente a mesma aparência, mas as suas diferenças podem ser simplificadas por `radcan` para zero.

Para algumas expressões `radcan` demora muito tempo. Esse é o custo de explorar as realções entre as componentes da expressão para simplificar expoentes usando factorização e expansão em frações parciais.

Quando `%e_to_numlog` for `true`, `%e^(r*log(expr))` simplifica para `expr^r` se `r` for um número racional.

Quando `radexpand` for `false`, certas transformações são inibidas. `radcan (sqrt (1-x))` permanece `sqrt (1-x)` e não é simplificada para `%i sqrt (x-1)`. `radcan (sqrt (x^2 - 2*x + 11))` permanece `sqrt (x^2 - 2*x + 1)` e não é simplificada para `x - 1`.

`example (radcan)` mostra alguns exemplos.

radexpand [Variável de opção]

Valor por omissão: `true`

`radexpand` controla algumas simplificações de radicais.

Quando `radexpand` for `all`, todos os factores que forem potências de ordem n, dentro de uma raiz de ordem n, serão puxados para fora do radical. Por exemplo, se `radexpand` for `all`, `sqrt (16*x^2)` simplifica para `4*x`.

Mais particularmente, considere `sqrt (x^2)`.

- Se `radexpand` for `all` ou `assume (x > 0)` tiver sido executado, `sqrt(x^2)` simplifica para `x`.
- Se `radexpand` for `true` e `domain` for `real` (valores usados por omissão), `sqrt(x^2)` simplifica para `abs(x)`.

- Se `radexpand` for `false`, ou `radexpand` for `true` e `domain` for `complex`, `sqrt(x^2)` não é simplificado.

Note que, neste exemplo, `domain` somente interessa quando `radexpand` for `true`.

`radsubstflag` [Variável de opção]

Valor por omissão: `false`

Se `radsubstflag` for `true`, permite a `ratsubst` fazer substituições tais como `u` por `sqrt (x)` em `x`.

`rassociative` [Declaração]

`declare (g, rassociative)` diz ao simplificador do Maxima que `g` é associativa à direita, isto é, `g(g(a, b), g(c, d))` simplifica para `g(a, g(b, g(c, d)))`.

`scsimp` (*expr*, *rule_1*, ..., *rule_n*) [Função]

Simplificação Sequêncial Comparativa (método devido a Stoute). `scsimp` tenta simplificar *expr* conforme as regras *rule_1*, ..., *rule_n*. Se uma expressão pequena for obtida, o processo repete-se. De outra forma após todas as simplificações serem tentadas, `scsimp` retorna a resposta original.

`example (scsimp)` mostra alguns exemplos.

`simpsum` [Variável de opção]

Valor por omissão: `false`

Quando `simpsum` for `true`, o resultado de um comando `sum` é simplificado. Essa simplificação pode algumas vezes produzir uma forma fechada. Se `simpsum` for `false`, ou se a forma com apóstrofo `'sum` for usada, o valor é uma forma substantiva aditiva que é uma representação da notação sigma usada em matemática.

`sumcontract` (*expr*) [Função]

Combina vários somatórios que possuem limites superiores e inferiores que diferem por constantes. O resultado é uma expressão que contém apenas um somatório mais todos os termos adicionais que tiveram de ser extraídos para obter essa forma. `sumcontract` combina todas as somas compatíveis e usa os indices de uma das somas, se puder, ou tenta formar um índice razoável se não poder usar nenhum dos que foram fornecidos.

Poderá ser necessário usar `intosum (expr)` antes de `sumcontract`.

`sumexpand` [Variável de opção]

Valor por omissão: `false`

Quando `sumexpand` for `true`, produtos de somas e somas exponeciadas simplificam para somas aninhadas.

Veja também `cauchysum`.

Exemplos:

```
(%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` [Variável de opção]

Valor por omissão: `true`

Quando `sumsplitfact` for `false`, `minfactorial` é aplicado após `factcomb`.

`symmetric` [Declaração]

`declare (h, symmetric)` diz ao simplificador do Maxima que `h` é uma função simétrica. Nomeadamente, `h (x, z, y)` simplifica para `h (x, y, z)`.

`commutative` é sinônimo de `symmetric`.

`unknown` (*expr*) [Função]

Retorna `true` se e somente se *expr* contém um operador ou função não reconhecida pelo simplificador do Maxima.

8 Criação de Gráficos

8.1 Definições para Criação de Gráficos

`in_netmath` [Variável]

Valor por omissão: `false`

Quando `in_netmath` é `true`, `plot3d` imprime uma saída OpenMath para a consola se `plot_format` é `openmath`; caso contrário `in_netmath` (mesmo se for `true`) não tem efeito. `in_netmath` não tem efeito sobre `plot2d`.

`plot2d` (*expr*, *intervalo_x*, ..., *opções*, ...) [Função]
`plot2d` ([*expr_1*, ..., *expr_n*], ..., *opções*, ...) [Função]
`plot2d` ([*expr_1*, ..., *expr_n*], *intervalo_x*,..., *opções*, ...) [Função]

Onde *expr*, *expr_1*, ..., *expr_n* podem ser expressões, funções ou operadores do Maxima ou do Lisp, ou ainda uma lista da forma `[discrete, [x1, ..., xn], [y1, ..., yn]]`, `[discrete, [[x1, y1], ..., [xn, ..., yn]]` ou `[parametric, expr_x, expr_y, intervalo_t]`.

Mostra o gráfico de uma ou mais expressões em função de uma variável.

`plot2d` produz o gráfico de uma expressão *expr* ou de várias expressões `[expr_1, ..., expr_n]`. As expressões que não forem do tipo paramétrico ou discreto, deverão depender todas de uma única variável *var* e será obrigatório usar *intervalo_x* para indicar o nome dessa variável, e os seus valores mínimo e máximo, usando a sintaxe: `[var, min, max]`. O gráfico mostrará o eixo horizontal delimitado pelos valores *min* e *max*.

Uma expressão a ser representada no gráfico pode ser dada também na forma discreta, ou paramétrica. Nomeadamente, por meio de uma lista a começar pela palavra "discrete" ou "parametric". A palavra chave *discrete* deverá ir seguida por duas listas, ambas do mesmo comprimento, que serão as coordenadas horizontais e verticais de um conjunto de pontos; em alternativa, as coordenadas de cada ponto podem ser colocadas numa lista de dois valores, e todas essas coordenadas deverão estar dentro de outra lista. A palavra chave *parametric* deverá ir seguida por duas expressões *expr_x* e *expr_y*, e um intervalo *intervalo_t* da forma `[param, min, max]`. As duas expressões deverão depender unicamente no parâmetro *param*, e o gráfico mostrará o percurso seguido pelo ponto com coordenadas (*expr_x*, *expr_y*) à medida que *param* aumenta desde *min* até *max*.

O intervalo de valores no eixo vertical não é obrigatório. É mais uma das opções do comando, com a sintaxe: `[y, min, max]`. Se essa opção for usada, o gráfico apresentará esse intervalo completo, inclusivamente quando as expressões não cheguem a atingir esses valores. De outra forma, se não for indicado um intervalo no eixo vertical por meio de `set_plot_option`, as fronteiras do eixo vertical serão seleccionadas automaticamente.

Todas as outras opções deverão ser listas, a começar pelo nome da opção. A opção *xlabel* pode ser usada para dar um texto que identificará o eixo horizontal; se essa opção não for usada, o eixo será identificado com o nome da variável indicada em *intervalo_x*, ou com a expressão *expr_x*, se houver unicamente uma expressão paramétrica, ou caso contrário ficará em branco.

O texto para identificar o eixo vertical pode ser indicado com a opção *ylabel*. Se só houver uma única expressão a ser representada, e a opção *ylabel* não tiver sido usada, o eixo vertical será identificado com essa expressão, a menos que for muito comprido, ou com a expressão *expr_y*, se a expressão for paramétrica, ou com o texto "discrete data" se a expressão for discreta.

As opções *logx* e *logy* não precisam de quaisquer paraâmetros. Fazem com que os eixos horizontal e vertical sejam apresentados em forma logarítmica.

Se houver vá rias expressões a serem representadas, será escrita uma legenda para identificar cada uma dessas expressões. O texto que deverá ser usado nessa legenda pode ser indicado por meio da opção *legend*. Se essa opção não for usada, Maxima criará textos para identificar cada expressão.

Por omissão, as expressões dadas serão representadas por pequenos segmentos de recta a ligarem pontos adjacentes num conjunto de pontos que, ou é dado usando a forma *discrete*, ou é calculado automaticamente a partir das expressões dadas, por meio de um algoritmo com ajuste automático dos intervalos entre pontos, usando como estimativa inicial do número de pontos o valor indicado pela opção *nticks*. A opção *style* serve para fazer com que alguma das expressões seja representada por pontos isolados ou por pontos mais segmentos de recta.

Existem várias opções globais, armazenadas na lista *plot_options*, quu podem ser modificadas usando a função `set_plot_option`; qualquer uma dessad opções pode ser contrariada pelos valores locais dados no comando *plot2d*.

Uma função a ser representada poderá ser identificada pelo nome de uma função ou operador do Maxima ou do Lisp, por meio duma expressão lambda do Maxima, ou como uma expressão geral do Maxima. Se for especificada como um nome ou como expressão lambda, a respectiva função deverá depender dum único argumento.

Exemplos:

Gráficos de funções ordinárias.

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

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

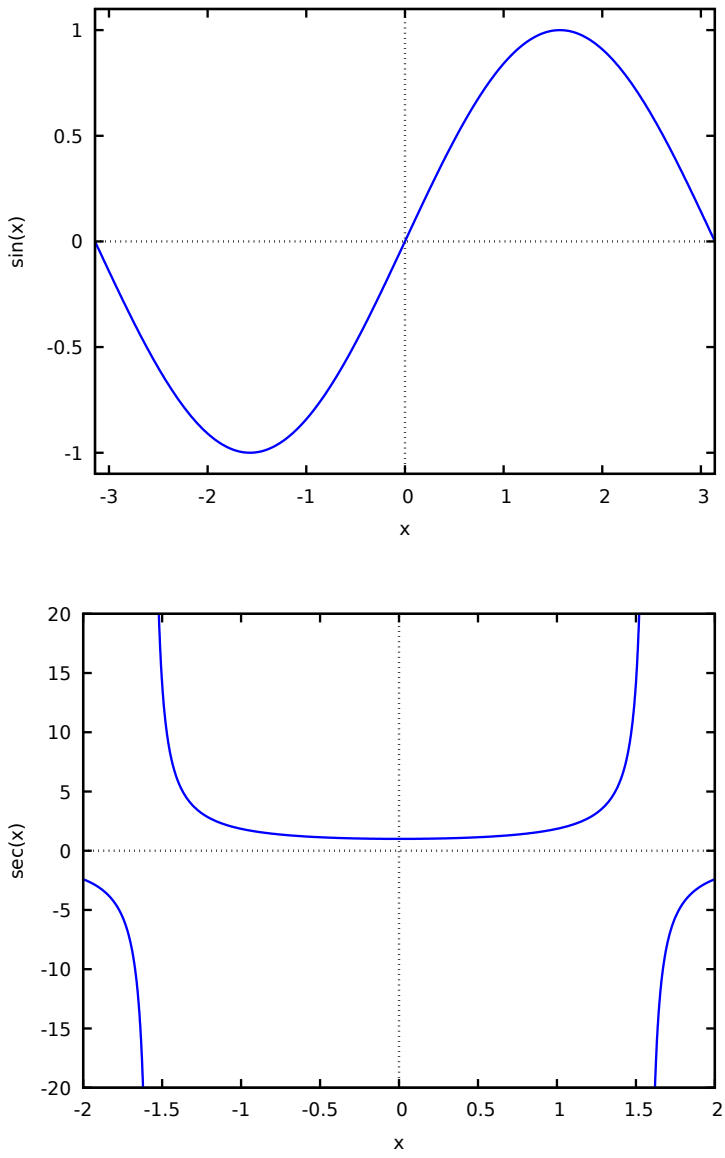
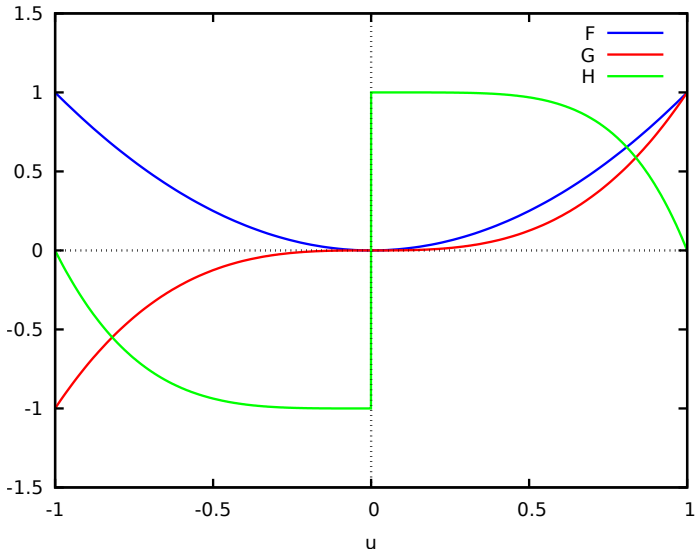


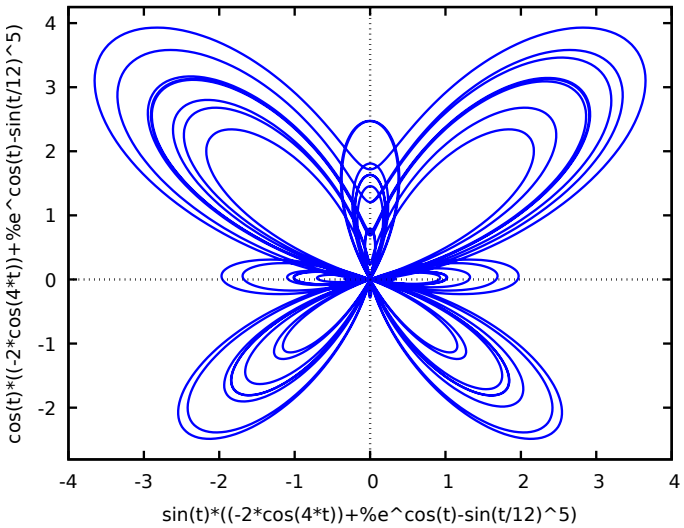
Gráfico de funções identificadas pelo seu nome.

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



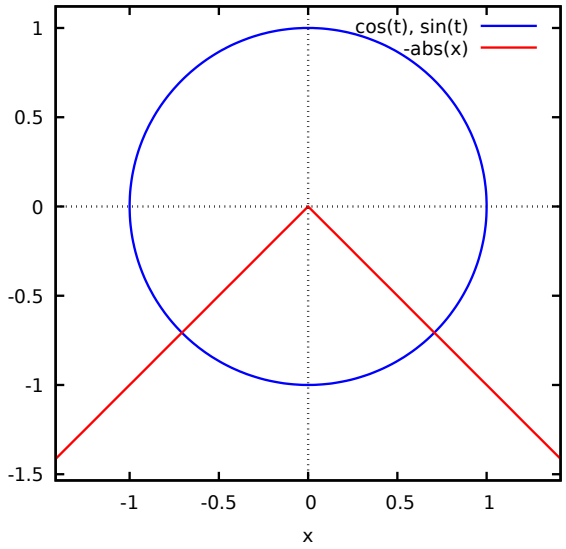
A curva "borboleta", definida paramétricamente:

```
(%i1) r: (exp(cos(t))-2*cos(4*t)-sin(t/12)^5)$
(%i2) plot2d([parametric, r*sin(t), r*cos(t), [t, -8*%pi, 8*%pi]])$
```



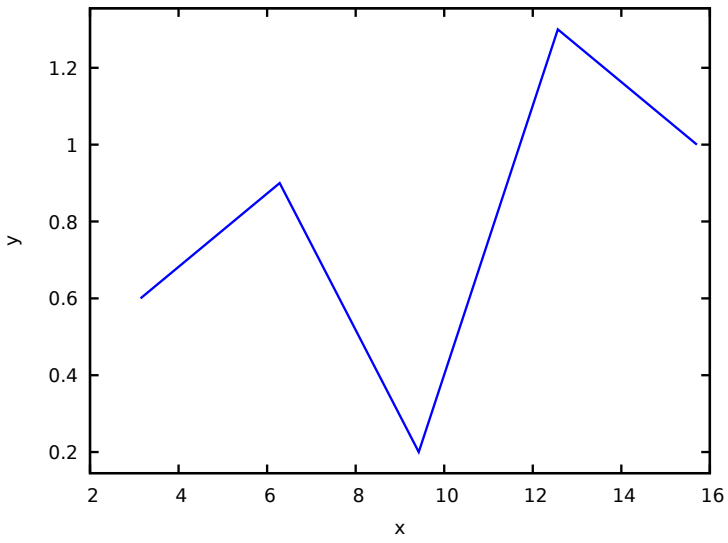
Função -|x| e círculo por meio de um gráfico paramétrico com um parâmetro t. Usa-se a opção *same_xy* para obter a mesma escala nos dois eixos:

```
(%i1) plot2d([[parametric, cos(t), sin(t), [t,0,2*%pi]], -abs(x)],
         [x, -sqrt(2), sqrt(2)], same_xy)$
```



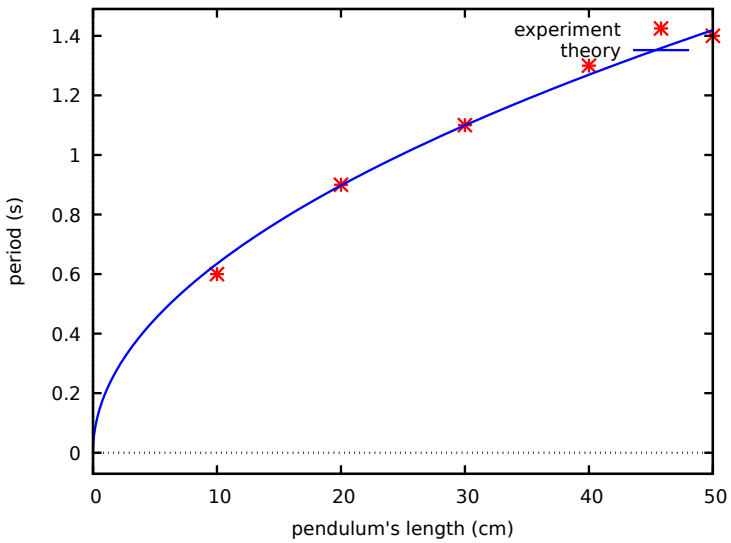
Gráficos de um conjunto discreto de pontos, definindo as coordenadas x e y por separado:

```
(%i1) plot2d ([discrete, makelist(i*%pi, i, 1, 5),
                                [0.6, 0.9, 0.2, 1.3, 1]])$
```



O gráfico dos pontos dos dados pode ser apresentado junto com o gráfico de uma função teórica que ajusta esses valores:

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



Veja também `plot_options`, que descreve as opções das funções gráficas e mostra mais exemplos.

`plot_options` [Variável de sistema]

Os elementos desta lista estabelecem os valores por omissaão para as opções usadas na elaboração de gráficos. Se uma opção estiver presente numa chamada a `plot2d` ou `plot3d`, esse valor terá precedência sobre o valor por omissção. De outra forma, será usado o valor em `plot_options`. Os valores por omissção das opções podem ser modificados usando `set_plot_option`.

Cada elemento de `plot_options` é uma lista de dois ou mais ítens. O primeiro item é o nome de uma opção, e os restantes compreendem o valor ou valores atribuídos à opção. Em alguns casos, o valor atribuído é uma lista, que pode compreender muitos itens.

As opções globais que são reconhecidas por `plot2d` e `plot3d` são as seguintes:

- Opção: `plot_format`

  Determina a interface gráfica que será usada por `plot2d` e `plot3d`.

  - Valor por omissão: `gnuplot`

    Gnuplot é o pcote gráfico mais avançado entre os disponíveis no Maxima. Será preciso que o pacote externo gnuplot esteja instalado.

  - Valor: `mgnuplot`

    Mgnuplot é uma interface Tk para o gnuplot. Vem incluída na distribuíção do Maxima. Mgnuplot oferece uma GUI rudimentar para o gnuplot, mas tem menos recursos em geral que a interface padrão do gnuplot. Mgnuplot precisa que os pacotes externos gnuplot e Tcl/Tk estejam instalados.

  - Valor: `openmath`

    Openmath é um programa gráfico escrito em Tcl/Tk. Este formato é fornecido pelo pacote Xmaxima, que é distribuido com Maxima. Se quiser usar este formato instalar o pacote Xmaxima, que funcionará não só a partir do próprio Xmaxima mas também a partir da linha de comandos de outras GUI para o Maxima.

- Opção: `run_viewer`

  Controla se será executado ou não o visualizador apropriado para o formato do gráfico.

  - Valor por omissão: `true`

    Executa-se o programa visualizador.

  - Valor: `false`

    Não se executa o programa visualizador.

- Opção: `y`

  O intervalo vertical do gráfico.

  Exemplo:

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

  Faz com que o intervalo vertical seja [-3, 3].

- Opção: `plot_realpart`

  Quando `plot_realpart` for `true`, nos pontos onde o valor a ser representado no eixo vertical for complexo, será apresentada a sua parte real x; isso é equivalente a mostrar `realpart(x)` em lugar de x. De outra forma, somente valores com a parte imaginária igual a 0 são mostrados no gráfico, e os valores complexos serão ignorados.

  Exemplo:

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

  O valor por omissão é `false`.

- Opção: `nticks`

  No plot2d, é o número de pontos usados, inicialmente, pela rotina gráfica adaptativa. É também o número de pontos que serão apresentados num gráfico paramétrico.

  Exemplo:

  ```
  [nticks, 20]
  ```

  O valor por omissão para `nticks` é 10.

- Opção: `adapt_depth`

  O número maximo de subdivisões usadas pela rotina gráfica adaptativa.

  Exemplo:

  ```
  [adapt_depth, 5]
  ```

  O valor por omissão para `adapt_depth` é 10.

- Opção: `xlabel`

  O texto que identifica o eixo horizontal num gráfico a 2d.

  Exemplo:

  ```
  [xlabel, "Tempo em segundos"]
  ```

- Opção: `ylabel`

  O texto que identifica o eixo vertical num gráfico a 2d.

Exemplo:

```
[ylabel, "Temperatura"]
```

- Opção: `logx`

  Faz com que o eixo horizontal num gráfico a 2d seja representado em escala logarítmica. Não precisa de nenhum parâmetro adicional.

- Opção: `logy`

  Faz com que o eixo vertical num gráfico a 2d seja representado em escala logarítmica. Não precisa de nenhum parâmetro adicional.

- Opção: `legend`

  Os textos para identificar as diversas expressões num gráfico a 2d com muitas expressões. Se existirem mais expressões do que os textos dados, serão repetidos. Por omissão, seraão usados os nomes das expressões ou das funções, ou as palavras discrete1, discrete2, ..., no caso de conjuntos discretos de pontos.

  Exemplo:

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

- Opção: `style`

  Os estilos que serão usados para as diversas funções ou conjuntos discretos de pontos, num gráfico a 2d. A palavra *style* deverá ir seguida por um ou mais estilos. Se houver mais funções e conjuntos de dados do que os estilos definidos, serão repetidos estilos. Cada estilo poderá ser *lines* para segmentos de recta, *points* para pontos isolados, *linespoints* para segmentos e pontos, ou *dots* para pequenos pontos isolados. O Gnuplot também aceita o estilo *impulses*.

  Cada um dos estilos poderá ser incorporado numa lista, seguido de alguns parâmetros adicionais. *lines* admite um ou dois números: a largura da linha e um inteiro que identifica uma cor. *points* admite um ou dois números; o primeiro número é o raio dos pontos, e o segundo número é um inteiro que no Gnuplot permite seleccionar diferentes formas e cores para os pontos e no Openmath muda a cor dos pontos usados. *linesdots* admite até quatro números; os dois primeiros são os mesmos do que para *lines* e os dois últimos são os mesmos do que para *points*.

  Exemplo:

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

  No Gnuplot, isso faz com que a primeira (e terceira, quinta, etc) expressão seja apresentada com segmentos de recta azuis de largura 2, e a segunda (quarta, sexta, etc) expressão com quadrados verdes de tamanho 1. No Openmath, a primeira expressão será apresentada com rectas magenta de largura 2, e a segunda com pontos laranja de raio 1; repare que openmath_color(3) e openmath_color(4) produzem “magenta” e “orange”.

  O estilo por omissão é segmentos de recta, com largura 1, e com diferentes cores.

- Opção: `grid`

  Define o número de pontos nas direções x e y, na grelha usada nos gráficos tridimensionais.

Exemplo:

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

Define uma grelha de 50 por 50 pontos. A grelha padrão é 30 por 30.

- Opção: `transform_xy`

  Permite a aplicação de transformações nos gráficos tridimensionais.

  Exemplo:

  ```
  [transform_xy, false]
  ```

  O valor por omiss˜ao de `transform_xy` é `false`. Se não for `false`, deverá ser o resultado produzido por

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

  A transformação `polar_xy` está previamente definida no Maxima. É igual ao resultado da transformação

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

Opções do Gnuplot:

Existem muitas opções específicas para o Gnuplot. Muitas dessas opções são comandos próprios do Gnuplot, especificados como sequências de caracteres. Consulte a documentação do gnuplot para mais pormenores.

- `gnuplot_term`

  Define o tipo terminal de saída para gnuplot.

  - Valor por omissão: `default`

    A saída do Gnuplot é mostrada em uma janela gráfica separada.

  - Valor: `dumb`

    A saída do Gnuplot é mostrada na consola do Maxima, usando uma aproximação "arte ASCII" para gráficos.

  - Valor: `ps`

    Gnuplot gera comandos na linguagem PostScript de descrição de páginas. Se à opção `gnuplot_out_file` tiver sido dada o nome de um ficheiro, gnuplot escreverá os comandos PostScript nesse ficheiro. De outra forma, os comandos PostScript serão gravados no ficheiro `maxplot.ps`.

  - Valor: qualquer outro tipo de terminal aceite pelo gnuplot

    Gnuplot pode produzir gráficos em muitos outros formatos gráficos tais como png, jpeg, svg, etc. Para criar gráficos em algum desses deverá dar-se a `gnuplot_term` um (símbolo) suportado pelo gnuplot ou uma especificação completa de terminal do gnuplot com opções válidas (sequência de caracteres). Por exemplo `[gnuplot_term,png]` cria gráficos no formato PNG (Portable Network Graphics) enquanto `[gnuplot_term,"png size 1000,1000"]` cria gráficos no formato PNG com tamanho de 1000x1000 pixels. Se à opção `gnuplot_out_file` for dado o nome de um ficheiro, gnuplot gravará o graáfico nesse ficheiro. De outra forma, o gráfico é gravado no ficheiro `maxplot.`*`term`*, onde *term* é o nome do terminal do gnuplot.

- Opção: `gnuplot_out_file`

  Grava o gráfico criado por gnuplot para um ficheiro.

  - Valor por omissão: `false`

    Nenhum ficheiro de saída especificado.

  - Valor: *filename*

    Exemplo: `[gnuplot_out_file, "myplot.ps"]` Quando usada em conjunto com o terminal PostScript do gnuplot, neste exemplo o gráfico será gravado em formato PostScript no ficheiro `myplot.ps`, .

- Opção: `gnuplot_pm3d`

  Controla o uso do modo PM3D, que possui recursos avançados em 3D. O modo PM3D está somente disponível nas versões de gnuplot posteriores a 3.7. O valor padrão para `gnuplot_pm3d` é `false`.

  Exemplo:

  ```
  [gnuplot_pm3d, true]
  ```

- Opção: `gnuplot_preamble`

  Insere comandos antes que o gráfico seja desenhado. Quaisquer comandos válidos para o gnuplot podem ser usados. Multiplos comandos podem ser separados com um ponto e vírgula. O exemplo mostrado produz uma escala logarítmica no gráfico. O valor padrão para `gnuplot_preamble` é uma sequência de caracteres vazia `""`.

  Exemplo:

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

- Opção: `gnuplot_curve_titles`

  Controla os títulos dados na legenda do gráfico. O valor padrão é `[default]`, que escolhe automaticamente um título para função cujo gráfico está a ser desenhado. Se não for igual a `[default]`, `gnuplot_curve_titles` poderá conter uma lista de sequências de caracteres, cada uma das quais é `"title '`*`nome`*`'"`. (Para desabilitar a legenda do gráfico, adicione `"set nokey"` a `gnuplot_preamble`.)

  Exemplo:

  ```
  [gnuplot_curve_titles,
  ["title 'Minha primeira função'", "title 'Minha segunda função'"]]
  ```

- Opção: `gnuplot_curve_styles`

  Uma lista de sequências de caracteres a ser enviada para o gnuplot para controlar a aparência das curvas, nomeadamente, cor, largura, brilho, etc. O valor padrão é `["with lines 3", "with lines 1", "with lines 2", "with lines 5", "with lines 4", "with lines 6", "with lines 7"]`, que circula através de diferentes cores. Consulte a documentação de `plot` no manual do gnuplot para mais informações.

  Exemplo:

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

- Opção: `gnuplot_default_term_command`

  O comando do Gnuplot para escolher o tipo de terminal gráfico. O valor padrão é a sequência de caracteres vazia `""`, nomeadamente, usar-se-á o formato padrão do gnuplot.

  Exemplo:

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

- Opção: `gnuplot_dumb_term_command`

  O comando gnuplot para escolher o tipo de terminal não gráfico. O valor padrão é `"set term dumb 79 22"`, que produz saída em texto com 79 por 22 caracteres.

  Exemplo:

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

- Opção: `gnuplot_ps_term_command`

  O comando gnuplot para escolher o tipo de terminal para o terminal PostScript. O valor padrão é `"set size 1.5, 1.5;set term postscript eps enhanced color solid 24"`, que escolhe o tamanho para 1.5 vezes o padrão do gnuplot, e o tamanho da fonte para 24, além de outras coisas. Para mais informação, consulte a documentação de `set term postscript` no manual do gnuplot.

  Exemplo:

  Toda as figuras nos exemplos para a função *plot2d* neste manual forma obtidas a partir de ficheiros Postscript que foram produzidos após ter mudado `gnuplot_ps_term_command` par:

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

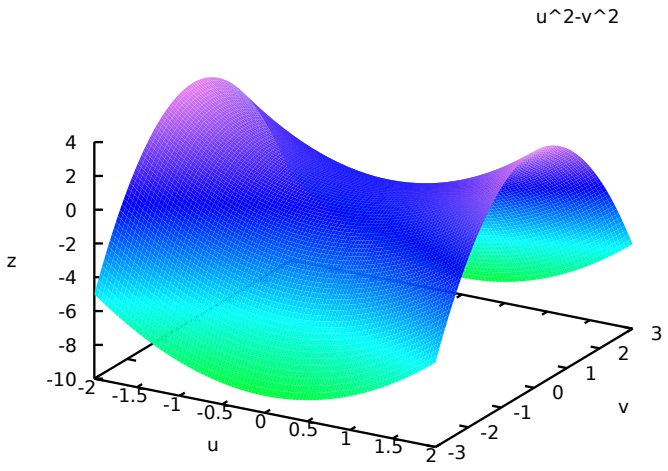
`plot3d` ([*expr_1*, *expr_2*, *expr_3*], *x_range*, *y_range*, ..., *opções*, ...) [Função]
`plot3d` (*expr*, *x_range*, *y_range*, ..., *opções*, ...) [Função]
`plot3d` (*name*, *x_range*, *y_range*, ..., *opções*, ...) [Função]
`plot3d` ([*expr_1*, *expr_2*, *expr_3*], *x_rge*, *y_rge*) [Função]
`plot3d` ([*nome_1*, *nome_2*, *nome_3*], *x_range*, *y_range*, ..., *opções*, ...) [Função]

Mostra o gráfico de uma ou três expressões como funções de duas variáveis.

Exemplos:

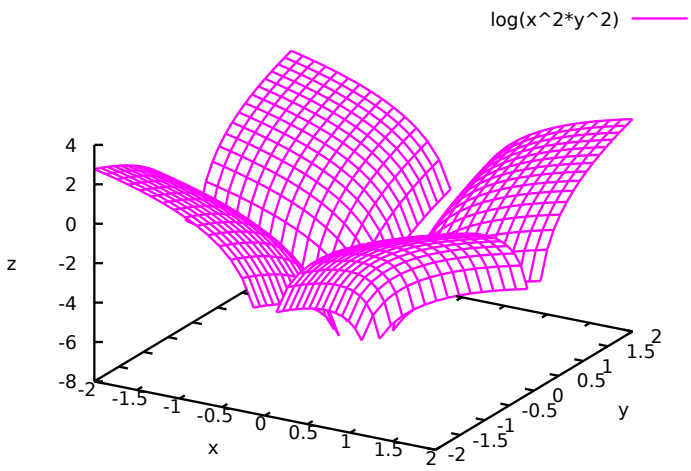
Função de duas variáveis:

```
(%i1) plot3d (u^2 - v^2, [u, -2, 2], [v, -3, 3], [grid, 100, 100],
        [mesh_lines_color,false])$
```



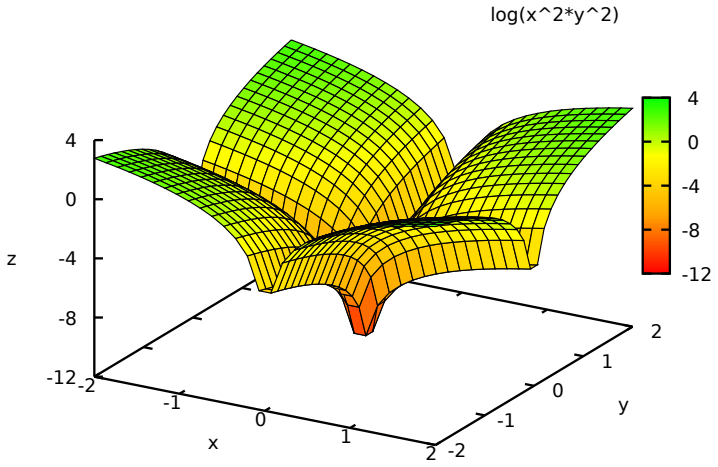
Uso da op.cão *z* para limitar uma função que se aproxima de infinito (neste caso a função aproxima-se de menos infinito nos eixos x e y); este exemplo mostra também como traçar gráficos apenas com linhas, sem superfícies coloridas.

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



Os valores infinitos de z podem ser também evitados escolhendo uma gralha que não inclua pontos onde a função é indeterminada, como no exemplo seguinte, que mostra também como modificar a paleta de cores e como incluir uma barra que relaciona as cores com os valores da variável z:

```
(%i1) plot3d (log (x^2*y^2), [x, -2, 2], [y, -2, 2],[grid, 29, 29],
        [palette, [gradient, red, orange, yellow, green]],
        color_bar, [xtics, 1], [ytics, 1], [ztics, 4],
        [color_bar_tics, 4])$
```



Duas superfícies no mesmo gráfico. Definem-se domínios diferentes para cada uma, colocando cada expressão e o seu domínio dentro de uma lista separada; define-se também um domínio global para o gráfico completo, após as definições das funções.

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

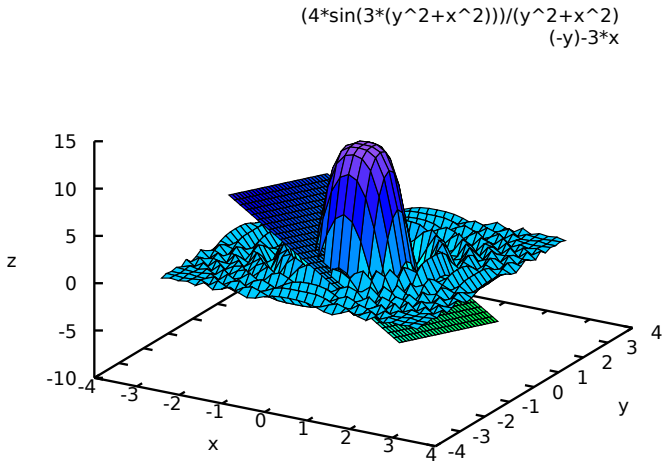


Gráfico de uma garrafa de Klein, definida parametricamente:

```
(%i1) expr_1: 5*cos(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3)-10$
(%i2) expr_2: -5*sin(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3)$
(%i3) expr_3: 5*(-sin(x/2)*cos(y)+cos(x/2)*sin(2*y))$
(%i4) plot3d ([expr_1, expr_2, expr_3], [x, -%pi, %pi],
        [y, -%pi, %pi], [grid, 50, 50])$
```

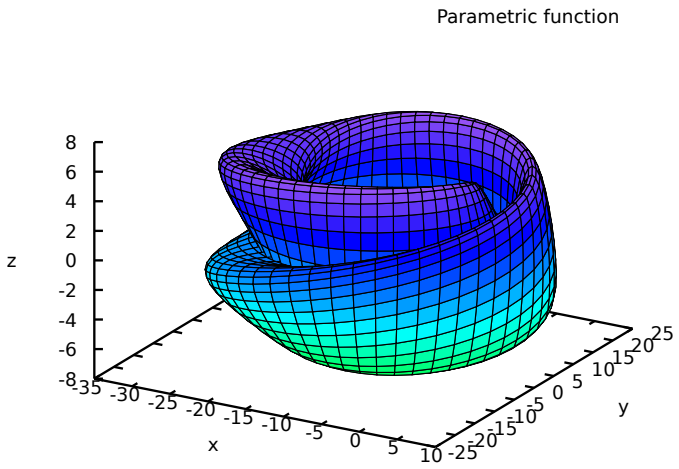
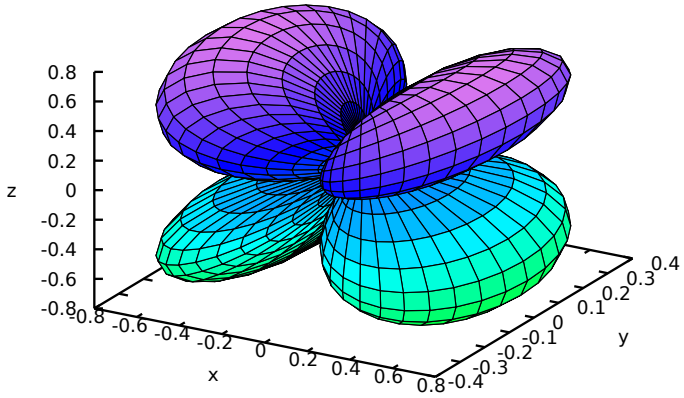


Gráfico de uma função "harmónica esférica", usando a transformaçã pré-definida `spherical_to_xyz`, para transformar de coordenadas esféricas para retangulares. Consulte a documentação de `spherical_to_xyz`.

```
(%i1) plot3d (sin(2*theta)*cos(phi), [theta, 0, %pi],
        [phi, 0, 2*%pi],
        [transform_xy, spherical_to_xyz], [grid,30,60],
   [legend,false])$
```



Uso da função pré-definida `polar_to_xy` para transformar de coordenadas cilíndricas para retangulares. Consulte a documentação de `polar_to_xy`,

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

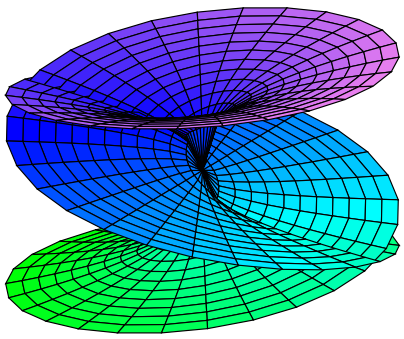
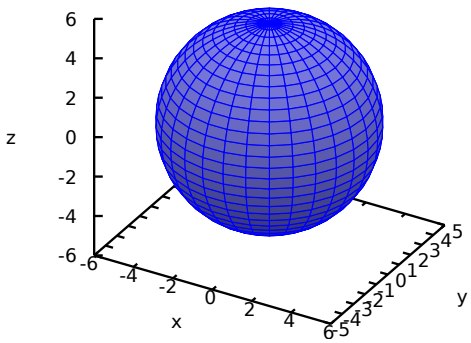


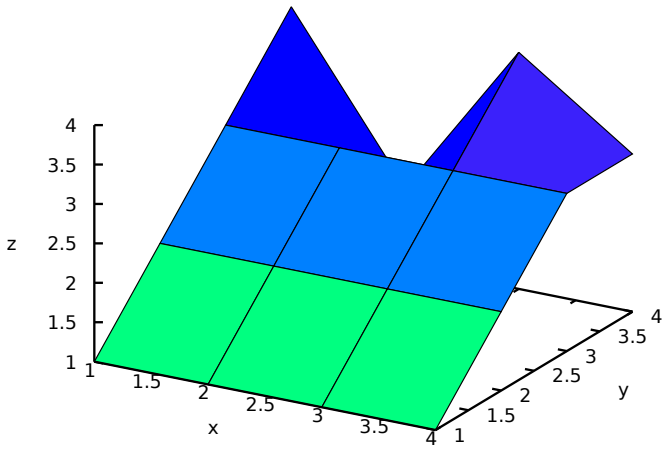
Gráfico de uma esfera, usando transformação de coordenadas esféricas para retangulares. Usa-se a opção `same_xyz` para obter a mesma escala nos três eixos. Quando se usam transformações de coordenadas, não convém eliminar as curvas traçadas na superfície, porque Gnuplot não mostrará o gráfico corretamente.

```
(%i1) plot3d ( 5, [theta, 0, %pi], [phi, 0, 2*%pi], same_xyz,
  [transform_xy, spherical_to_xyz], [mesh_lines_color,blue],
  [palette,[gradient,"#1b1b4e", "#8c8cf8"]], [legend, false])$
```



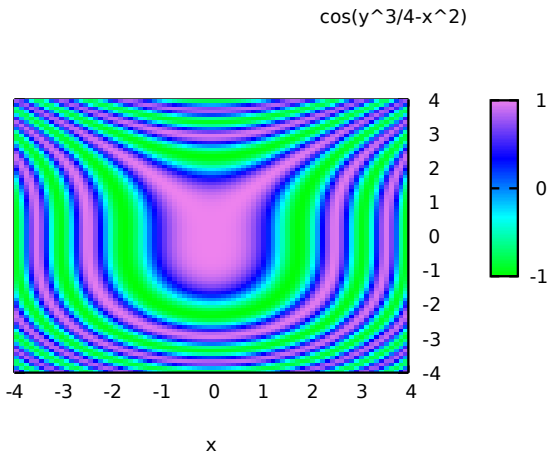
Definição de uma função de duas variáveis usando uma matriz. Repare-se no uso do apóstrofo na definição da função, para evitar que `plot3d` falhe queixando-se de que os índices da matriz deveriam ser números inteiros.

```
(%i1) M: matrix([1,2,3,4], [1,2,3,2], [1,2,3,4], [1,2,3,3])$
(%i2) f(x, y) := float('M [round(x), round(y)])$
(%i3) plot3d (f(x,y), [x,1,4], [y,1,4], [grid,3,3], [legend,false])$
```



Fixando um valor nulo para a elevação, uma superfície pode ser visualizada como um mapa, em que cada cor representa um valor diferente.

```
(%i1) plot3d (cos (-x^2 + y^3/4), [x,-4,4], [y,-4,4], [zlabel,""],
       [mesh_lines_color,false], [elevation,0], [azimuth,0],
       color_bar, [grid,80,80], [ztics,false], [color_bar_tics,1])$
```



Veja `plot_options` para mais exemplos.

`make_transform` (*vars*, *fx*, *fy*, *fz*) [Função]

Produz uma função adequada para a função transformação em plot3d. Usa-se conjuntamente com a opção gráfica `transform_xy`.

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

é uma transformação para coordenadas polares.

`set_plot_option` (*opção*) [Função]

Atribui valores às opções globais para impressão. *opção* é especificada como uma lista de dois ou mais elementos, na qual o primeiro elemeto é uma das palavras chave dentro da lista `plot_options`.

O argumento dado a `set_plot_option` é avaliado e `set_plot_option` retorna a lista completa `plot_options` (após modificar um desses elementos).

Veja também `plot_options`, `plot2d` e `plot3d`.

Exemplos:

Modifica a gralha (`grid`) e o intervalo de `x`. Quando uma palavra chave em `plot_options` tiver um valor atribuído, colocar um apóstrofo evita que seja avaliado.

```
(%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]]
```

9 Entrada e Saída

9.1 Comentários

Um comentário na entrada do Maxima é qualquer texto entre `/*` e `*/`.

O analisador do Maxima trata um comentário como espação em branco para o propósito de encontrar indicações no fluxo de entrada; uma indicação sempre termina um comentário. Uma entrada tal como `a/* foo */b` contém duas indicações, `a` e `b`, e não uma indicação simples `ab`. Comentários são de outra Comments are otherwise ignored by Maxima; nem o conteúdo nem a localização dos comentários são armazenados pelo analisador de expressões de entrada.

Comentários podem ser aninhados de forma a terem um nível de estratificação arbitrario. O delimitador `/*` e o delimitador `*/` formam pares. A quantidade de `/*` deve ser a mesma quantidade de `*/`.

Exemplos:

```
(%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 arbitrary depth */ */ */  1 + xyz;
(%o5)                         xyz + 1
```

9.2 Ficheiros

Um ficheiro é simplesmente uma área sobre um dispositivo particular de armazenagem que contém dados ou texto. Ficheiros em disco são figurativamente agrupados dentro de "directórios". Um directório é apenas uma lista de ficheiros. Comandos que lidam com ficheiros são: `save`, `load`, `loadfile`, `stringout`, `batch`, `demo`, `writefile`, `closefile`, e `appendfile`.

9.3 Definições para Entrada e Saída de Dados

`__` [Variável de sistema]

`__` é a expressão de entrada actualmente sendo avaliada. Isto é, enquanto um expressão de entrada *expr* está sendo avaliada, `__` é *expr*.

`__` é atribuída à expressão de entrada antes de a entrada ser simplificada ou avaliada. Todavia, o valor de `__` é simplificado (mas não avaliado) quando for mostrado.

`__` é reconhecido por `batch` e `load`. Em um ficheiro processado por `batch`, `__` tem o mesmo significado que na linha de comando interativa. Em um ficheiro processado por `load`, `__` está associado à expressão de entrada mais recentemente informada no prompt interativo ou em um ficheiro de lote (batch); `__` não é associado à expressões de entrada no ficheiro que está sendo processado. Em particular, quando

`load (`*`nomeficheiro`*`)` for chamado a partir da linha de comando interativa, `__` é associado a `load (`*`filename`*`)` enquanto o ficheiro está sendo processado.

Veja também `_` e `%`.

Exemplos:

```
(%i1) print ("Eu fui chamada como", __);
Eu fui chamada como print(Eu fui chamada como, __)
(%o1)              print(Eu fui chamada como, __)
(%i2) foo (__);
(%o2)                       foo(foo(__))
(%i3) g (x) := (print ("Expressão actual de entrada =", __), 0);
(%o3) g(x) := (print("Expressão actual de entrada =", __), 0)
(%i4) [aa : 1, bb : 2, cc : 3];
(%o4)                         [1, 2, 3]
(%i5) (aa + bb + cc)/(dd + ee + g(x));
                                cc + bb + aa
Expressão actual de entrada = --------------
                               g(x) + ee + dd
                                 6
(%o5)                           -------
                                ee + dd
```

`_` [Variável de sistema]

`_` é a mais recente expressão de entrada (e.g., `%i1`, `%i2`, `%i3`, ...).

A `_` é atribuída à expressão de entrada antes dela ser simplificada ou avaliada. Todavia, o valor de `_` é simplificado (mas não avaliado) quando for mostrado.

`_` é reconhecido por `batch` e `load`. Em um ficheiro processado por `batch`, `_` tem o mesmo significado que na linha de comando interativa. Em um ficheiro processado por load `load`, `_` está associado à expressão de entrada mais recentemente avaliada na linha de comando interativa ou em um ficheiro de lote; `_` não está associada a expressões de entrada no ficheiro que está sendo processado.

Veja também `__` e `%`.

Exemplos:

```
(%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
```

% [Variável de sistema]

% é a expressão de saída (e.g., %o1, %o2, %o3, ...) mais recentemente calculada pelo Maxima, pode ou não ser mostrada.

% é reconhecida por batch e load. Em um ficheiro processado por batch, % tem o mesmo significado que na linha de comando interativa. Em um ficheiro processado por load, % é associado à expressão de entrada mais recentemente calculada na linha de comando interativa ou em um ficheiro de lote; % não está associada a expressões de saída no ficheiro que está sendo processado.

Veja também _, %%, e %th

%% [Variável de sistema]

Em declaração composta, a saber block, lambda, ou (*s_1*, ..., *s_n*), %% é os valor da declaração anterior. Por exemplo,

```
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));
```

retornam o mesmo resultado, a saber 21/2.

Uma declaração composta pode compreender outras declarações compostas. Pode uma declaração ser simples ou composta, %% é o valor da declaração anterior. Por exemplo,

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

retorna 7*a^n.

Dentro da declaração composta, o valor de %% pode ser inspecionado em uma parada de linha de comando, que é aberta pela execução da função break. Por exemplo, na parada de linha de comando aberta por

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

digitando %%; retorna 42.

Na primeira declaração em uma declaração composta, ou fora de uma declaração composta, %% é indefinido.

%% reconhecido por batch e load, e possem o mesmo significao que na linha de comando interativa.

Veja também %.

`%edispflag` [Variável de opção]

Valor por omissão: `false`

Quando `%edispflag` é `true`, Maxima mostra `%e` para um expoente negativo como um quociente. Por exemplo, `%e^-x` é mostrado como `1/%e^x`.

`%th (i)` [Função]

O valor da i'ésima expressão prévia de saída. Isto é, se a próxima expressão a ser calculada for a n'ésima saída, `%th (m)` será a (n - m)'ésima saída.

`%th` é útil em ficheiros `batch` ou para referir-se a um grupo de expressões de saída. Por exemplo,

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

escolhe `s` para a soma das últimas dez expressões de saída.

`%th` é reconhecido por `batch` e `load`. Em um ficheiro processado por `batch`, `%th` possue o mesmo significado que na linha de comando interativa. Em um ficheiro processado por `load`, `%th` refere-se a expressões de saída mais recentemente calculadas na linha de comando interativa ou em um ficheiro de lote; `%th` não se refere a expressões de saída no ficheiro que está sendo processado.

Veja também `%`.

`?` [Símbolo especial]

Como prefixo para uma função ou nome de variável, `?` significa que o nome é um nome Lisp, não um nome Maxima. Por exemplo, `?round` significa a função Lisp `ROUND`. Veja *Lisp e Maxima* para mais sobre esse ponto.

A notação `? palavra` (um ponto de interrogação seguido de uma palavra e separado desta por um espaço em branco) é equivalente a `describe("palavra")`. O ponto de interrogação deve aparecer no início de uma linha de entrada; de outra forma o ponto de interrogação não é reconhecido com um pedido de documentação.

`??` [Símbolo especial]

A notação `?? palavra` (`??` seguido de um espaço em branco e uma palavra) é equivalente a `describe("palavra", inexact)`. O ponto de interrogação deve ocorrer no início de uma linha de entrada; de outra forma não é reconhecido com um pedido de documentação.

`absboxchar` [Variável de opção]

Valor por omissão: `!`

`absboxchar` é o caracter usado para para desenhar o sinal de valor absoluto em torno de expressões que são maiores que uma linha de altura.

`file_output_append` [Variável de opção]

Valor por omissão: `false`

`file_output_append` governa se funções de saída de ficheiro anexam ao final ou truncam seu ficheiro de saída. Quando `file_output_append` for `true`, tais funções anexam ao final de seu ficheiro de saída. De outra forma, o ficheiro de saída é truncado.

`save`, `stringout`, e `with_stdout` respeitam `file_output_append`. Outras funções que escrevem ficheiros de saída não respeitam `file_output_append`. Em partivular, montagem de gráficos e traduções de funções sempre truncam seu ficheiro de saída, e `tex` e `appendfile` sempre anexam ao final.

`appendfile` (*filename*) [Função]

Adiciona ao final de *filename* uma transcrição do console. `appendfile` é o mesmo que `writefile`, excepto que o ficheiro transcrito, se já existe, terá sempre alguma coisa adicionada ao seu final.

`closefile` fecha o ficheiro transcrito que foi aberto anteriormente por `appendfile` ou por `writefile`.

`batch` (*filename*) [Função]

Lê expressões Maxima do ficheiro *filename* e as avalia. `batch` procura pelo ficheiro *filename* na lista `file_search_maxima`. Veja `file_search`.

filename compreende uma sequência de expressões Maxima, cada uma terminada com `;` ou `$`. A varável especial `%` e a função `%th` referem-se a resultados prévios dentro do ficheiro. O ficheiro pode incluir construções `:lisp`. Espaços, tabulações, e o caracter de nova linha no ficheiro serão ignorados. um ficheiro de entrada conveniente pode ser criado por um editor de texto ou pela função `stringout`.

`batch` lê cada expressão de entrada de *filename*, mostra a entrada para o console, calcula a correspondente expressão de saída, e mostra a expressão de saída. Rótulos de entrada são atribuídos para expressões de entrada e rótulos de saída são atribuídos para expressões de saída. `batch` avalia toda expressão de entrada no ficheiro a menos que exista um erro. Se uma entrada de utilizador for requisitada (by `asksign` ou `askinteger`, por exemplo) `batch` interrompe para colectar a entrada requisitada e então continua.

Isso possibilita interromper `batch` pela digitação de `control-C` no console. O efeito de `control-C` depende da subjacente implementação do Lisp.

`batch` tem muitos usos, tais como fornecer um reservatório para trabalhar linhas de comando, para fornecer demonstrações livres de erros, ou para ajudar a organizar alguma coisa na solução de problemas complexos.

`batch` avalia seu argumento. `batch` não possui valor de retorno.

Veja também `load`, `batchload`, e `demo`.

`batchload` (*filename*) [Função]

Lê expressões Maxima de *filename* e as avalia, sem mostrar a entrada ou expressões de saída e sem atribuir rótulos para expressões de saída. Saídas impressas (tais como produzidas por `print` ou `describe`) são mostradas, todavia.

A variável especial `%` e a função `%th` referem-se a resultados anteriores do interpretador interativo, não a resultados dentro do ficheiro. O ficheiro não pode incluir construções `:lisp`.

`batchload` retorna o caminho de *filename*, como uma sequência de caracteres. `batchload` avalia seu argumento.

Veja também `batch` e `load`.

`closefile` () [Função]

Fecha o ficheiro transcrito aberto por `writefile` ou `appendfile`.

`collapse` (*expr*) [Função]

Reduz *expr* fazendo com que todas as suas subexpressões comuns (i.e., iguais) serem compartilhadas (i.e., usam a mesma células), dessa forma exonomizando espaço.

(`collapse` é uma subrotina usada pelo comando `optimize`.) Dessa forma, chamar `collapse` pode ser útil após um `save` ficheiro. Pode diminuir muitas expressões juntas pelo uso de `collapse ([`*expr_1*`, ..., `*expr_n*`])`. Similarmente, pode diminuir os elementos de um array `A` fazendo `collapse (listarray ('A))`.

`concat (`*arg_1*`, `*arg_2*`, ...)` [Função]

Concatena seus argumentos. Os argumentos devem obrigatóriamente serem avaliados para atomos. O valor de retorno é um símbolo se o primeiro argumento for um símbolo e uma sequência de caracteres no formato do Maxima em caso contrário.

`concat` avalia seus argumentos. O apóstrofo `'` evita avaliação.

```
(%i1) y: 7$
(%i2) z: 88$
(%i3) concat (y, z/2);
(%o3)                          744
(%i4) concat ('y, z/2);
(%o4)                          y44
```

Um símbolo construído por `concat` pode ser atribuído a um valor e aparecer em expressões. O operador de atribuição `::` (duplo dois pontos) avalia seu lado esquerdo.

```
(%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 que embora `concat (1, 2)` seja visto como um números, isso é uma sequência de caracteres no formato do Maxima.

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

`sconcat (`*arg_1*`, `*arg_2*`, ...)` [Função]

Concatena seus argumentos em uma sequência de caracteres. Ao contrário de `concat`, os argumentos arrumados *não* precisam ser atômicos.

O resultado é uma sequência de caracteres no format do Lisp.

```
(%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*`, ...)` [Função]

é como `display` mas somente os valores dos argumentos são mostrados em lugar de equações. Isso é útil para argumentos complicados que não possuem nomes ou onde somente o valor do argumento é de interesse e não o nome.

`dispcon` (*tensor_1*, *tensor_2*, ...) [Função]
`dispcon` (*all*) [Função]

Mostram as propriedades de contração de seus argumentos como foram dados para `defcon`. `dispcon(all)` mostra todas as propriedades de contração que foram definidas.

`display` (*expr_1*, *expr_2*, ...) [Função]

Mostra equações cujo lado esquerdo é *expr_i* não avaliado, e cujo lado direito é o valor da expressão centrada na linha. Essa função é útil em blocos e em `for` declarações com o objectivo de ter resultados intermédios mostrados. The Os argumentos para `display` são usualmente átomos, variáveis subscritas, ou chamadas de função. Veja também `disp`.

```
(%i1) display(B[1,2]);
                                      2
                             B     = X - X
                              1, 2
(%o1)                                done
```

`display2d` [Variável de opção]

Valor por omissão: `true`

Quando `display2d` é `false`, O console visualizador é unidimensional ao invés de bidimensional.

`display_format_internal` [Variável de opção]

Valor por omissão: `false`

Quando `display_format_internal` é `true`, expressões são mostradas sem ser por caminhos que escondam a representação matemática interna. O visualizador então corresponde ao que `inpart` retorna em lugar de `part`.

Exemplos:

```
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*) [Função]

Mostra *expr* em partes uma abaixo da outra. Isto é, primeiro o operador de *expr* é mostrado, então cada parcela em uma adição, ou factores em um produto, ou parte de uma expressão mais geral é mostrado separadamente. Isso é útil se *expr* é muito larga para ser mostrada de outra forma. Por exemplo se `P1`, `P2`, ... são expressões muito

largas então o programa visualizador pode sair fora do espaço de armazenamento na tentativa de mostrar `P1 + P2 + ...` tudo de uma vez. Todavia, `dispterms (P1 + P2 + ...)` mostra `P1`, então abaixo disso `P2`, etc. Quando não usando `dispterms`, se uma expressão exponencial é muito alta para ser mostrada como `A^B` isso aparece como `expt (A, B)` (ou como `ncexpt (A, B)` no caso de `A^^B`).

`error_size` [Variável de opção]

Valor por omissão: 10

`error_size` modifica mensagens de erro conforme o tamanho das expressões que aparecem nelas. Se o tamanho de uma expressão (como determinado pela função Lisp `ERROR-SIZE`) é maior que `error_size`, a expressão é substituída na mensagem por um símbolo, e o o símbolo é atribuído à expressão. Os símbolos são obtidos da lista `error_syms`.

De outra forma, a expressão é menor que `error_size`, e a expressão é mostrada na mensagem.

Veja também `error` e `error_syms`.

Exemplo:

O tamanho de `U`, como determinado por `ERROR-SIZE`, é 24.

```
(%i1) U: (C^D^E + B + A)/(cos(X-1) + 1)$

(%i2) error_size: 20$

(%i3) error ("Expressão exemplo é", U);

Expressão exemplo é 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 ("Expressão exemplo é", U);

                             E
                            D
                           C   + B + A
Expressão exemplo é --------------
                          cos(X - 1) + 1
 -- an error.  Quitting.  To debug this try debugmode(true);
```

`error_syms` [Variável de opção]

Valor por omissão: `[errexp1, errexp2, errexp3]`

Em mensagens de erro, expressões mais largas que `error_size` são substituídas por símbolos, e os símbolos são escolhidos para as expressões. Os símbolos são obtidos da lista `error_syms`. A primeira expressão muito larga é substituída por `error_syms[1]`, a segunda por `error_syms[2]`, e assim por diante.

Se houverem mais expressões muito largas que há elementos em `error_syms`, símbolos são construídos automaticamente, com o *n*-ésimo símbolo equivalente a `concat ('errexp, n)`.

Veja também `error` e `error_size`.

`expt (a, b)` [Função]

Se uma expressão exponencial é muito alta para ser mostrada cmo `a^b` isso aparece como `expt (a, b)` (ou como `ncexpt (a, b)` no caso de `a^^b`).

`expt` e `ncexpt` não são reconhecidas em entradas.

`exptdispflag` [Variável de opção]

Valor por omissão: `true`

Quando `exptdispflag` é `true`, Maxima mostra expressões com expoente negativo usando quocientes, e.g., `X^(-1)` como `1/X`.

`filename_merge (path, filename)` [Função]

Constroem um caminho modificado de *path* e *filename*. Se o componente final de *path* é da forma `###.algumacoisa`, o componente é substituído com `filename.algumacoisa`. De outra forma, o componente final é simplesmente substituído por *filename*.

`file_search (filename)` [Função]
`file_search (filename, pathlist)` [Função]

`file_search` procura pelo ficheiro *filename* e retorna o caminho para o ficheiro (como uma sequência de caracteres) se ele for achado; de outra forma `file_search` retorna `false`. `file_search (filename)` procura nos directórios padrões de busca, que são especificados pelas variáveis `file_search_maxima`, `file_search_lisp`, e `file_search_demo`.

`file_search` primeiro verifica se o nome actual passado existe, antes de tentar coincidir esse nome actual com o modelo "coringa" de busca do ficheiro. Veja `file_search_maxima` concernente a modelos de busca de ficheiros.

O argumento *filename* pode ser um caminho e nome de ficheiro, ou apenas um nome de ficheiro, ou, se um directório de busca de ficheiro inclui um modelo de busca de ficheiro, apenas a base do nome de ficheiro (sem uma extensão). Por exemplo,

```
file_search ("/home/wfs/special/zeta.mac");
file_search ("zeta.mac");
file_search ("zeta");
```

todos acham o mesmo ficheiro, assumindo que o ficheiro exista e `/home/wfs/special/###.mac` está em `file_search_maxima`.

`file_search (filename, pathlist)` procura somente nesses directórios especificados por *pathlist*, que é uma lista de sequências de caracteres. O argumento *pathlist* substitui os directórios de busca padrão, então se a lista do caminho é dada, `file_search` procura somente nesses especificados, e não qualquer dos directórios padrão

de busca. Mesmo se existe somente um directório em *pathlist*, esse deve ainda ser dado como uma lista de um único elemento.

O utilizador pode modificar o directório de busca padrão. Veja `file_search_maxima`.

`file_search` é invocado por `load` com `file_search_maxima` e `file_search_lisp` como directórios de busca.

`file_search_maxima` [Variável de opção]
`file_search_lisp` [Variável de opção]
`file_search_demo` [Variável de opção]

Essas variáveis especificam listas de directórios a serem procurados por `load`, `demo`, e algumas outras funções do Maxima. O valor padrão dessas variáveis nomeia vários directórios na instalaçã padrão do Maxima.

O usuáro pode modificar essas variáveis, quer substituindo os valores padrão ou colocando no final directórios adicionais. Por exemplo,

```
file_search_maxima: ["/usr/local/foo/###.mac",
    "/usr/local/bar/###.mac"]$
```

substitui o valor padrão de `file_search_maxima`, enquanto

```
file_search_maxima: append (file_search_maxima,
    ["/usr/local/foo/###.mac", "/usr/local/bar/###.mac"])$
```

adiciona no final da lista dois directórios adicionais. Isso pode ser conveniente para colocar assim uma expressão no ficheiro `maxima-init.mac` de forma que o caminho de busca de ficheiro é atribuído automaticamente quando o Maxima inicia.

Múltiplas extensões de ficheiro e e múltiplos caminhos podem ser especificados por construções "coringa" especiais. A sequência de caracteres `###` expande a busca para além do nome básico, enquanto uma lista separada por vírgulas e entre chaves `{foo,bar,baz}` expande em múltiplas sequências de caracteres. Por exemplo, supondo que o nome básico a ser procurado seja `neumann`,

```
"/home/{wfs,gcj}/###.{lisp,mac}"
```

expande em `/home/wfs/neumann.lisp`, `/home/gcj/neumann.lisp`, `/home/wfs/neumann.mac`, e `/home/gcj/neumann.mac`.

`file_type` (*filename*) [Função]

Retorna uma suposta informação sobre o conteúdo de *filename*, baseada na extensão do ficheiro. *filename* não precisa referir-se a um ficheiro actual; nenhuma tentativa é feita para abrir o ficheiro e inspecionar seu conteúdo.

O valor de retorno é um símbolo, qualquer um entre `object`, `lisp`, ou `maxima`. Se a extensão começa com `m` ou `d`, `file_type` retorna `maxima`. Se a extensão começa om `l`, `file_type` retorna `lisp`. Se nenhum dos acima, `file_type` retorna `object`.

`grind` (*expr*) [Função]
`grind` [Variável de opção]

A função `grind` imprime *expr* para o console em uma forma adequada de entrada para Maxima. `grind` sempre retorna `done`.

Quando *expr* for um nome de uma função ou o nome de uma macro, `grind` mostra na tela a definição da função ou da macro em lugar de apenas o nome.

Veja também `string`, que retorna uma sequência de caracteres em lugar de imprimir sua saída. `grind` tenta imprimir a expressão de uma maneira que a faz levemente mais fácil para ler que a saída de `string`.

Quando a variável `grind` é `true`, a saída de `string` e `stringout` tem o mesmo formato que `grind`; de outra forma nenhuma tentativa é feita para formatar especialmente a saída dessas funções. O valor padrão da variável `grind` é `false`.

`grind` pode também ser especificado como um argumento de `playback`. Quando `grind` está presente, `playback` imprime expressões de entrada no mesmo formato que a função `grind`. De outra forma, nenhuma tentativa é feita para formatar especialmente as expressões de entrada. `grind` avalia seus argumentos.

Exemplos:

```
(%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` [Variável de opção]

Valor por omissão: 10

Inteiros fornecidos dentro do Maxima são interpretados com respeito à base `ibase`.

A `ibase` pode ser atribuído qualquer inteiro entre 2 e 35 (decimal), inclusive. Quando `ibase` é maior que 10, os numerais compreendem aos numerais decimais de 0 até 9 mais as letras maiúsculas do alfabeto A, B, C, ..., como necessário. Os numerais para a base 35, a maior base aceitável, compreendem de 0 até 9 e de A até Y.

Veja também `obase`.

`inchar` [Variável de opção]

Valor por omissão: `%i`

`inchar` é o prefixo dos rótulos de expressões fornecidas pelo utilizador. Maxima automaticamente constrói um rótulo para cada expressão de entrada por concatenação de `inchar` e `linenum`. A `inchar` pode ser atribuído qualquer sequência de caracteres ou símbolo, não necessariamente um caracter simples.

```
(%i1) inchar: "input";
(%o1)                                input
(input1) expand ((a+b)^3);
                             3        2      2      3
(%o1)                       b  + 3 a b  + 3 a  b + a
(input2)
```

Veja também `labels`.

`ldisp` (*expr_1*, ..., *expr_n*) [Função]

Mostra expressões *expr_1*, ..., *expr_n* para o console como saída impressa na tela. `ldisp` atribue um rótulo de expressão intermédia a cada argumento e retorna a lista de rótulos.

Veja também `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*) [Função]

Mostra expressões *expr_1*, ..., *expr_n* para o console como saída impressa na tela. Cada expressão é impressa como uma equação da forma `lhs = rhs` na qual `lhs` é um dos argumentos de `ldisplay` e `rhs` é seu valor. Tipicamente cada argumento é uma variável. `ldisp` atribui um rótulo de expressão intermediáia a cada equação e retorna a lista de rótulos.

Veja também `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` [Variável de opção]

Valor por omissão: `%t`

`linechar` é o refixo de rótulos de expressões intermédias gerados pelo Maxima. Maxima constrói um rótulo para cada expressão intermédia (se for mostrada) pela concatenação de `linechar` e `linenum`. A `linechar` pode ser atribuído qualquer sequência de caracteres ou símbolo, não necessáriamente um caractere simples.

Expressões intermédias podem ou não serem mostradas. See `programmode` e `labels`.

`linel` [Variável de opção]

Valor por omissão: 79

`linel` é a largura assumida (em caracteres) do console para o propósito de mostrar expressões. A `linel` pode ser atribuído qualquer valor pelo utilizador, embora valores muio pequenos ou muito grandes possam ser impraticáveis. Textos impressos por funções internas do Maxima, tais como mensagens de erro e a saída de `describe`, não são afectadas por `linel`.

`lispdisp` [Variável de opção]

Valor por omissão: `false`

Quando `lispdisp` for `true`, símbolos Lisp são mostrados com um ponto de interrogação `?` na frente. De outra forma, símbolos Lisp serão mostrados sem o ponto de interrogação na frente.

Exemplos:

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

```
(%i4) ?foo + ?bar;
(%o4)                        ?foo + ?bar
```

load (*nomeficheiro*) [Função]

Avalia expressões em *nomeficheiro*, dessa forma conduzindo variáveis, funções, e outros objectos dentro do Maxima. A associação de qualquer objecto existente é substituída pela associação recuperada de *nomeficheiro*. Para achar o ficheiro, `load` chama `file_search` com `file_search_maxima` e `file_search_lisp` como directórios de busca. Se `load` obtém sucesso, isso retorna o nome do ficheiro. De outra forma `load` imprime uma mensagem e erro.

`load` trabalha igualmente bem para códigos Lisp e códigos Maxima. Ficheiros criados por `save`, `translate_file`, e `compile_file`, que criam códigos Lisp, e `stringout`, que criam códigos Maxima, podem ser processadas por `load`. `load` chama `loadfile` para carregar ficheiros Lisp e `batchload` para carregar ficheiros Maxima.

`load` não reconhece construções `:lisp` em ficheiros do Maxima, e quando processando *nomeficheiro*, as variáveis globais `_`, `__`, `%`, e `%th` possuem as mesmas associações que possuiam quando `load` foi chamada.

Veja também `loadfile`, `batch`, `batchload`, e `demo`. `loadfile` processa ficheiros Lisp; `batch`, `batchload`, e `demo` processam ficheiros Maxima.

Veja `file_search` para mais detalhes sobre o mecanismo de busca de ficheiros.

`load` avalia seu argumento.

loadfile (*nomeficheiro*) [Função]

Avalia expressões Lisp em *nomeficheiro*. `loadfile` não invoca `file_search`, então `nomeficheiro` deve obrigatóriamente incluir a extensão do ficheiro e tanto quanto o caminho como necessário para achar o ficheiro.

`loadfile` pode processar ficheiros criados por `save`, `translate_file`, e `compile_file`. O utilizador pode achar isso mais conveniente para usar `load` em lugar de `loadfile`.

`loadfile` avalia seu argumento, então `nomeficheiro` deve obrigatóriamente ser uma sequência de caracteres literal, não uma variável do tipo sequência de caracteres. O operador apóstrofo-apóstrofo `''` não aceita avaliação.

loadprint [Variável de opção]

Valor por omissão: `true`

`loadprint` diz se deve imprimir uma mensagem quando um ficheiro é chamado.

- Quando `loadprint` é `true`, sempre imprime uma mensagem.
- Quando `loadprint` é `'loadfile`, imprime uma mensagem somente se um ficheiro é chamado pela função `loadfile`.
- Quando `loadprint` é `'autoload`, imprime uma mensagem somente se um ficheiro é automaticamente carregado. Veja `setup_autoload`.
- Quando `loadprint` é `false`, nunca imprime uma mensagem.

obase [Variável de opção]

Valor por omissão: 10

`obase` é a base para inteiros mostrados pelo Maxima.

A `obase` poode ser atribuído qualquer inteiro entre 2 e 35 (decimal), inclusive. Quando `obase` é maior que 10, os numerais compreendem os numerais decimais de 0 até 9 e letras maiúsulas do alfabeto A, B, C, ..., quando necessário. Os numerais para a base 35, a maior base aceitável, compreendem de 0 até 9, e de A até Y.

Veja também `ibase`.

`outchar` [Variável de opção]

Valor por omissão: `%o`

`outchar` é o prefixo dos rótulos de expressões calculadas pelo Maxima. Maxima automaticamente constrói um rótulo para cada expressão calculada pela concatenação de `outchar` e `linenum`. A `outchar` pode ser atribuído qualquer sequência de caracteres ou símbolo, não necessáriamente um caractere simples.

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

Veja também `labels`.

`packagefile` [Variável de opção]

Valor por omissão: `false`

Projetistas de pacotes que usam `save` ou `translate` para criar pacotes (ficheiros) para outros usarem podem querer escolher `packagefile: true` para prevenir qu informações sejam acrescentadas à lista de informações do Maxima (e.g. `values`, `funções`) excepto onde necessário quando o ficheiro é carregado. Nesse caminho, o conteúdo do pacote não pegará no caminho do utilizador quando ele adicionar seus próprios dados. Note que isso não resolve o problema de possíveis conflitos de nome. Também note que o sinalizador simplesmente afecta o que é saída para o ficheiro pacote. Escolhendo o sinalizador para `true` é também útil para criar ficheiros de init do Maxima.

`pfeformat` [Variável de opção]

Valor por omissão: `false`

Quando `pfeformat` é `true`, uma razão de inteiros é mostrada com o caractere sólido (barra normal), e um denominador inteiro `n` é mostrado como um termo multiplicativo em primeiro lugar `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*) [Função]

Avalia e mostra *expr_1*, ..., *expr_n* uma após a outra, da esquerda para a direita, iniciando no lado esquerdo do console.

O valor retornado por `print` é o valor de seu último argumento. `print` não gera rótulos de expressão intermédia.

Veja também `display`, `disp`, `ldisplay`, e `ldisp`. Essas funções mostram uma expressão por linha, enquanto `print` tenta mostrar duas ou mais expressões por linha.

Para mostrar o conteúdo de um ficheiro, veja `printfile`.

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

`tcl_output` (*list*, *i0*, *skip*) [Função]
`tcl_output` (*list*, *i0*) [Função]
`tcl_output` ([*list_1*, ..., *list_n*], *i*) [Função]

Imprime os elementos de uma lista entre chaves `{ }`, conveniente como parte de um programa na linguagem Tcl/Tk.

`tcl_output` (*list*, *i0*, *skip*) imprime *list*, começando com o elemento *i0* e imprimindo elementos *i0* + *skip*, *i0* + 2 *skip*, etc.

`tcl_output` (*list*, *i0*) é equivalente a `tcl_output` (*list*, *i0*, 2).

`tcl_output` ([*list_1*, ..., *list_n*], *i*) imprime os *i*'ésimos elementos de *list_1*, ..., *list_n*.

Exemplos:

```
(%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*) [Função]

Imprime *expr_1*, ..., *expr_n*, então lê uma expressão do console e retorna a expressão avaliada. A expressão é terminada com um ponto e vírgula `;` ou o sinal de dólar `$`.

Veja também `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*) [Função]

Imprime *expr_1*, ..., *expr_n*, então lê uma expressão do console e retorna a expressão (sem avaliação). A expressão é terminada com um `;` (ponto e vírgula) ou `$` (sinal de dólar).

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

Veja também `read`.

`reveal` (*expr*, *depth*) [Função]

Substitue partes de *expr* no inteiro especificado *depth* com sumário descritivo.

- Somas e diferenças são substituídas por `sum(n)` onde *n* é o número de operandos do produto.
- Produtos são substituídos por `product(n)` onde *n* é o número de operandos da multiplicação.
- Exponenciais são substituídos por `expt`.
- Quocientes são substituídos por `quotient`.
- Negação unária é substituída por `negterm`.

Quando *depth* é maior que ou igual à máxima intensidade de *expr*, `reveal (expr, depth)` retornam *expr* sem modificações.

`reveal` avalia seus argumentos. `reveal` retorna expressão sumarizada.

Exemplo:

```
(%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` [Variável de opção]

Valor por omissão: `]`

`rmxchar` é the caractere desenhado lado direito de uma matriz.

Veja também `lmxchar`.

`save` (*filename*, *nome_1*, *nome_2*, *nome_3*, ...) [Função]
`save` (*filename*, *values*, *functions*, *labels*, ...) [Função]
`save` (*filename*, [*m*, *n*]) [Função]
`save` (*filename*, *nome_1*=*expr_1*, ...) [Função]
`save` (*filename*, *all*) [Função]
`save` (*filename*, *nome_1*=*expr_1*, *nome_2*=*expr_2*, ...) [Função]

Armazena os valores correntes de *nome_1*, *nome_2*, *nome_3*, ..., em *filename*. Os argumentos são os nomes das variáveis, funções, ou outros objectos. Se um nome não possui valore ou função associada a ele, esse nome sem nenhum valor ou função associado será ignorado. `save` retorna *filename*.

`save` armazena dados na forma de expressões Lisp. Os dados armazenados por `save` podem ser recuperados por `load (`*filename*`)`.

O sinalizador global `file_output_append` governa se `save` anexa ao final ou trunca o ficheiro de saída. Quando `file_output_append` for `true`, `save` anexa ao final doficheiro de saída. De outra forma, `save` trunca o ficheiro de saída. Nesse caso, `save` cria o ficheiro se ele não existir ainda.

A forma especial `save (`*filename*`, values, functions, labels, ...)` armazena os ítens nomeados por `values`, `funções`, `labels`, etc. Os nomes podem ser quaisquer especificados pela variável `infolists`. `values` compreende todas as variáveis definidas pelo utilizador.

A forma especial `save (`*filename*`, [`*m*`, `*n*`])` armazena os valores de rótulos de entrada e saída de *m* até *n*. Note que *m* e *n* devem obrigatóriamente ser inteiros literais. Rótulos de entrada e saída podem também ser armazenados um a um, e.g., `save ("foo.1", %i42, %o42)`. `save (`*filename*`, labels)` armazena todos os rótulos de entrada e saída. Quando rótulos armazenados são recuperados, eles substituem rótulos existentes.

A forma especial `save (`*filename*`, `*nome_1*`=`*expr_1*`, `*nome_2*`=`*expr_2*`, ...)` armazena os valores de *expr_1*, *expr_2*, ..., com nomes *nome_1*, *nome_2*, ... Isso é útil para aplicar essa forma para rótulos de entrada e saída, e.g., `save ("foo.1", aa=%o88)`. O lado direito dessa igualdade nessa forma pode ser qualquer expressão, que é avaliada. Essa forma não introduz os novos nomes no ambiente corrente do Maxima, mas somente armazena-os em *filename*.

Essa forma especial e a forma geral de `save` podem ser misturados. Por exemplo, `save (`*filename*`, aa, bb, cc=42, funções, [11, 17])`.

A forma especial `save (`*filename*`, all)` armazena o estado corrente do Maxima. Isso inclui todas as variáveis definidas pelo utilizador, funções, arrays, etc., bem como alguns ítens definidos automaticamente. Os ítes salvos incluem variáveis de sistema,

tais como `file_search_maxima` ou `showtime`, se a elas tiverem sido atribuídos novos valores pelo utilizador; veja `myoptions`.

`save` avalia seus argumentos. *filename* deve obrigatóriamente ser uma sequência de caracteres, não uma variável tipo sequência de caracteres. O primeiro e o último rótulos a salvar, se especificado, devem obrigatóriamente serem inteiros. O operador apóstrofo-apóstrofo `''` avalia uma variável tipo sequência de caracteres para seu valor sequência de caracteres, e.g., `s: "foo.1"$ save (''s, all)$`, e variáveis inteiras para seus valores inteiros, e.g., `m: 5$ n: 12$ save ("foo.1", [''m, ''n])$`.

`savedef` [Variável de opção]

Valor por omissão: `true`

Quando `savedef` é `true`, a vesão Maxima de uma função de utilizador é preservada quando a função é traduzida. Isso permite que a definição seja mostrada por `dispfun` e autoriza a função a ser editada.

Quando `savedef` é `false`, os nomes de funções traduzidas são removidos da lista de `funções`.

`show` (*expr*) [Função]

Mostra `expr` com os objectos indexados tendo índices covariantes como subscritos, índices contravariantes como sobrescritos. Os índices derivativos são mostrados como subscritos, separados dos índices covariantes por uma vírgula.

`showratvars` (*expr*) [Função]

Retorna uma lista de variáveis expressão racional canónica (CRE) na expressão `expr`.

Veja também `ratvars`.

`stardisp` [Variável de opção]

Valor por omissão: `false`

Quando `stardisp` é `true`, multiplicação é mostrada com um asterisco `*` entre os operandos.

`string` (*expr*) [Função]

Converte `expr` para a notação linear do Maxima apenas como se tivesse sido digitada.

O valor de retorno de `string` é uma sequência de caracteres, e dessa forma não pode ser usada em um cálculo.

`stringdisp` [Variável de opção]

Valor por omissão: `false`

Quando `stringdisp` for `true`, sequências de caracteres serão mostradas contidas em aspas duplas. De outra forma, aspas não são mostradas.

`stringdisp` é sempre `true` quando mostrando na tela uma definição de função.

Exemplos:

```
(%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*, ...) [Função]
`stringout` (*filename*, [*m*, *n*]) [Função]
`stringout` (*filename*, *input*) [Função]
`stringout` (*filename*, *functions*) [Função]
`stringout` (*filename*, *values*) [Função]

`stringout` escreve expressões para um ficheiro na mesma forma de expressões que foram digitadas para entrada. O ficheiro pode então ser usado como entrada para comandos `batch` ou `demo`, e isso pode ser editado para qualquer propósito. `stringout` pode ser executado enquanto `writefile` está em progresso.

O sinalizador global `file_output_append` governa se `stringout` anexa ao final ou trunca o ficheiro de saída. Quando `file_output_append` for `true`, `stringout` anexa ao final do ficheiro de saída. De outra forma, `stringout` trunca o ficheiro de saída. Nesse caso, `stringout` cria o ficheiro de saída se ele não existir ainda.

A forma geral de `stringout` escreve os valores de um ou mais expressões para o ficheiro de saída. Note que se uma expressão é uma variável, somente o valor da variável é escrito e não o nome da variável. Como um útil caso especial, as expressões podem ser rótulos de entrada (`%i1`, `%i2`, `%i3`, ...) ou rótulos de saída (`%o1`, `%o2`, `%o3`, ...).

Se `grind` é `true`, `stringout` formata a saída usando o formato `grind`. De outra forma o formato `string` é usado. Veja `grind` e `string`.

A forma especial `stringout (`*filename*`, [`*m*`, `*n*`])` escreve os valores dos rótulos de entrada de m até n, inclusive.

A forma especial `stringout (`*filename*`, input)` escreve todos os rótulos de entrada para o ficheiro.

A forma especial `stringout (`*filename*`, functions)` escreve todas as funções definidas pelo utilizador (nomeadas pela lista global `functions`) para o ficheiro.

A forma especial `stringout (`*filename*`, values)` escreve todas as variáveis atribuídas pelo utilizador (nomeadas pela lista global `values`) para o ficheiro. Cada variável é impressa como uma declaração de atribuição, com o nome da variável seguida de dois pontos, e seu valor. Note que a forma geral de `stringout` não imprime variáveis como declarações de atribuição.

`tex` (*expr*) [Função]
`tex` (*rótulo*) [Função]
`tex` (*expr*, *momeficheiro*) [Função]
`tex` (*label*, *nomeficheiro*) [Função]

Imprime uma representação de uma expressão adequada para o sistema TeX de preparação de documento. O resultado é um fragmento de um documento, que pode ser copiado dentro de um documento maior. Esse fragmento não pode ser processado de forma directa e isolada.

`tex (`*expr*`)` imprime uma representação TeX da *expr* no console.

`tex (`*`rótulo`*`)` imprime uma representação TeX de uma expressão chamada *rótulo* e atribui a essa um rótulo de equação (a ser mostrado à esquerda da expressão). O rótulo de equação TeX é o mesmo que o rótulo da equação no Maxima.

`tex (`*`expr`*`, `*`nomeficheiro`*`)` anexa ao final uma representação TeX de *expr* no ficheiro *nomeficheiro*. `tex` não avalia o argumento *nomeficheiro*; apóstrofo-apóstrofo `''` força a avaliação so argumento.

`tex (`*`rótulo`*`, `*`nomeficheiro`*`)` anexa ao final uma representação TeX da expressão chamada de *rótulo*, com um rótulo de equação, ao ficheiro *nomeficheiro*.

`tex` não avalia o argumento *nomeficheiro*; apóstrofo-apóstrofo `''` força a avaliação so argumento. `tex` avalia seus argumentos após testar esse argumento para ver se é um rótulo. duplo apóstrofo `''` força a avaliação do argumento, desse modo frustrando o teste e prevenindo o rótulo.

Veja também `texput`.

Exemplos:

```
(%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)
```

`texput (`a`, `*`s`*`)` [Função]
`texput (`a`, `*`s`*`, `*`operator_type`*`)` [Função]
`texput (`a`, [`*`s_1`*`, `*`s_2`*`], `*`matchfix`*`)` [Função]
`texput (`a`, [`*`s_1`*`, `*`s_2`*`, `*`s_3`*`], `*`matchfix`*`)` [Função]

Atribui a saída TeX para o átomo *a*, que pode ser um símbolo ou o nome de um operador.

`texput (a, `*`s`*`)` faz com que a função `tex` interpole a sequência de caracteres *s* dentro da saída TeX em lugar de *a*.

`texput (a, `*`s`*`, `*`operator_type`*`)`, onde *operator_type* é `prefix`, `infix`, `postfix`, `nary`, ou `nofix`, faz com que a função `tex` interpole *s* dentro da saída TeX em lugar de *a*, e coloca o texto interpolado na posição apropriada.

`texput (a, [`*`s_1`*`, `*`s_2`*`], matchfix)` faz com que a função `tex` interpole *s_1* e *s_2* dentro da saída TeX sobre qualquer lado dos argumentos de *a*. Os argumentos (se mais de um) são separados por vírgulas.

`texput (a, [`*s_1*`, `*s_2*`, `*s_3*`], matchfix)` faz com que a função `tex` interpole *s_1* e *s_2* dentro da saída TeX sobre qualquer lado dos argumentos de *a*, com *s_3* separando os argumentos.

Exemplos:

Atribui saída TeX a uma variável.

```
(%i1) texput (me,"\\mu_e");
(%o1)                          \mu_e
(%i2) tex (me);
$$\mu_e$$
(%o2)                          false
```

Atribui saída TeX a uma função comum (não a um operador).

```
(%i1) texput (lcm, "\\mathrm{lcm}");
(%o1)                      \mathrm{lcm}
(%i2) tex (lcm (a, b));
$$\mathrm{lcm}\left(a , b\right)$$
(%o2)                          false
```

Atribui saída TeX a um operador prefixado.

```
(%i1) prefix ("grad");
(%o1)                          grad
(%i2) texput ("grad", " \\nabla ", prefix);
(%o2)                         \nabla
(%i3) tex (grad f);
$$ \nabla f$$
(%o3)                          false
```

Atribui saída TeX a um operador infixado.

```
(%i1) infix ("~");
(%o1)                            ~
(%i2) texput ("~", " \\times ", infix);
(%o2)                         \times
(%i3) tex (a ~ b);
$$a \times b$$
(%o3)                          false
```

Atribui saída TeX a um operadro pósfixado.

```
(%i1) postfix ("##");
(%o1)                           ##
(%i2) texput ("##", "!!", postfix);
(%o2)                           !!
(%i3) tex (x ##);
$$x!!$$
(%o3)                          false
```

Atribui saída TeX a um operador n-ário.

```
(%i1) nary ("@@");
(%o1)                           @@
(%i2) texput ("@@", " \\circ ", nary);
```

```
(%o2)                          \circ
(%i3) tex (a @@ b @@ c @@ d);
$$a \circ b \circ c \circ d$$
(%o3)                          false
```

Atribui saída TeX a um operador nofix.

```
(%i1) nofix ("foo");
(%o1)                           foo
(%i2) texput ("foo", "\\mathsc{foo}", nofix);
(%o2)                      \mathsc{foo}
(%i3) tex (foo);
$$\mathsc{foo}$$
(%o3)                          false
```

Atribui saída TeX a um operadro matchfix.

```
(%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
```

`system` (*comando*) [Função]

Executa *comando* como um processo separado. O comando é passado ao shell padrão para execução. `system` não é suportado por todos os sistemas operacionais, mas geralmente existe em ambientes Unix e Unix-like.

Supondo que `_hist.out` é uma lista de frequências que deseja imprimir como um gráfico em barras usando `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"));
```

Com o objectivo de fazer com que a impressão do gráfico seja concluída em segundo plano (retornando o controle para o Maxima) e remover o ficheiro temporário após isso ter sido concluído faça:

```
system("(xgraph -bar -brw .7 -nl < _hist.out;  rm -f _hist.out)&")
```

`ttyoff` [Variável de opção]

Valor por omissão: `false`

Quando `ttyoff` é `true`, expressões de saída não são mostradas. Expressões de saída são ainda calculadas e atribuídas rótulos. Veja `labels`.

Textos impresso por funções internas do Maxima, tais como mensagens de erro e a saída de `describe`, não são afectadas por `ttyoff`.

`with_stdout` (*filename*, *expr_1*, *expr_2*, *expr_3*, ...) [Função]

Abre *filename* e então avalia *expr_1*, *expr_2*, *expr_3*, ... Os valores dos argumentos não são armazenados em *filename*, mas qualquer saída impressa gerada pela avaliação dos argumentos (de `print`, `display`, `disp`, ou `grind`, por exemplo) vai para *filename* em lugar do console.

O sinalizador global `file_output_append` governa se `with_stdout` anexa ao final ou trunca o ficheiro de saída. Quando `file_output_append` for `true`, `with_stdout` anexa ao final do ficheiro de saída. De outra forma, `with_stdout` trunca o ficheiro de saída. Nesse caso, `with_stdout` cria o ficheiro se ele não existir ainda.

`with_stdout` retorna o valor do seu argumento final.

Veja também `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*) [Função]

Começa escrevendo uma transcrição da sessão Maxima para *filename*. Toda interação entre o utilizador e Maxima é então gravada nesse ficheiro, da mesma forma que aparece no console.

Como a transcrição é impressa no formato de saída do console, isso não pode ser reaproveitado pelo Maxima. Para fazer um ficheiro contendo expressões que podem ser reaproveitadas, veja `save` e `stringout`. `save` armazena expressões no formato Lisp, enquanto `stringout` armazena expressões no formato Maxima.

O efeito de executar `writefile` quando *filename* ainda existe depende da implementação Lisp subjacente; o ficheiro transcrito pode ser substituído, ou o ficheiro pode receber um anexo. `appendfile` sempre anexa para o ficheiro transcrito.

Isso pode ser conveniente para executar `playback` após `writefile` para salvar a visualização de interações prévias. Como `playback` mostra somente as variáveis de entrada e saída (`%i1`, `%o1`, etc.), qualquer saída gerada por uma declaração de impressão em uma função (como oposição a um valor de retorno) não é mostrada por `playback`.

`closefile` fecha o ficheiro transcrito aberto por `writefile` ou `appendfile`.

10 Ponto Flutuante

10.1 Definições para ponto Flutuante

`bffac` (*expr*, *n*) [Função]

Versão para grandes números em ponto flutuante da função `factorial` (usa o artifício gamma). O segundo argumento informa quantos dígitos reter e retornar, isso é uma boa idéia para requisitar precisão adicional.

`load ("bffac")` chama essa função.

`algepsilon` [Variável de Opção]

Valor por omissão: 10^8

`algepsilon` é usada por `algsys`.

`bfloat` (*expr*) [Função]

Converte todos os números e funções de números em *expr* para grandes números em ponto flutuante (bigfloat). O número de algarismos significativos no grande número em ponto flutuante resultante é especificado através da variável global `fpprec`.

Quando `float2bf` for `false` uma mensagem de alerta é mostrada quando uma número em ponto flutuante (float) é convertido em um grande número em ponto flutuante (bigfloat - uma vez que isso pode resultar em perda de precisão).

`bfloatp` (*expr*) [Função]

Retorna `true` se a avaliação da *expr* resultar em um grande número em ponto flutuante, de outra forma retorna `false`.

`bfpsi` (*n*, *z*, *fpprec*) [Função]
`bfpsi0` (*z*, *fpprec*) [Função]

`bfpsi` é a função `polygamma` de argumentos reais *z* e ordem de inteiro *n*. `bfpsi0` é a função `digamma`. `bfpsi0 (z, fpprec)` é equivalente a `bfpsi (0, z, fpprec)`.

Essas funções retornam valores em grandes números em ponto flutuante. *fpprec* é a precisão do valor de retorno dos grandes números em ponto flutuante.

`load ("bffac")` chama essas funções.

`bftorat` [Variável de Opção]

Valor por omissão: `false`

`bftorat` controla a conversão de `bfloats` para números racionais. Quando `bftorat` for `false`, `ratepsilon` será usada para controlar a conversão (isso resulta em números racionais relativametne pequenos). Quando `bftorat` for `true`, o número racional gerado irá representar precisamente o `bfloat`.

`bftrunc` [Variável de Opção]

Valor por omissão: `true`

`bftrunc` faz com que tilhas de zeros em grandes números em ponto flutuante diferentes de zero sejam ocultadas. Desse modo, se `bftrunc` for `false`, `bfloat (1)` será mostrado como `1.000000000000000B0`. De outra forma, será mostrado como `1.0B0`.

`cbffac` (*z*, *fpprec*) [Função]

Factorial complexo de grandes números em ponto flutuante.

`load ("bffac")` chama essa função.

`float` (*expr*) [Função]

Converte inteiros, números racionais e grandes números em ponto flutuante em *expr* para números em ponto flutuante. Da mesma forma um `evflag`, `float` faz com que números racionais não-inteiros e grandes números em ponto flutuante sejam convertidos para ponto flutuante.

`float2bf` [Variável de Opção]

Valor por omissão: `false`

Quando `float2bf` for `false`, uma mensagem de alerta é mostrada quando um número em ponto flutuante é convertido em um grande número em ponto flutuante (uma vez que isso pode resultar em perda de precisão).

`floatnump` (*expr*) [Função]

Retorna `true` se *expr* for um número em ponto flutuante, de outra forma retorna `false`.

`fpprec` [Variável de Opção]

Valor por omissão: 16

`fpprec` é o número de algarismos significativos para aritmética sobre grandes números em ponto flutuante `fpprec` não afecta cálculos sobre números em ponto flutuante comuns.

Veja também `bfloat` e `fpprintprec`.

`fpprintprec` [Variável de Opção]

Valor por omissão: 0

`fpprintprec` é o n;umero de dígitos a serem mostrados na tela quando no caso de nuúmeros em ponto flutuante e no caso de grandes números em ponto flutuante.

Para números em ponto flutuante comuns, quando `fpprintprec` tiver um valor entre 2 e 16 (inclusive), o n;umero de dígitos mostrado na tela é igual a `fpprintprec`. De outra forma, `fpprintprec` é 0, ou maior que 16, e o número de dígitos mostrados é 16.

Para grandes números em ponto flutuante, quando `fpprintprec` tiver um valor entre 2 e `fpprec` (inclusive), o n;umero de dígitos mostrados é giaul a `fpprintprec`. De outra forma, `fpprintprec` é 0, ou maior que `fpprec`, e o n;umero de dígitos mostrados é igual a `fpprec`.

`fpprintprec` não pode ser 1.

11 Contextos

11.1 Definições para Contextos

`activate` (*context_1*, ..., *context_n*) [Função]

Ativa os contextos *context_1*, ..., *context_n*. Os factos nesses contextos estão então disponíveis para fazer deduções e recuperar informação. Os factos nesses contextos não são listadas através de `facts ()`.

A variável `activecontexts` é a lista de contextos que estão activos pelo caminho da função `activate`.

`activecontexts` [Variável de sistema]

Valor por omissão: `[]`

`activecontexts` é a lista de contextos que estão activos pelo caminho da função `activate`, em oposição a sendo activo porque eles são subcontextos do contexto corrente.

`assume` (*pred_1*, ..., *pred_n*) [Função]

Adiciona predicados *pred_1*, ..., *pred_n* ao contexto corrente. Se um predicado for incossistente ou redundante com os predicados no contexto corrente, esses predicados não são adicionados ao contexto. O contexto acumula predicados de cada chamada a `assume`.

`assume` retorna uma lista cujos elementos são os predicados adicionados ao contexto ou os átomos `redundant` ou `inconsistent` onde for aplicável.

Os predicados *pred_1*, ..., *pred_n* podem somente ser expressões com os operadores relacionais `<` `<=` `equal` `notequal` `>=` e `>`. Predicados não podem ser expressões de igualdades literais `=` ou expressões de desigualdades literais `#`, nem podem elas serem funções de predicado tais como `integerp`.

Predicados combinados da forma *pred_1* `and ...` `and` *pred_n* são reconhecidos, mas não *pred_1* `or ...` `or` *pred_n*. `not` *pred_k* é reconhecidos se *pred_k* for um predicado relacional. Expressões da forma `not (`*pred_1* `e` *pred_2*`)` and `not (`*pred_1* `or` *pred_2*`)` não são reconhecidas.

O mecanismo de dedução do Maxima não é muito forte; exitem consequências muito óbvias as quais não podem ser determinadas por meio de `is`. Isso é uma fraqueza conhecida.

`assume` avalia seus argumentos.

Veja também `is`, `facts`, `forget`, `context`, e `declare`.

Exemplos:

```
(%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` [Variável de opção]

Valor por omissão: `true`

`assumescalar` ajuda a governar se expressões `expr` para as quais `nonscalarp (expr)` for `false` são assumidas comportar-se como escalares para certas transformações.

Tomemos `expr` representando qualquer expressão outra que não uma lista ou uma matriz, e tomemos `[1, 2, 3]` representando qualquer lista ou matriz. Então `expr . [1, 2, 3]` retorna `[expr, 2 expr, 3 expr]` se `assumescalar` for `true`, ou `scalarp (expr)` for `true`, ou `constantp (expr)` for `true`.

Se `assumescalar` for `true`, tais expressões irão comportar-se como escalares somente para operadores comutativos, mas não para multiplicação não comutativa `..`

Quando `assumescalar` for `false`, tais expressões irão comportar-se como não escalares.

Quando `assumescalar` for `all`, tais expressões irão comportar-se como escalares para todos os operadores listados acima.

`assume_pos` [Variável de opção]

Valor por omissão: `false`

Quando `assume_pos` for `true` e o sinal de um parâmetro x não pode ser determinado a partir do contexto corrente ou outras considerações, `sign` e `asksign (x)` retornam `true`. Isso pode impedir algum questionamento de `asksign` gerado automaticamente, tal como pode surgir de `integrate` ou de outros cálculos.

Por padrão, um parâmetro é x tal como `symbolp (x)` or `subvarp (x)`. A classe de expressões consideradas parâmetros pode ser modificada para alguma abrangência através da variável `assume_pos_pred`.

`sign` e `asksign` tentam deduzir o sinal de expressões a partir de sinais de operandos dentro da expressão. Por exemplo, se `a` e `b` são ambos positivos, então `a + b` é também positivo.

Todavia, não existe caminho para desviar todos os questionamentos de `asksign`. Particularmente, quando o argumento de `asksign` for uma diferença `x - y` ou um logaritmo `log(x)`, `asksign` sempre solicita uma entrada ao utilizador, mesmo quando

`assume_pos` for `true` e `assume_pos_pred` for uma função que retorna `true` para todos os argumentos.

`assume_pos_pred` [Variável de opção]

Valor por omissão: `false`

Quando `assume_pos_pred` for atribuído o nome de uma função ou uma expressão lambda de um argumento x, aquela função é chamada para determinar se x é considerado um parâmetro para o propósito de `assume_pos`. `assume_pos_pred` é ignorado quando `assume_pos` for `false`.

A função `assume_pos_pred` é chamada através de `sign` e de `asksign` com um argumento x que é ou um átomo, uma variável subscrita, ou uma expressão de chamada de função. Se a função `assume_pos_pred` retorna `true`, x é considerado um parâmetro para o propósito de `assume_pos`.

Por padrão, um parâmetro é x tal que `symbolp (x)` ou `subvarp (x)`.

Veja também `assume` e `assume_pos`.

Exemplos:

```
(%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` [Variável de opção]

Valor por omissão: `initial`

`context` nomeia a colecção de factos mantida através de `assume` e `forget`. `assume` adiciona factos à colecção nomeada através de `context`, enquanto `forget` remove factos.

Associando `context` para um nome *foo* altera o contexto corrente para *foo*. Se o contexto especificado *foo* não existe ainda, ele é criado automaticamente através de uma chamada a `newcontext`. O contexto especificado é activado automaticamente.

Veja `contexts` para uma descrição geral do mecanismo de contexto.

`contexts` [Variável de opção]

Valor por omissão: `[initial, global]`

`contexts` é uma lista dos contextos que existem actualmente, incluindo o contexto activo actualmente.

O mecanismo de contexto torna possível para um utilizador associar e nomear uma porção seleccionada de factos, chamada um contexto. Assim que isso for concluído, o utilizador pode ter o Maxima assumindo ou esquecendo grande quantidade de factos meramente através da activação ou desativação seu contexto.

Qualquer átomo simbólico pode ser um contexto, e os factos contidos naquele contexto irão ser retidos em armazenamento até que sejam destruídos um por um através de chamadas a `forget` ou destruídos com um conjunto através de uma chamada a `kill` para destruir o contexto que eles pertencem.

Contextos existem em uma hierarquía, com o raíz sempre sendo o contexto `global`, que contém informações sobre Maxima que alguma função precisa. Quando em um contexto dado, todos os factos naquele contexto estão "ativos" (significando que eles são usados em deduções e recuperados) como estão também todos os factos em qualquer contexto que for um subcontexto do contexto activo.

Quando um novo Maxima for iniciado, o utilizador está em um contexto chamado `initial`, que tem `global` como um subcontexto.

Veja também `facts`, `newcontext`, `supcontext`, `killcontext`, `activate`, `deactivate`, `assume`, e `forget`.

`deactivate` (*context_1*, ..., *context_n*) [Função]

Desativa os contextos especificados *context_1*, ..., *context_n*.

`facts` (*item*) [Função]

`facts` () [Função]

Se *item* for o nome de um contexto, `facts` (*item*) retorna uma lista de factos no contexto especificado.

Se *item* não for o nome de um contexto, `facts` (*item*) retorna uma lista de factos conhecidos sobre *item* no contexto actual. Fatos que estão atuvos, mas em um diferente contexto, não são listados.

`facts` () (i.e., sem argumento) lista o contexto actual.

`features` [Declaração]

Maxima recnhece ceertas propriedades matemáticas de funções e variáveis. Essas são chamadas "recursos".

`declare` (*x*, *foo*) fornece a propriedade *foo* para a função ou variável *x*.

`declare` (*foo*, `recurso`) declara um novo recurso *foo*. Por exemplo, `declare` (`[red, green, blue], feature`) declara três novos recursos, `red`, `green`, e `blue`.

O predicado `featurep` (*x*, *foo*) retorna `true` se *x* possui a propriedade *foo*, e `false` de outra forma.

A infolista `features` é uma lista de recursos conhecidos. São esses `integer`, `noninteger`, `even`, `odd`, `rational`, `irrational`, `real`, `imaginary`, `complex`, `analytic`, `increasing`, `decreasing`, `oddfun`, `evenfun`, `posfun`, `commutative`, `lassociative`, `rassociative`, `symmetric`, e `antisymmetric`, mais quaisquer recursos definidos pelo utilizador.

`features` é uma lista de recursos matemáticos. Existe também uma lista de recursos não matemáticos, recursos dependentes do sistema. Veja `status`.

`forget` (*pred_1*, ..., *pred_n*) [Função]

`forget` (*L*) [Função]

Remove predicados estabelecidos através de `assume`. Os predicados podem ser expressões equivalentes a (mas não necessáriamente idênticas a) esses prevamentes assumidos.

`forget` (*L*), onde *L* é uma lista de predicados, esquece cada item da lista.

`killcontext` (*context_1*, ..., *context_n*) [Função]

Mata os contextos *context_1*, ..., *context_n*.

Se um dos contextos estiver for o contexto actual, o novo contexto actual irá tornar-se o primeiro subcontexto disponível do contexto actual que não tiver sido morto. Se o primeiro contexto disponível não morto for `global` então `initial` é usado em seu lugar. Se o contexto `initial` for morto, um novo, porém vazio contexto `initial` é criado.

`killcontext` recusa-se a matar um contexto que estiver ativo actualmente, ou porque ele é um subcontexto do contexto actual, ou através do uso da função `activate`.

`killcontext` avalia seus argumentos. `killcontext` retorna `done`.

`newcontext` (*nome*) [Função]

Cria um novo contexto, porém vazio, chamado *nome*, que tem `global` como seu único subcontexto. O contexto recentemente criado torna-se o contexto activo actualmente.

`newcontext` avalia seu argumento. `newcontext` retorna *nome*.

`supcontext` (*nome*, *context*) [Função]
`supcontext` (*nome*) [Função]

Cria um novo contexto, chamado *nome*, que tem *context* como um subcontexto. *context* deve existir.

Se *context* não for especificado, o contexto actual é assumido.

12 Polinómios

12.1 Introdução a Polinómios

Polinómios são armazenados no Maxima ou na forma geral ou na forma de Expressões Racionais Canónicas (CRE). Essa última é uma forma padrão, e é usada internamente por operações tais como `factor`, `ratsimp`, e assim por diante.

Expressões Racionais Canónicas constituem um tipo de representação que é especialmente adequado para polinómios expandidos e funções racionais (também para polinómios parcialmente factorizados e funções racionais quando RATFAC for escolhida para `true`). Nessa forma CRE uma ordenação de variáveis (da mais para a menos importante) é assumida para cada expressão. Polinómios são representados recursivamente por uma lista consistindo da variável principal seguida por uma série de pares de expressões, uma para cada termo do polinómio. O primeiro membro de cada par é o expoente da variável principal naquele termo e o segundo membro é o coeficiente daquele termo que pode ser um número ou um polinómio em outra variável novamente respresentado nessa forma. Sendo assim a parte principal da forma CRE de 3*X^2-1 é (X 2 3 0 -1) e que a parte principal da forma CRE de 2*X*Y+X-3 é (Y 1 (X 1 2) 0 (X 1 1 0 -3)) assumindo Y como sendo a variável principal, e é (X 1 (Y 1 2 0 1) 0 -3) assumindo X como sendo a variável principal. A variável principal é usualmente determineda pela ordem alfabética reversa. As "variáveis" de uma expressão CRE não necessariamente devem ser atômicas. De facto qualquer subexpressão cujo principal operador não for + - * / or ^ com expoente inteiro será considerado uma "variável" da expressão (na forma CRE) na qual essa ocorrer. Por exemplo as variáveis CRE da expressão X+SIN(X+1)+2*SQRT(X)+1 são X, SQRT(X), e SIN(X+1). Se o utilizador não especifica uma ordem de variáveis pelo uso da função RATVARS Maxima escolherá a alfabética por conta própria. Em geral, CREs representam expressões racionais, isto é, razões de polinómios, onde o numerador e o denominador não possuem factores comuns, e o denominador for positivo. A forma interna é essencialmente um par de polinómios (o numerador e o denominador) precedidos pela lista de ordenação de variável. Se uma expressão a ser mostrada estiver na forma CRE ou se contiver quaisquer subexpressões na forma CRE, o símbolo /R/ seguirá o rótulo da linha. Veja a função RAT para saber como converter uma expressão para a forma CRE. Uma forma CRE extendida é usada para a representação de séries de Taylor. A noção de uma expressão racional é extendida de modo que os expoentes das variáveis podem ser números racionais positivos ou negativos em lugar de apenas inteiros positivos e os coeficientes podem eles mesmos serem expressões racionais como descrito acima em lugar de apenas polinómios. Estes são representados internamente por uma forma polinomial recursiva que é similar à forma CRE e é a generalização dessa mesma forma CRE, mas carrega informação adicional tal com o grau de truncação. Do mesmo modo que na forma CRE, o símbolo /T/ segue o rótulo de linha que contém as tais expressões.

12.2 Definições para Polinómios

`algebraic` [Variável de opção]

Valor Padrão: `false`

`algebraic` deve ser escolhida para `true` com o objectivo de que a simplificação de inteiros algébricos tenha efeito.

`berlefact` [Variável de opção]

Valor Padrão: `true`

Quando `berlefact` for `false` então o algoritmo de factorização de Kronecker será usado. De outra forma o algoritmo de Berlekamp, que é o padrão, será usado.

`bezout` (*p1*, *p2*, *x*) [Função]

uma alternativa para o comando `resultant`. Isso retorna uma matriz. `determinant` dessa matriz é o resultante desejado.

`bothcoef` (*expr*, *x*) [Função]

Retorna uma lista da qual o primeiro membro é o coeficiente de *x* em *expr* (como achado por `ratcoef` se *expr* está na forma CRE de outro modo por `coeff`) e cujo segundo membro é a parte restante de *expr*. Isto é, `[A, B]` onde *expr* = `A*x + B`.

Exemplo:

```
(%i1) islinear (expr, x) := block ([c],
        c: bothcoef (rat (expr, x), x),
        é (freeof (x, c) and c[1] # 0))$
(%i2) islinear ((r^2 - (x - r)^2)/x, x);
(%o2)                          true
```

`coeff` (*expr*, *x*, *n*) [Função]

Retorna o coeficiente de *x*`^`*n* em *expr*. *n* pode ser omitido se for 1. *x* pode ser um átomo, ou subexpressão completa de *expr* e.g., `sin(x)`, `a[i+1]`, `x + y`, etc. (No último caso a expressão `(x + y)` pode ocorrer em *expr*). Algumas vezes isso pode ser necessário para expandir ou factorizar *expr* com o objectivo de fazer *x*`^`*n* explicito. Isso não é realizado por `coeff`.

Exemplos:

```
(%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*) [Função]

Simplifica a adição *expr* por termos combinados com o mesmo denominador dentro de um termo simples.

`content` (*p_1*, *x_1*, ..., *x_n*) [Função]

Retorna uma lista cujo primeiro elemento é o máximo divisor comum dos coeficientes dos termos do polinómio *p_1* na variável *x_n* (isso é o conteúdo) e cujo segundo elemento é o polinómio *p_1* dividido pelo conteúdo.

Exemplos:

```
(%i1) content (2*x*y + 4*x^2*y^2, y);
                                     2
(%o1)                    [2 x, 2 x y  + y]
```

`denom` (*expr*) [Função]

Retorna o denominador da expressão racional *expr*.

`divide` (*p_1*, *p_2*, *x_1*, ..., *x_n*) [Função]

calcula o quocietne e o resto do polinómio *p_1* dividido pelo polinómio *p_2*, na variável principal do polinómio, *x_n*. As outras variáveis são como na função `ratvars`. O resultado é uma lista cujo primeiro elemento é o quociente e cujo segundo elemento é o resto.

Exemplos:

```
(%i1) divide (x + y, x - y, x);
(%o1)                         [1, 2 y]
(%i2) divide (x + y, x - y);
(%o2)                        [- 1, 2 x]
```

Note que `y` é a variável principal no segundo exemplo.

`eliminate` ([*eqn_1*, ..., *eqn_n*], [*x_1*, ..., *x_k*]) [Função]

Elimina variáveis de equações (ou expressões assumidas iguais a zero) obtendo resultantes sucessivos. Isso retorna uma lista de `n - k` expressões com *k* variáveis *x_1*, ..., *x_k* eliminadas. Primeiro *x_1* é eliminado retornando `n - 1` expressões, então `x_2` é eliminado, etc. Se `k = n` então uma expressão simples em uma lista é retornada livre das variáveis *x_1*, ..., *x_k*. Nesse caso `solve` é chamado para resolver a última resultante para a última variável.

Exemplo:

```
(%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*, ...) [Função]

Retorna uma lista cujo primeiro elemento é o m.d.c. dos polinómios *p_1*, *p_2*, *p_3*, ... e cujos restantes elementos são os polinómios divididos pelo mdc. Isso sempre usa o algoritmo `ezgcd`.

`facexpand` [Variável de opção]

Valor Padrão: `true`

`facexpand` controla se os factores irredutíveis retornados por `factor` estão na forma expandida (o padrão) ou na forma recursiva (CRE normal).

`factcomb` (*expr*) [Função]

Tenta combinar os coeficientes de factoriais em *expr* com os próprios factoriais convertendo, por exemplo, `(n + 1)*n!` em `(n + 1)!`.

`sumsplitfact` se escolhida para `false` fará com que `minfactorial` seja aplicado após um `factcomb`.

`factor` (*expr*) [Função]
`factor` (*expr*, *p*) [Função]

Factoriza a expressão *expr*, contendo qualquer número de variáveis ou funções, em factores irredutíveis sobre os inteiros. `factor (expr, p)` factoriza *expr* sobre o campo dos inteiros com um elemento adjunto cujo menor polinómio é *p*.

`factor` usa a função `ifactors` para factorizar inteiros.

`factorflag` se `false` suprime a factorização de factores inteiros de expressões racionais.

`dontfactor` pode ser escolhida para uma lista de variáveis com relação à qual factorização não é para ocorrer. (Essa é inicialmente vazia). Factorização também não acontece com relação a quaisquer variáveis que são menos importantes (usando a ordenação de variável assumida pela forma CRE) como essas na lista `dontfactor`.

`savefactors` se `true` faz com que os factores de uma expressão que é um produto de factores seja guardada por certas funções com o objectivo de aumentar a velocidade de futuras factorizações de expressões contendo alguns dos mesmos factores.

`berlefact` se `false` então o algoritmo de factorização de Kronecker será usado de outra forma o algoritmo de Berlekamp, que é o padrão, será usado.

`intfaclim` se `true` maxima irá interromper a factorização de inteiros se nenhum factor for encontrado após tentar divisões e o método rho de Pollard. Se escolhida para `false` (esse é o caso quando o utilizador chama `factor` explicitamente), a factorização completa do inteiro será tentada. A escolha do utilizador para `intfaclim` é usada para chamadas internas a `factor`. Dessa forma, `intfaclim` pode ser resetada para evitar que o Maxima gaste um tempo muito longo factorizando inteiros grandes.

Exemplos:

```
(%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` [Variável de opção]

Valor Padrão: `false`

Quando `factorflag` for `false`, suprime a factorização de factores inteiros em expressões racionais.

`factorout` (*expr*, *x_1*, *x_2*, ...) [Função]

Rearranja a adição *expr* em uma adição de parcelas da forma `f (x_1, x_2, ...)*g` onde `g` é um produto de expressões que não possuem qualquer *x_i* e `f` é factorizado.

`factorsum` (*expr*) [Função]

Tenta agrupar parcelas em factores de *expr* que são adições em grupos de parcelas tais que sua adição é factorável. `factorsum` pode recuperar o resultado de `expand ((x + y)^2 + (z + w)^2)` mas não pode recuperar `expand ((x + 1)^2 + (x + y)^2)` porque os termos possuem variáveis em comum.

Exemplo:

```
(%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*) [Função]

Retorna o produto dos polinómios *p_1* e *p_2* usando um algoritmo especial para a multiplicação de polinómios. `p_1` e `p_2` podem ser de várias variáveis, densos, e aproximadamente do mesmo tamanho. A multiplicação clássica é de ordem `n_1 n_2` onde `n_1` é o grau de `p_1` and `n_2` é o grau de `p_2`. `fasttimes` é da ordem `max (n_1, n_2)^1.585`.

`fullratsimp` (*expr*) [Função]

`fullratsimp` aplica repetidamente `ratsimp` seguido por simplificação não racional a uma expressão até que nenhuma mudança adicional ocorra, e retorna o resultado.

Quando expressões não racionais estão envolvidas, uma chamada a `ratsimp` seguida como é usual por uma simplificação não racional ("geral") pode não ser suficiente para retornar um resultado simplificado. Algumas vezes, mais que uma tal chamada pode ser necessária. `fullratsimp` faz esse processo convenientemente.

`fullratsimp` (*expr*, *x_1*, ..., *x_n*) aceita um ou mais argumentos similar a `ratsimp` e `rat`.

Exemplo:

```
(%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*) [Função]

é o mesmo que `ratsubst` excepto que essa chama a si mesma recursivamente sobre esse resultado até que o resultado para de mudar. Essa função é útil quando a expressão de substituição e a expressão substituída tenham uma ou mais variáveis em comum.

`fullratsubst` irá também aceitar seus argumentos no formato de `lratsubst`. Isto é, o primeiro argumento pode ser uma substituição simples de equação ou uma lista de tais equações, enquanto o segundo argumento é a expressão sendo processada.

`load ("lrats")` chama `fullratsubst` e `lratsubst`.

Exemplos:

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

- `subst` pode realizar multiplas substituições. `lratsubst` é analogo a `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
```

- Se somente uma substituição é desejada, então uma equação simples pode ser dada como primeiro argumento.

```
(%i4) lratsubst (a^2 = b, a^3);
(%o4)                           a b
```

- `fullratsubst` é equivalente a `ratsubst` excepto que essa executa recursivamente até que seu resultado para de mudar.

```
(%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` também aceita uma lista de equações ou uma equação simples como primeiro argumento.

```
(%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` pode causar uma recursão infinita.

```
(%i9) errcatch (fullratsubst (b*a^2, a^2, a^3));

*** - Lisp stack overflow. RESET
```

`gcd` (*p_1*, *p_2*, *x_1*, ...) [Função]

Retorna o máximo divisor comum entre *p_1* e *p_2*. O sinalizador `gcd` determina qual algoritmo é empregado. Escolhendo `gcd` para `ez`, `subres`, `red`, ou `spmod` selecciona o algoritmo `ezgcd`, subresultante `prs`, reduzido, ou modular, respectivamente. Se `gcd` for `false` então `gcd (p_1, p_2, x)` sempre retorna 1 para todo *x*. Muitas funções

(e.g. `ratsimp`, `factor`, etc.) fazem com que mdc's sejam feitos implicitamente. Para polinómios homogêneos é recomendado que `gcd` igual a `subres` seja usado. Para obter o mdc quando uma expressão algébrica está presente, e.g. `gcd (x^2 - 2*sqrt(2)*x + 2, x - sqrt(2))`, `algebraic` deve ser `true` e `gcd` não deve ser `ez`. `subres` é um novo algoritmo, e pessoas que tenham estado usando a opção `red` podem provavelmente alterar isso para `subres`.

O sinalizador `gcd`, padrão: `subres`, se `false` irá também evitar o máximo divisor comum de ser usado quando expressões são convertidas para a forma de expressão racional canónica (CRE). Isso irá algumas vezes aumentar a velocidade dos cálculos se mdc's não são requeridos.

`gcdex (`*f*`, `*g*`)` [Função]
`gcdex (`*f*`, `*g*`, `*x*`)` [Função]

Retornam uma lista `[a, b, u]` onde *u* é o máximo divisor comum (mdc) entre *f* e *g*, e *u* é igual a `a` *f* `+ b` *g*. Os argumentos *f* e *g* podem ser polinómios de uma variável, ou de outra forma polinómios em *x* uma **main**(principal) variável suprida desde que nós precisamos estar em um domínio de ideal principal para isso trabalhar. O mdc significa o mdc considerando *f* e *g* como polinómios de uma única variável com coeficientes sendo funções racionais em outras variáveis.

`gcdex` implementa o algoritmo Euclideano, onde temos a sequência of `L[i]: [a[i], b[i], r[i]]` que são todos perpendiculares a `[f, g, -1]` e o próximo se é construído como se `q = quotient(r[i]/r[i+1])` então `L[i+2]: L[i] - q L[i+1]`, e isso encerra em `L[i+1]` quando o resto `r[i+2]` for 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 que o mdc adiante é `1` uma vez que trabalhamos em `k(y)[x]`, o `y+1` não pode ser esperado em `k[y, x]`.

```
(%i1) gcdex (x*(y + 1), y^2 - 1, x);
                                1
(%o1)/R/                 [0, ------, 1]
                              2
                             y  - 1
```

`gcfactor (`*n*`)` [Função]

Factoriza o inteiro Gaussiano *n* sobre os inteiros Gaussianos, i.e., números da forma `a + b %i` onde *a* e *b* são inteiros raconais (i.e., inteiros comuns). Factorizações são normalizadas fazendo *a* e *b* não negativos.

`gfactor (`*expr*`)` [Função]

Factoriza o polinómio *expr* sobre os inteiros de Gauss (isto é, os inteiros com a unidade imaginária `%i` adjunta). Isso é como `factor (`*expr*`, a^2+1)` trocando *a* por `%i`.

Exemplo:

```
(%i1) gfactor (x^4 - 1);
(%o1)           (x - 1) (x + 1) (x - %i) (x + %i)
```

`gfactorsum` (*expr*) [Função]

é similar a `factorsum` mas aplica `gfactor` em lugar de `factor`.

`hipow` (*expr*, *x*) [Função]

Retorna o maior expoente explícito de *x* em *expr*. *x* pode ser uma variável ou uma expressão geral. Se *x* não aparece em *expr*, `hipow` retorna `0`.

`hipow` não considera expressões equivalentes a `expr`. Em particular, `hipow` não expande `expr`, então `hipow (expr, x)` e `hipow (expand (expr, x))` podem retornar diferentes resultados.

Exemplos:

```
(%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` [Variável de opção]

Valor por omissão: true

Se `true`, maxima irá interromper a factorização de inteiros se nenhum factor for encontrado após tentar divisões e o método rho de Pollard e a factorização não será completada.

Quando `intfaclim` for `false` (esse é o caso quando o utilizador chama `factor` explicitamente), a factorização completa será tentada. `intfaclim` é escolhida para `false` quando factores são calculados em `divisors`, `divsum` e `totient`.

Chamadas internas a `factor` respeitam o valor especificado pelo utilizador para `intfaclim`. Setting `intfaclim` to `true` may reduce `intfaclim`. Escolhendo `intfaclim` para `true` podemos reduzir o tempo gasto factorizando grandes inteiros.

`keepfloat` [Variável de opção]

Valor Padrão: `false`

Quando `keepfloat` for `true`, evitamos que números em ponto flutuante sejam racionalizados quando expressões que os possuem são então convertidas para a forma de expressão racional canónica (CRE).

`lratsubst` (*L*, *expr*) [Função]

é análogo a `subst (L, expr)` excepto que esse usa `ratsubst` em lugar de `subst`.

O primeiro argumento de `lratsubst` é uma equação ou uma lista de equações idênticas em formato para que sejam aceitas por `subst`. As substituições são feitas na ordem dada pela lista de equações, isto é, da esquerda para a direita.

`load ("lrats")` chama `fullratsubst` e `lratsubst`.

Exemplos:

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

- `subst` pode realizar multiplas substituições. `lratsubst` é analoga a `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
```

- Se somente uma substituição for desejada, então uma equação simples pode ser dada como primeiro argumento.

```
(%i4) lratsubst (a^2 = b, a^3);
(%o4)                           a b
```

`modulus` [Variável de opção]

Valor Padrão: `false`

Quando `modulus` for um número positivo *p*, operações sobre os números racionais (como retornado por `rat` e funções relacionadas) são realizadas módulo *p*, usando o então chamado sistema de módulo "balanceado" no qual *n* `módulo` *p* é definido como um inteiro *k* em `[-(`*p*`-1)/2, ..., 0, ..., (`*p*`-1)/2]` quando *p* for ímpar, ou `[-(`*p*`/2 - 1), ..., 0, ...., `*p*`/2]` quando *p* for par, tal que `a` *p* `+` *k* seja igual a *n* para algum inteiro *a*.

Se *expr* já estiver na forma de expressão racional canónica (CRE) quando `modulus` for colocado no seu valor original, então pode precisar repetir o rat *expr*, e.g., `expr: rat (ratdisrep (expr))`, com o objectivo de obter resultados correctos.

Tipicamente `modulus` é escolhido para um número primo. Se `modulus` for escolhido para um inteiro não primo positivo, essa escolha é aceita, mas uma mensagem de alerta é mostrada. Maxima permitirá que zero ou um inteiro negativo seja atribuído a `modulus`, embora isso não seja limpo se aquele tiver quaisquer consequências úteis.

`num` (*expr*) [Função]

Retorna o numerador de *expr* se isso for uma razão. Se *expr* não for uma razão, *expr* é retornado.

`num` avalia seu argumento.

`polydecomp` (*p*, *x*) [Função]

Decompões o polinómio *p* na variável *x* em uma composição funcional de polinómios em *x*. `polydecomp` retorna uma lista `[`*p_1*`, ..., `*p_n*`]` tal que

```
lambda ([x], p_1) (lambda ([x], p_2) (... (lambda ([x], p_n) (x)) ...))
```

seja igual a *p*. O grau de *p_i* é maior que 1 para *i* menor que *n*.

Tal decomposição não é única.

Exemplos:

```
(%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]
```

As seguintes funções compõem `L = [e_1, ..., e_n]` como funções em `x`; essa funçào é a inversa de `polydecomp`:

```
compose (L, x) :=
  block ([r : x], for e in L do r : subst (e, x, r), r) $
```

Re-exprimindo o exemplo acima usando `compose`:

```
(%i3) polydecomp (compose ([x^2 - a, x^3 - x - 1], x), x);
                             2       3
(%o3)                   [x  - a, x  - x - 1]
```

Note que apesar de `compose (polydecomp (`*p*`, `*x*`), `*x*`)` sempre retornar *p* (não expandido), `polydecomp (compose ([`*p_1*`, ..., `*p_n*`], `*x*`), `*x*`)` *não* necessáriamente retorna `[`*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*`)` [Função]
`quotient (`*p_1*`, `*p_2*`, `*x_1*`, ..., `*x_n*`)` [Função]

Retorna o polinómio *p_1* dividido pelo polinómio *p_2*. Os argumentos *x_1*, ..., *x_n* são interpretados como em `ratvars`.

`quotient` retorna o primeiro elemento de uma lista de dois elementos retornada por `divide`.

`rat (`*expr*`)` [Função]
`rat (`*expr*`, `*x_1*`, ..., `*x_n*`)` [Função]

Converte *expr* para a forma de expressão racional canónica (CRE) expandindo e combinando todos os termos sobre um denominador comum e cancelando para fora o máximo divisor comum entre o numerador e o denominador, também convertendo números em ponto flutuante para números racionais dentro da tolerância de `ratepsilon`. As variáveis são ordenadas de acordo com *x_1*, ..., *x_n*, se especificado, como em `ratvars`.

`rat` geralmente não simplifica funções outras que não sejam adição `+`, subtração `-`, multiplicação `*`, divisão `/`, e exponenciação com expoente inteiro, uma vez que `ratsimp` não manuseia esses casos. Note que átomos (números e variáveis) na forma

CRE não são os mesmos que eles são na forma geral. Por exemplo, `rat(x)- x` retorna `rat(0)` que tem uma representação interna diferente de 0.

Quando `ratfac` for `true`, `rat` retorna uma forma parcialmente factorizada para CRE. Durante operações racionais a expressão é mantida como totalmente factorizada como possível sem uma chamada ao pacote de factorização (`factor`). Isso pode sempre economizar espaço de memória e algum tempo em algumas computações. O numerador e o denominador são ainda tidos como relativamente primos (e.g. `rat ((x^2 - 1)^4/(x + 1)^2)` retorna `(x - 1)^4 (x + 1)^2)`, mas os factores dentro de cada parte podem não ser relativamente primos.

`ratprint` se `false` suprime a impressão de mensagens informando o utilizador de conversões de números em ponto flutuante para números racionais.

`keepfloat` se `true` evita que números em ponto flutuante sejam convertidos para números racionais.

Veja também `ratexpand` e `ratsimp`.

Exemplos:

```
(%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` [Variável de opção]

Valor Padrão: `true`

Quando `ratalgdenom` for `true`, permite racionalização de denominadores com respeito a radicais tenham efeito. `ratalgdenom` tem efeito somente quando expressões racionais canónicas (CRE) forem usadas no modo algébrico.

`ratcoef` (*expr*, *x*, *n*) [Função]

`ratcoef` (*expr*, *x*) [Função]

Retorna o coeficiente da expressão `x^n` dentro da expressão *expr*. Se omitido, *n* é assumido ser 1.

O valor de retorno está livre (excepto possivelmente em um senso não racional) das variáveis em *x*. Se nenhum coeficiente desse tipo existe, 0 é retornado.

`ratcoef` expande e simplifica racionalmente seu primeiro argumento e dessa forma pode produzir respostas diferentes das de `coeff` que é puramente sintática. Dessa forma `ratcoef ((x + 1)/y + x, x)` retorna `(y + 1)/y` ao passo que `coeff` retorna 1.

`ratcoef (expr, x, 0)`, visualiza *expr* como uma adição, retornando uma soma desses termos que não possuem *x*. portanto se *x* ocorre para quaisquer expoentes negativos, `ratcoef` pode não ser usado.

Uma vez que *expr* é racionalmente simplificada antes de ser examinada, coeficientes podem não aparecer inteiramente no caminho que eles foram pensados.

Exemplo:

```
(%i1) s: a*x + b*x + 5$
(%i2) ratcoef (s, a + b);
(%o2)                           x
```

`ratdenom (expr)` [Função]

Retorna o denominador de *expr*, após forçar a conversão de *expr* para expressão racional canónica (CRE). O valor de retorno é a CRE.

expr é forçada para uma CRE por `rat` se não for já uma CRE. Essa conversão pode mudar a forma de *expr* colocando todos os termos sobre um denominador comum.

`denom` é similar, mas retorna uma expressão comum em lugar de uma CRE. Também, `denom` não tenta colocar todos os termos sobre um denominador comum, e dessa forma algumas expressões que são consideradas razões por `ratdenom` não são consideradas razões por `denom`.

`ratdenomdivide` [Variável de opção]

Valor Padrão: `true`

Quando `ratdenomdivide` for `true`, `ratexpand` expande uma razão cujo o numerador for uma adição dentro de uma soma de razões, tendo todos um denominador comum. De outra forma, `ratexpand` colapsa uma adição de razões dentro de uma razão simples, cujo numerador seja a adição dos numeradores de cada razão.

Exemplos:

```
(%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*) [Função]

Realiza a derivação da expressão racional *expr* com relação a *x*. *expr* deve ser uma razão de polinómios ou um polinómio em *x*. O argumento *x* pode ser uma variável ou uma subexpressão de *expr*.

O resultado é equivalente a `diff`, embora talvez em uma forma diferente. `ratdiff` pode ser mais rápida que `diff`, para expressões racionais.

`ratdiff` retorna uma expressão racional canónica (CRE) se `expr` for uma CRE. De outra forma, `ratdiff` retorna uma expressão geral.

`ratdiff` considera somente as dependências de *expr* sobre *x*, e ignora quaisquer dependências estabelecidas por `depends`.

Exemplo:

```
(%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*) [Função]

Retorna seu argumento como uma expressão geral. Se *expr* for uma expressão geral, é retornada inalterada.

Tipicamente `ratdisrep` é chamada para converter uma expressão racional canónica (CRE) em uma expressão geral. Isso é algumas vezes conveniente se deseja-se parar o "contágio", ou caso se esteja usando funções racionais em contextos não racionais.

Veja também `totaldisrep`.

`ratepsilon` [Variável de opção]

Valor Padrão: 2.0e-8

`ratepsilon` é a tolerância usada em conversões de números em ponto flutuante para números racionais.

`ratexpand` (*expr*) [Função]

`ratexpand` [Variável de opção]

Expande *expr* multiplicando para fora produtos de somas e somas exponenciadas, combinando frações sobre um denominador comum, cancelando o máximo divisor comum entre entre o numerador e o denominador, então quebrando o numerador (se for uma soma) dentro de suas respectivas parcelas divididas pelo denominador.

O valor de retorno de `ratexpand` é uma expressão geral, mesmo se *expr* for uma expressão racional canónica (CRE).

O comutador `ratexpand` se `true` fará com que expressões CRE sejam completamente expandidas quando forem convertidas de volta para a forma geral ou mostradas, enquanto se for `false` então elas serão colocadas na forma recursiva. Veja também `ratsimp`.

Quando `ratdenomdivide` for `true`, `ratexpand` expande uma razão na qual o numerador é uma adição dentro de uma adição de razões, todas tendo um denominador comum. De outra forma, `ratexpand` contrai uma soma de razões em uma razão simples, cujo numerador é a soma dos numeradores de cada razão.

Quando `keepfloat` for `true`, evita que números em ponto flutuante sejam racionalizados quando expressões que contenham números em ponto flutuante forem convertidas para a forma de expressão racional canónica (CRE).

Exemplos:

```
(%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` [Variável de opção]

Valor Padrão: `false`

Quando `ratfac` for `true`, expressões racionais canónicas (CRE) são manipuladas na forma parcialmente factorizada.

Durante operações racionais a expressão é mantida como completamente factorizada como foi possível sem chamadas a `factor`. Isso pode sempre economizar espaço e pode economizar tempo em algumas computações. O numerador e o denominador são feitos relativamente primos, por exemplo `rat ((x^2 - 1)^4/(x + 1)^2)` retorna `(x - 1)^4 (x + 1)^2)`, mas o factor dentro de cada parte pode não ser relativamente primo.

No pacote `ctensor` (Manipulação de componentes de tensores), tensores de Ricci, Einstein, Riemann, e de Weyl e a curvatura escalar são factorizados automaticamente quando `ratfac` for `true`. `ratfac` *pode somente ser escolhido para casos onde as componentes tensoriais sejam sabidametne consistidas de poucos termos.*

Os esquemas de `ratfac` e de `ratweight` são incompatíveis e não podem ambos serem usados ao mesmo tempo.

`ratnumer` (*expr*) [Função]

Retorna o numerador de *expr*, após forçar *expr* para uma expressão racional canónica (CRE). O valor de retorno é uma CRE.

expr é forçada para uma CRE por `rat` se isso não for já uma CRE. Essa conversão pode alterar a forma de *expr* pela colocação de todos os termos sobre um denominador comum.

`num` é similar, mas retorna uma expressão comum em lugar de uma CRE. Também, `num` não tenta colocar todos os termos sobre um denominador comum, e dessa forma algumas expressões que são consideradas razões por `ratnumer` não são consideradas razões por `num`.

`ratnump` (*expr*) [Função]

Retorna `true` se *expr* for um inteiro literal ou razão de inteiros literais, de outra forma retorna `false`.

`ratp` (*expr*) [Função]

Retorna `true` se *expr* for uma expressão racional canónica (CRE) ou CRE extendida, de outra forma retorna `false`.

CRE são criadas por `rat` e funções relacionadas. CRE extendidas são criadas por `taylor` e funções relacionadas.

`ratprint` [Variável de opção]

Valor Padrão: `true`

Quando `ratprint` for `true`, uma mensagem informando ao utilizador da conversão de números em ponto flutuante para números racionais é mostrada.

`ratsimp` (*expr*) [Função]
`ratsimp` (*expr*, *x_1*, ..., *x_n*) [Função]

Simplifica a expressão *expr* e todas as suas subexpressões, incluindo os argumentos para funções não racionais. O resultado é retornado como o quociente de dois polinómios na forma recursiva, isto é, os coeficientes de variável principal são polinómios em outras variáveis. Variáveis podem incluir funções não racionais (e.g., `sin (x^2 + 1)`) e os argumentos para quaisquer tais funções são também simplificados racionalmente.

`ratsimp (`*expr*`, `*x_1*`, ..., `*x_n*`)` habilita simplificação racional com a especiicação de variável ordenando como em `ratvars`.

Quando `ratsimpexpons` for `true`, `ratsimp` é aplicado para os expoentes de expressões durante a simplificação.

Veja também `ratexpand`. Note que `ratsimp` é afectado por algum dos sinalizadores que afectam `ratexpand`.

Exemplos:

```
(%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` [Variável de opção]

Valor Padrão: `false`

Quando `ratsimpexpons` for `true`, `ratsimp` é aplicado para os expoentes de expressões durante uma simplificação.

`ratsubst (a, b, c)` [Função]

Substitue *a* por *b* em *c* e retorna a expressão resultante. *b* pode também ser uma adição, produto, expoente, etc.

`ratsubst` sabe alguma coisa do significado de expressões uma vez que `subst` não é uma substituição puramente sintática. Dessa forma `subst (a, x + y, x + y + z)` retorna `x + y + z` ao passo que `ratsubst` retorna `z + a`.

Quando `radsubstflag` for `true`, `ratsubst` faz substituição de radicais em expressões que explicitamente não possuem esses radicais.

Exemplos:

```
(%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);
                               x
(%o6)
(%i7) radsubstflag: true$
(%i8) ratsubst (u, sqrt(x), x);
                                2
(%o8)                          u
```

`ratvars (x_1, ..., x_n)` [Função]

`ratvars ()` [Função]

`ratvars` [Variável de sistema]

Declara variáveis principais *x_1*, ..., *x_n* para expressões racionais. *x_n*, se presente em uma expressão racional, é considerada a variável principal. De outra forma, *x_[n-1]* é considerada a variável principal se presente, e assim por diante até as variáveis precedentes para *x_1*, que é considerada a variável principal somente se nenhuma das variáveis que a sucedem estiver presente.

Se uma variável em uma expressão racional não está presente na lista `ratvars`, a ela é dada uma prioridade menor que *x_1*.

Os argumentos para `ratvars` podem ser ou variáveis ou funções não racionais tais como `sin(x)`.

A variável `ratvars` é uma lista de argumentos da função `ratvars` quando ela foi chamada mais recentemente. Cada chamada para a função `ratvars` sobre-grava a lista apagando seu conteúdo anterior. `ratvars ()` limpa a lista.

`ratweight` (*x_1*, *w_1*, ..., *x_n*, *w_n*) [Função]
`ratweight ()` [Função]

Atribui um peso *w_i* para a variável *x_i*. Isso faz com que um termo seja substituído por 0 se seu peso exceder o valor da variável `ratwtlvl` (o padrão retorna sem truncação). O peso de um termo é a soma dos produtos dos pesos de uma variável no termo vezes seu expoente. Por exemplo, o peso de `3 x_1^2 x_2` é `2 w_1 + w_2`. A truncação de acordo com `ratwtlvl` é realizada somente quando multiplicando ou exponencializando expressões racionais canónicas (CRE).

`ratweight ()` retorna a lista cumulativa de atribuições de pesos.

Nota: Os esquemas de `ratfac` e `ratweight` são incompatíveis e não podem ambo serem usados ao mesmo tempo.

Exemplos:

```
(%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` [Variável de sistema]

Valor Padrão: `[]`

`ratweights` é a lista de pesos atribuídos por `ratweight`. A lista é cumulativa: cada chamada a `ratweight` coloca ítens adicionais na lista.

`kill (ratweights)` e `save (ratweights)` ambos trabalham como esperado.

`ratwtlvl` [Variável de opção]

Valor Padrão: `false`

`ratwtlvl` é usada em combinação com a função `ratweight` para controlar a truncação de expressão racionais canónicas (CRE). Para o valor padrão `false`, nenhuma truncação ocorre.

`remainder` (*p_1*, *p_2*) [Função]
`remainder` (*p_1*, *p_2*, *x_1*, ..., *x_n*) [Função]

Retorna o resto do polinómio *p_1* dividido pelo polinómio *p_2*. Os argumentos *x_1*, ..., *x_n* são interpretados como em `ratvars`.

`remainder` retorna o segundo elemento de uma lista de dois elementos retornada por `divide`.

`resultant` (*p_1*, *p_2*, *x*) [Função]
`resultant` [Variável]

Calcula o resultante de dois polinómios *p_1* e *p_2*, eliminando a variável *x*. O resultante é um determinante dos coeficientes de *x* em *p_1* e *p_2*, que é igual a zero se e somente se *p_1* e *p_2* tiverem um factor em comum não constante.

Se *p_1* ou *p_2* puderem ser factorizados, pode ser desejável chamar `factor` antes de chamar `resultant`.

A variável `resultant` controla que algoritmo será usado para calcular o resultante. `subres` para o prs subresultante, `mod` para o algoritmo resultante modular, e `red` para prs reduzido. Para muitos problemas `subres` pode ser melhor. Para alguns problemas com valores grandes de grau de uma única variável ou de duas variáveis `mod` pode ser melhor.

A função `bezout` aceita os mesmos argumentos que `resultant` e retorna uma matriz. O determinante do valor de retorno é o resultante desejado.

`savefactors` [Variável de opção]

Valor Padrão: `false`

Quando `savefactors` for `true`, faz com que os factores de uma expressão que é um produto de factores sejam gravados por certas funções com o objectivo de aumentar a velocidade em posteriores factorizações de expressões contendo algum desses mesmos factores.

`sqfr` (*expr*) [Função]

é similar a `factor` excepto que os factores do polinómio são "livres de raízes". Isto é, eles possuem factores somente de grau um. Esse algoritmo, que é também usado no primeiro estágio de `factor`, utiliza o facto que um polinómio tem em comum com sua n'ésima derivada todos os seus factores de grau maior que n. Dessa forma obtendo o maior divisor comum com o polinómio das derivadas com relação a cada variável no polinómio, todos os factores de grau maior que 1 podem ser achados.

Exemplo:

```
(%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*) [Função]
`tellrat` () [Função]

Adiciona ao anel dos inteiros algébricos conhecidos do Maxima os elementos que são as soluções dos polinómios *p_1*, ..., *p_n*. Cada argumento *p_i* é um polinómio concoeficientes inteiros.

`tellrat (x)` efectivamente significa substituir 0 por *x* em funções racionais.

`tellrat ()` retorna uma lista das substituições correntes.

`algebraic` deve ser escolhida para `true` com o objectivo de que a simplificação de inteiros algébricos tenha efeito.

Maxima inicialmente sabe sobre a unidade imaginária `%i` e todas as raízes de inteiros.

Existe um comando `untellrat` que recebe núcleos e remove propriedades `tellrat`.

Quando fazemos `tellrat` em um polinómio de várias variáveis, e.g., `tellrat (x^2 - y^2)`, pode existir uma ambiguidade como para ou substituir `y^2` por `x^2` ou vice-versa. Maxima selecciona uma ordenação particular, mas se o utilizador desejar especificar qual e.g. `tellrat (y^2 = x^2)` forneçe uma sintaxe que diga para substituir `y^2` por `x^2`.

Exemplos:

```
(%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*) [Função]

Converte toda subexpressão de *expr* da forma de expressão racionais canónicas (CRE) para a forma geral e retorna o resultado. Se *expr* é em sí mesma na forma CRE então `totaldisrep` é identica a `ratdisrep`.

`totaldisrep` pode ser usada para fazer um `ratdisrep` em expressões tais como equações, listas, matrizes, etc., que tiverem algumas subexpressões na forma CRE.

`untellrat` (*x_1*, ..., *x_n*) [Função]

Remove propriedades `tellrat` de *x_1*, ..., *x_n*.

13 Constantes

13.1 Definições para Constantes

`%e` [Constante]

`%e` representa a base do logaritmo natural, também conhecido como constante de Euler. O valor numérico de `%e` é um número em ponto flutuante de precisão dupla 2.718281828459045d0.

`%i` [Constante]

`%i` representa a unidade imaginária, $sqrt(-1)$.

`false` [Constante]

`false` representa a constante Booleana falso. Maxima implementa `false` através do valor `NIL` no Lisp.

`inf` [Constante]

`inf` representa o infinito positivo real.

`infinity` [Constante]

`infinity` representa o infinito complexo.

`minf` [Constante]

`minf` representa o menos infinito (i.e., negativo) real.

`%phi` [Constante]

`%phi` representa o então chamado *número áureo*, $(1 + sqrt(5))/2$. O valor numérico de `%phi` é o número em ponto flutuante de de dupla precisão 1.618033988749895d0.

`fibtophi` expressa números de Fibonacci `fib(n)` em termos de `%phi`.

Por padrão, Maxima não conhece as propriedade algébricas de `%phi`. Após avaliar `tellrat(%phi^2 - %phi - 1)` e `algebraic: true`, `ratsimp` pode simplificar algumas expressãoes contendo `%phi`.

Exemplos:

`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
```

Por padrão, Maxima não conhece as propriedade algébricas de `%phi`. Após avaliar `tellrat(%phi^2 - %phi - 1)` e `algebraic: true`, `ratsimp` pode simplificar algumas expressãoes contendo `%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` [Constante]

`%pi` representa a razão do perímetro de um círculo para seu diâmetro. O valor numérico de `%pi` é o n;umero em ponto flutuante de dupla precisão 3.141592653589793d0.

`true` [Constante]

`true` representa a constante Booleana verdadeiro. Maxima implementa `true` através do valor `T` no Lisp.

14 Logaritmos

14.1 Definições para Logaritmos

`%e_to_numlog` [Variável de opção]

Valor por omissão: `false`

Quando `true`, sendo `r` algum número racional, e `x` alguma expressão, `%e^(r*log(x))` será simplificado em `x^r` . Note-se que o comando `radcan` também faz essa transformação, assim como algumas transformações mais complicadas. O comando `logcontract` *contrai* expressões contendo `log`.

`li` [*s*] (*z*) [Função]

Representa a função polilogaritmo de ordem *s* e argumento *z*, definida por meio da série infinita

$$\mathrm{Li}_s(z) = \sum_{k=1}^{\inf} \frac{z^k}{k^s}$$

`li [1]` é `- log (1 - z)`. `li [2]` e `li [3]` são as funções dilogaritmo e trilogaritmo, respectivamente.

Quando a ordem for 1, o polilogaritmo simplifica para `- log (1 - z)`, o qual por sua vez simplifica para um valor numérico se *z* for um número em ponto flutuante real ou complexo ou o sinalizador de avaliação `numer` estiver presente.

Quando a ordem for 2 ou 3, o polilogaritmo simplifica para um valor numérico se *z* for um número real em ponto flutuante ou o sinalizador de avaliação `numer` estiver presente.

Exemplos:

```
(%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*) [Função]

Representa o logaritmo natural (base *e*) de *x*.

Maxima não possui uma função interna para logaritmo de base 10 ou de outras bases. `log10(x) := log(x) / log(10)` é uma definição útil.

A simplificação e avaliação de logaritmos são governadas por vários sinalizadores globais:

`logexpand` - faz com que `log(a^b)` se transfome em `b*log(a)`. Se `logexpand` tiver o valor `all`, `log(a*b)` irá também simplificar para `log(a)+log(b)`. Se `logexpand` for igual a `super`, então `log(a/b)` irá também simplificar para `log(a)-log(b)` para números racionais `a/b`, `a#1` (`log(1/b)`, para `b` inteiro, sempre simplifica). Se `logexpand` for igaul a `false`, todas essas simplificações irão ser desabilitadas.

`logsimp` - se tiver valor `false`, não será feita nenhuma simplificação de `%e` para um expoente contendo `log`'s.

`lognumer` - se tiver valor `true`, os argumentos negativos em ponto flutuante para `log` irá sempre ser convertidos para seu valor absoluto antes que `log` seja calculado. Se `numer` for também `true`, então argumentos negativos inteiros para `log` irão também ser convertidos para os seus valores absolutos.

`lognegint` - se tiver valor `true`, implementa a regra `log(-n) -> log(n)+%i*%pi` para `n` um inteiro positivo.

`%e_to_numlog` - quando for igual a `true`, `%e^(r*log(x))`, sendo `r` algum número racional, e `x` alguma expressão, será simplificado para `x^r`. Note-se que o comando `radcan` também faz essa transformação, e outras transformações mais complicadas desse género.

O comando `logcontract` "contrai" expressões contendo `log`.

`logabs` [Variável de opção]

Valor por omissão: `false`

No cálculo de primitivas em que sejam gerados logaritmos, por exemplo, `integrate(1/x,x)`, a resposta será dada em termos de `log(abs(...))` se `logabs`

for `true`, mas em termos de `log(...)` se `logabs` for `false`. Para integrais definidos, usa-se `logabs:true`, porque nesse caso muitas vezes é necessário calcular a primitiva nos extremos.

`logarc` [Variável de opção]
`logarc` (*expr*) [Função]

Quando a variável global `logarc` for igual a `true`, as funções trigononométricas inversas, circulares e hiperbólicas, serão substituídas por suas funções logarítmicas equivalentes. O valor padrão de `logarc` é `false`.

A função `logarc(`*expr*`)` realiza essa substituição para uma expressão *expr* sem modificar o valor da variável global `logarc`.

`logconcoeffp` [Variável de opção]

Valor por omissão: `false`

Controla quais coeficientes são contraídos quando se usa `logcontract`. Poderá ser igual ao nome de uma função de um argumento. Por exemplo, se quiser gerar raízes quadradas, pode fazer `logconcoeffp:'logconfun$ logconfun(m):=featurep(m,integer) or ratnump(m)$`. E assim, `logcontract(1/2*log(x));` produzirá `log(sqrt(x))`.

`logcontract` (*expr*) [Função]

Examina recursivamente a expressão *expr*, transformando subexpressões da forma `a1*log(b1) + a2*log(b2) + c` em `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 )
```

Se fizer `declare(n,integer);` então `logcontract(2*a*n*log(x));` produzirá `a*log(x^(2*n))`. Os coeficientes que *contraem* dessa maneira são os que, tal como 2 e `n` neste exemplo, satisfazem `featurep(coeficiente,integer)`. O utilizador pode controlar quais coeficientes são contraídos, dando à variável `logconcoeffp` o nome de uma função de um argumento. Por exemplo, se quiser gerar raízes quadradas, pode fazer `logconcoeffp:'logconfun$ logconfun(m):=featurep(m,integer) or ratnump(m)$`. E assim, `logcontract(1/2*log(x));` produzirá `log(sqrt(x))`.

`logexpand` [Variável de opção]

Valor por omissão: `true`

Faz com que `log(a^b)` se transfome em `b*log(a)`. Se `logexpand` tiver o valor `all`, `log(a*b)` irá também simplificar para `log(a)+log(b)`. Se `logexpand` for igual a `super`, então `log(a/b)` irá também simplificar para `log(a)-log(b)` para números racionais `a/b`, `a#1` (`log(1/b)`, para `b` inteiro, sempre simplifica). Se `logexpand` for igaul a `false`, todas essas simplificações irão ser desabilitadas.

`lognegint` [Variável de opção]

Valor por omissão: `false`

Se for igual a `true`, implementa a regra `log(-n) -> log(n)+%i*%pi` para `n` um inteiro positivo.

`lognumer` [Variável de opção]

Valor por omissão: `false`

Se tiver valor `true`, os argumentos negativos em ponto flutuante para `log` irá sempre ser convertidos para seu valor absoluto antes que `log` seja calculado. Se `numer` for também `true`, então argumentos negativos inteiros para `log` irão também ser convertidos para os seus valores absolutos.

`logsimp` [Variável de opção]

Valor por omissão: `true`

Se tiver valor `false`, não será feita nenhuma simplificação de `%e` para um expoente contendo `log`'s.

`plog` (*x*) [Função]

Representa o ramo principal dos logaritmos naturais no plano complexo, com `-%pi < carg(x) <= +%pi`.

15 Trigonometria

15.1 Introdução ao Pacote Trigonométrico

Maxima tem muitas funções trigonométricas definidas. Não todas as identidades trigonometricas estão programadas, mas isso é possível para o utilizador adicionar muitas delas usando a compatibilidade de correspondência de modelos do sistema. As funções trigonométricas definidas no Maxima são: `acos`, `acosh`, `acot`, `acoth`, `acsc`, `acsch`, `asec`, `asech`, `asin`, `asinh`, `atan`, `atanh`, `cos`, `cosh`, `cot`, `coth`, `csc`, `csch`, `sec`, `sech`, `sin`, `sinh`, `tan`, e `tanh`. Existe uma colecção de comandos especialmente para manusear funções trigonométricas, veja `trigexpand`, `trigreduce`, e o comutador `trigsign`. Dois pacotes compartilhados extendem as regras de simplificação construídas no Maxima, `ntrig` e `atrig1`. Faça `describe(`*comando*`)` para detalhes.

15.2 Definições para Trigonometria

`acos` (*x*) [Função]
- Arco Cosseno.

`acosh` (*x*) [Função]
- Arco Cosseno Hiperbólico.

`acot` (*x*) [Função]
- Arco Cotangente.

`acoth` (*x*) [Função]
- Arco Cotangente Hiperbólico.

`acsc` (*x*) [Função]
- Arco Cossecante.

`acsch` (*x*) [Função]
- Arco Cossecante Hiperbólico.

`asec` (*x*) [Função]
- Arco Secante.

`asech` (*x*) [Função]
- Arco Secante Hiperbólico.

`asin` (*x*) [Função]
- Arco Seno.

`asinh` (*x*) [Função]
- Arco Seno Hiperbólico.

`atan` (*x*) [Função]
- Arco Tangente.

`atan2` (*y*, *x*) [Função]
- retorna o valor de `atan(`*y*`/`*x*`)` no intervalo de `-%pi` a `%pi`.

`atanh (x)` [Função]
- Arco tangente Hiperbólico.

`atrig1` [Pacote]
O pacote `atrig1` contém muitas regras adicionais de simplificação para funções trigonométricas inversas. Junto com regras já conhecidas para Maxima, os seguintes ângulos estão completamente implementados: `0`, `%pi/6`, `%pi/4`, `%pi/3`, e `%pi/2`. Os ângulos correspondentes nos outros três quadrantes estão também disponíveis. Faça `load("atrig1");` para usá-lo.

`cos (x)` [Função]
- Cosseno.

`cosh (x)` [Função]
- Cosseno hiperbólico.

`cot (x)` [Função]
- Cotangente.

`coth (x)` [Função]
- Cotangente Hyperbólica.

`csc (x)` [Função]
- Cossecante.

`csch (x)` [Função]
- Cossecante Hyperbólica.

`halfangles` [Variável de opção]
Default value: `false`

Quando `halfangles` for `true`, meios-ângulos são simplificados imediatamente.

`ntrig` [Pacote]
O pacote `ntrig` contém um conjunto de regras de simplificação que são usadas para simplificar função trigonométrica cujos argumentos estão na forma `f(n %pi/10)` onde f é qualquer das funções `sin`, `cos`, `tan`, `csc`, `sec` e `cot`.

`sec (x)` [Função]
- Secante.

`sech (x)` [Função]
- Secante Hyperbólica.

`sin (x)` [Função]
- Seno.

`sinh (x)` [Função]
- Seno Hyperbólico.

`tan (x)` [Função]
- Tangente.

`tanh` (*x*) [Função]
- Tangente Hyperbólica.

`trigexpand` (*expr*) [Função]
Expande funções trigonometricas e hyperbólicas de adições de ângulos e de ângulos multiplos que ocorram em *expr*. Para melhores resultados, *expr* deve ser expandida. Para intensificar o controle do utilizador na simplificação, essa função expande somente um nível de cada vez, expandindo adições de ângulos ou ângulos multiplos. Para obter expansão completa dentro de senos e co-senos imediatamente, escolha o comutador `trigexpand: true`.

`trigexpand` é governada pelos seguintes sinalizadores globais:

`trigexpand`
Se `true` causa expansão de todas as expressões contendo senos e co-senos ocorrendo subsequêntemente.

`halfangles`
Se `true` faz com que meios-ângulos sejam simplificados imediatamente.

`trigexpandplus`
Controla a regra "soma" para `trigexpand`, expansão de adições (e.g. `sin(x + y)`) terão lugar somente se `trigexpandplus` for `true`.

`trigexpandtimes`
Controla a regra "produto" para `trigexpand`, expansão de produtos (e.g. `sin(2 x)`) terão lugar somente se `trigexpandtimes` for `true`.

Exemplos:

```
(%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` [Variável de opção]
Valor por omissão: `true`

`trigexpandplus` controla a regra da "soma" para `trigexpand`. Dessa forma, quando o comando `trigexpand` for usado ou o comutador `trigexpand` escolhido para `true`, expansão de adições (e.g. `sin(x+y)`) terão lugar somente se `trigexpandplus` for `true`.

`trigexpandtimes` [Variável de opção]
Valor por omissão: `true`

`trigexpandtimes` controla a regra "produto" para `trigexpand`. Dessa forma, quando o comando `trigexpand` for usado ou o comutador `trigexpand` escolhido para `true`, expansão de produtos (e.g. `sin(2*x)`) terão lugar somente se `trigexpandtimes` for `true`.

`triginverses` [Variável de opção]

Valor por omissão: `all`

`triginverses` controla a simplificação de composições de funções trigonométricas e hiperbólicas com suas funções inversas.

Se `all`, ambas e.g. `atan(tan(x))` e `tan(atan(x))` simplificarão para *x*.

Se `true`, a simplificação de *`arcfun`*`(`*`fun`*`(x))` é desabilitada.

Se `false`, ambas as simplificações *`arcfun`*`(`*`fun`*`(x))` e *`fun`*`(`*`arcfun`*`(x))` são desabilitadas.

`trigreduce (`*`expr`*`, `*`x`*`)` [Função]
`trigreduce (`*`expr`*`)` [Função]

Combina produtos e expoentes de senos e cossenso trigonométricos e hiperbólicos de *x* dentro daqueles de múltiplos de *x*. Também tenta eliminar essas funções quando elas ocorrerem em denominadores. Se *x* for omitido então todas as variáveis em *expr* são usadas.

Veja também `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
```

As rotinas de simplificação trigonométrica irão usar informações declaradas em alguns casos simples. Declarações sobre variáveis são usadas como segue, 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` [Variável de opção]

Valor por omissão: `true`

Quando `trigsign` for `true`, permite simplificação de argumentos negativos para funções trigonométricas. E.g., `sin(-x)` transformar-se-á em `-sin(x)` somente se `trigsign` for `true`.

`trigsimp (`*`expr`*`)` [Função]

Utiliza as identidades $sin(x)^2 + cos(x)^2 = 1$ and $cosh(x)^2 - sinh(x)^2 = 1$ para simplificar expressões contendo `tan`, `sec`, etc., para `sin`, `cos`, `sinh`, `cosh`.

`trigreduce`, `ratsimp`, e `radcan` podem estar habilitadas a adicionar simplificações ao resultado.

`demo ("trgsmp.dem")` mostra alguns exemplos de `trigsimp`.

`trigrat (`*`expr`*`)` [Função]

Fornece uma forma quase-linear simplificada canónica de uma expressão trigonométrica; *expr* é uma fração racional de muitos `sin`, `cos` ou `tan`, os argumentos

delas são formas lineares em algumas variáveis (ou kernels-núcleos) e `%pi/n` (n inteiro) com coeficientes inteiros. O resultado é uma fração simplificada com numerador e denominador ambos lineares em `sin` e `cos`. Dessa forma `trigrat` linearia sempre quando isso for passível.

```
(%i1) trigrat(sin(3*a)/sin(a+%pi/3));
(%o1)            sqrt(3) sin(2 a) + cos(2 a) - 1
```

O seguinte exemplo encontra-se em Davenport, Siret, and Tournier, *Calcul Formel*, Masson (ou em inglês, Addison-Wesley), secção 1.5.5, teorema de Morley.

```
(%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 Funções Especiais

16.1 Introdução a Funções Especiais

A notação de função especial segue adiante:

`bessel_j (index, expr)`	Função de Bessel, primeiro tipo
`bessel_y (index, expr)`	Função de Bessel, segundo tipo
`bessel_i (index, expr)`	Função de Bessel modificada, primeiro tipo
`bessel_k (index, expr)`	Função de Bessel modificada, segundo tipo
`%he[n] (z)`	Polinómio de Hermite (Note bem: he, não h. Veja A&S 22.
`%p[u,v] (z)`	Função de Legendre
`%q[u,v] (z)`	Função de Legendre, segundo tipo
`hstruve[n] (z)`	Função H de Struve H
`lstruve[n] (z)`	Função de L Struve
`%f[p,q] ([], [], expr)`	Função Hipergeométrica Generalizada
`gamma()`	Função Gamma
`gamma_incomplete_lower(a,z)`	Função gama incompleta inferior
`gammaincomplete(a,z)`	Final da função gama incompleta
`slommel`	
`%m[u,k] (z)`	Função de Whittaker, primeiro tipo
`%w[u,k] (z)`	Função de Whittaker, segundo tipo
`erfc (z)`	Complemento da função erf (função de er-
`ros - integral da distribuição`	normal)
`ei (z)`	Integral de exponencial (?)
`kelliptic (z)`	integral eliptica completa de primeiro tipo (K)
`%d [n] (z)`	Função cilíndrica parabólica

16.2 Definições para Funções Especiais

`airy_ai (x)` [Função]

A função de Airy Ai, como definida em Abramowitz e Stegun, *Handbook of Mathematical Functions*, Sessão 10.4.

A equação de Airy `diff (y(x), x, 2) - x y(x) = 0` tem duas soluções linearmente independentes, `y = Ai(x)` e `y = Bi(x)`. A derivada de `diff (airy_ai(x), x)` é `airy_dai(x)`.

Se o argumento `x` for um número real ou um número complexo qualquer deles em ponto flutuante , o valor numérico de `airy_ai` é retornado quando possível.

Veja também `airy_bi`, `airy_dai`, `airy_dbi`.

`airy_dai (x)` [Função]

A derivada da função de Airy Ai `airy_ai(x)`.

Veja `airy_ai`.

`airy_bi (x)` [Função]

A função de Airy Bi, como definida em Abramowitz e Stegun, *Handbook of Mathematical Functions*, Sessão 10.4, é a segunda solução da equação de Airy `diff (y(x), x, 2) - x y(x) = 0`.

Se o argumento `x` for um número real ou um número complexo qualquer deles em ponto flutuante, o valor numérico de `airy_bi` é retornado quando possível. Em outros casos a expressão não avaliada é retornada.

A derivada de `diff (airy_bi(x), x)` é `airy_dbi(x)`.

Veja `airy_ai`, `airy_dbi`.

`airy_dbi (x)` [Função]

A derivada de função de Airy Bi `airy_bi(x)`.

Veja `airy_ai` e `airy_bi`.

`asympa` [Função]

`asympa` é um pacote para análise assintótica. O pacote contém funções de simplificação para análise assintótica, incluindo as funções "grande O" e "pequeno o" que são largamente usadas em análises de complexidade e análise numérica.

`load ("asympa")` chama esse pacote.

`bessel (z, a)` [Função]

A função de Bessel de primeiro tipo.

Essa função está desactualizada. Escreva `bessel_j (z, a)` em lugar dessa.

`bessel_j (v, z)` [Função]

A função de Bessel do primeiro tipo de ordem v e argumento z.

`bessel_j` calcula o array `besselarray` tal que `besselarray [i] = bessel_j [i + v - int(v)] (z)` para `i` de zero a `int(v)`.

`bessel_j` é definida como

$$\sum_{k=0}^{\infty} \frac{(-1)^k \ \left(\frac{z}{2}\right)^{v+2\,k}}{k!\,\Gamma\left(v+k+1\right)}$$

todavia séries infinitas não são usadas nos cálculos.

`bessel_y (v, z)` [Função]

A função de Bessel do segundo tipo de ordem v e argumento z.

`bessel_y` calcula o array `besselarray` tal que `besselarray [i] = bessel_y [i + v - int(v)] (z)` para `i` de zero a `int(v)`.

`bessel_y` é definida como

$$\frac{\cos\left(\pi\,v\right)\ J_v(z) - J_{-v}(z)}{\sin\left(\pi\,v\right)}$$

quando v não for um inteiro. Quando v for um inteiro n, o limite com v aprocimando-se de n é tomado.

`bessel_i (v, z)` [Função]

A função de Bessel modificada de primeiro tipo de ordem v e argumento z.

`bessel_i` calcula o array `besselarray` tal que `besselarray [i] = bessel_i [i + v - int(v)] (z)` para `i` de zero a `int(v)`.

`bessel_i` é definida como

$$\sum_{k=0}^{\infty} \frac{1}{k!\,\Gamma\left(v+k+1\right)} \left(\frac{z}{2}\right)^{v+2\,k}$$

todavia séries infinitas não são usadas nos cálculos.

`bessel_k (v, z)` [Função]

A função de Bessel modificada de segundo tipo de ordem v e argumento z.

`bessel_k` calcula o array `besselarray` tal que `besselarray [i] = bessel_k [i + v - int(v)] (z)` para `i` de zero a `int(v)`.

`bessel_k` é definida como

$$\frac{\pi\,\csc\left(\pi\,v\right)\,\left(I_{-v}(z)-I_{v}(z)\right)}{2}$$

quando v não for inteiro. Se v for um inteiro n, então o limite com v aproximando-se de n é tomado.

`besselexpand` [Variável de opção]

Valor por omissão: `false`

Expansões de controle de funções de Bessel quando a ordem for a metade de um inteiro ímpar. Nesse caso, as funções de Bessel podem ser expandidas em termos de outras funções elementares. Quando `besselexpand` for `true`, a função de Bessel é expandida.

```
(%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)` [Função]

A função homotética modificada de Bessel de primeiro tipo de ordem v e argumento z. Isto é, $scaled_b essel_i(v,z) = exp(-abs(z)) * bessel_i(v,z)$. Essa função é particularmente útil para calcular $bessel_i$ para grandes valores de z. Todavia, maxima não conhece outra forma muito mais sobre essa função. Para computação simbólica, é provavelmete preferível trabalhar com a expressão `exp(-abs(z))*bessel_i(v, z)`.

`scaled_bessel_i0 (z)` [Função]

Idêntica a `scaled_bessel_i(0,z)`.

`scaled_bessel_i1 (z)` [Função]

Idêntica a `scaled_bessel_i(1,z)`.

`beta` (*x*, *y*) [Função]
A função beta, definida como `gamma(x) gamma(y)/gamma(x + y)`.

`gamma` (*x*) [Função]
A função gama.

Veja também `makegamma`.

A variável `gammalim` controla a simplificação da função gama.

A constante de Euler-Mascheroni é `%gamma`.

`gammalim` [Variável de opção]
Valor por omissão: 1000000

`gammalim` controla a simplificação da função gama para integral e argumentos na forma de números racionais. Se o valor absoluto do argumento não for maior que `gammalim`, então a simplificação ocorrerá. Note que `factlim` comuta controle de simplificaçcão do resultado de `gamma` de um argumento inteiro também.

`intopois` (*a*) [Função]
Converte *a* em um código de Poisson.

`makefact` (*expr*) [Função]
Transforma instâncias de funções binomiais, gama, e beta em *expr* para factoriais.

Veja também `makegamma`.

`makegamma` (*expr*) [Função]
Transforma instâncias de funções binomiais, factorial, e beta em *expr* para funções gama.

Veja também `makefact`.

`numfactor` (*expr*) [Função]
Retorna o factor numérico multiplicando a expressão *expr*, que pode ser um termo simples.

`content` retorna o máximo divisor comum (mdc) de todos os termos em uma adição.

```
(%i1) gamma (7/2);
                              15 sqrt(%pi)
(%o1)                         ------------
                                   8
(%i2) numfactor (%);
                                   15
(%o2)                              --
                                   8
```

`outofpois` (*a*) [Função]
Converte *a* de um código de Poisson para uma representação geral. Se *a* não for uma forma de Poisson, `outofpois` realiza a conversão, i.e., o valor de retorno é `outofpois (intopois (a))`. Essa função é desse modo um simplificador canónico para adições e potências de termos de seno e co-seno de um tipo particular.

`poisdiff (a, b)` [Função]
Deriva *a* com relação a *b*. *b* deve ocorrer somente nos argumentos trigonométricos ou somente nos coeficientes.

`poisexpt (a, b)` [Função]
Funcionalmente identica a `intopois (a^b)`. *b* deve ser um inteiro positico.

`poisint (a, b)` [Função]
Integra em um senso restrito similarmente (para `poisdiff`). Termos não periódicos em *b* são diminuídos se *b* estiver em argumentos trigonométricos.

`poislim` [Variável de opção]
Valor por omissão: 5

`poislim` determina o domínio dos coeficientes nos argumentos de funções trigonométricas. O valor inicial de 5 corresponde ao intervalo [-2^(5-1)+1,2^(5-1)], ou [-15,16], mas isso pode ser alterado para [-2^(n-1)+1, 2^(n-1)].

`poismap (series, sinfn, cosfn)` [Função]
mapeará as funções *sinfn* sobre os termos de seno e *cosfn* ssobre os termos de co-seno das séries de Poisson dadas. *sinfn* e *cosfn* são funções de dois argumentos que são um coeficiente e uma parte trigonométrica de um termo em séries respectivamente.

`poisplus (a, b)` [Função]
É funcionalmente identica a `intopois (a + b)`.

`poissimp (a)` [Função]
Converte *a* em séries de Poisson para *a* em representação geral.

`poisson` [Símbolo especial]
O símbolo `/P/` segue o rótulo de linha de uma expressão contendo séries de Poisson.

`poissubst (a, b, c)` [Função]
Substitue *a* por *b* em *c*. *c* é uma série de Poisson.

(1) Quando *B* é uma variável *u*, *v*, *w*, *x*, *y*, ou *z*, então *a* deve ser uma expressão linear nessas variáveis (e.g., `6*u + 4*v`).

(2) Quando *b* for outra que não essas variáveis, então *a* deve também ser livre dessas variáveis, e além disso, livre de senos ou co-senos.

`poissubst (a, b, c, d, n)` é um tipo especial d substituição que opera sobre *a* e *b* como no tipo (1) acima, mas onde *d* é uma série de Poisson, expande `cos(d)` e `sin(d)` para a ordem *n* como provendo o resultado da substituição `a + d` por *b* em *c*. A idéia é que *d* é uma expansão em termos de um pequeno parâmetro. Por exemplo, `poissubst (u, v, cos(v), %e, 3)` retorna `cos(u)*(1 - %e^2/2) - sin(u)*(%e - %e^3/6)`.

`poistimes (a, b)` [Função]
É funcionalmente idêntica a `intopois (a*b)`.

`poistrim ()` [Função]
é um nome de função reservado que (se o utilizador tiver definido uma função com esse nome) é aplicada durante multiplicação de Poisson. Isso é uma função predicada

de 6 argumentos que são os coeficientes de *u*, *v*, ..., *z* em um termo. Termos para os quais `poistrim` for `true` (para os coeficientes daquele termo) são eliminados durante a multiplicação.

`printpois (a)` [Função]

Mostra uma série de Poisson em um formato legível. Em comum com `outofpois`, essa função converterá *a* em um código de Poisson primeiro, se necessário.

`psi [n](x)` [Função]

A derivada de `log (gamma (x))` de ordem `n+1`. Dessa forma, `psi[0](x)` é a primeira derivada, `psi[1](x)` é a segunda derivada, etc.

Maxima não sabe como, em geral, calcular um valor numérico de `psi`, mas Maxima pode calcular alguns valores exatos para argumentos racionais. Muitas variáveis controlam qual intervalo de argumentos racionais `psi` irá retornar um valor exato, se possível. Veja `maxpsiposint`, `maxpsinegint`, `maxpsifracnum`, e `maxpsifracdenom`. Isto é, *x* deve localizar-se entre `maxpsinegint` e `maxpsiposint`. Se o valor absoluto da parte facionária de *x* for racional e tiver um numerador menor que `maxpsifracnum` e tiver um denominador menor que `maxpsifracdenom`, `psi` irá retornar um valor exato.

A função `bfpsi` no pacote `bffac` pode calcular valores numéricos.

`maxpsiposint` [Variável de opção]

Valor por omissão: 20

`maxpsiposint` é o maior valor positivo para o qual `psi[n](x)` irá tentar calcular um valor exato.

`maxpsinegint` [Variável de opção]

Valor por omissão: -10

`maxpsinegint` é o valor mais negativo para o qual `psi[n](x)` irá tentar calcular um valor exato. Isto é, se *x* for menor que `maxnegint`, `psi[n](x)` não irá retornar resposta simplificada, mesmo se isso for possível.

`maxpsifracnum` [Variável de opção]

Valor por omissão: 4

Tomemos *x* como sendo um número racional menor que a unidade e da forma `p/q`. Se `p` for menor que `maxpsifracnum`, então `psi[n](x)` não irá tentar retornar um valor simplificado.

`specint` (*exp*(- *s***t*) * *expr*, *t*) [Função]

Calcula a transformada de Laplace de *expr* com ralação à variável *t*. O integrando *expr* pode conter funções especiais.

Se `specint` não puder calcular a integral, o valore de retorno pode conter vários símbolos do Lisp, incluindo `other-defint-to-follow-negtest`, `other-lt-exponential-to-follow`, `product-of-y-with-nofract-indices`, etc.; isso é um erro.

`demo(hypgeo)` mostra muitos exemplos de transformadas de Laplace calculados por `specint`.

Exemplos:

```
(%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
```

`maxpsifracdenom` [Variável de opção]

Valor por omissão: 4

Tomemos x como sendo um número racional menor que a unidade e da forma `p/q`. Se `q` for maior que `maxpsifracdeonm`, então `psi[n](x)` não irá tentar retornar um valor simplificado.

17 Funções Elípticas

17.1 Introdução a Funções Elípticas e Integrais

Maxima inclui suporte a funções elípticas Jacobianas e a integrais elípticas completas e incompletas. Isso inclui manipulação simbólica dessas funções e avaliação numérica também. Definições dessas funções e muitas de suas propriedades podem ser encontradas em Abramowitz e Stegun, Capítulos 16–17. Tanto quanto possível, usamos as definições e relações dadas aí.

Em particular, todas as funções elípticas e integrais elípticas usam o parâmetro m em lugar de módulo k ou o ângulo modular α. Isso é uma área onde discordamos de Abramowitz e Stegun que usam o ângulo modular para as funções elípticas. As seguintes relações são verdadeiras:

$$m = k^2$$

and

$$k = \sin\alpha$$

As funções elípticas e integrais elípticas estão primariamente tencionando suportar computação simbólica. Portanto, a maiora das derivadas de funções e integrais são conhecidas. Todavia, se valores em ponto flutuante forem dados, um resultado em ponto flutuante é retornado.

Suporte para a maioria de outras propriedades das funções elípticas e integrais elípticas além das derivadas não foram ainda escritas.

Alguns exemplos de funções elípticas:

```
(%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)
```

Alguns exemplos de integrais elípticas:

```
(%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))
```

Suporte a funções elípticas e integrais elípticas foi escrito por Raymond Toy. Foi colocado sob os termos da Licençã Pública Geral (GPL) que governa a distribuição do Maxima.

17.2 Definições para Funções Elípticas

`jacobi_sn` (*u*, *m*) [Função]
A Função elíptica Jacobiana $sn(u, m)$.

`jacobi_cn` (*u*, *m*) [Função]
A função elíptica Jacobiana $cn(u, m)$.

`jacobi_dn` (*u*, *m*) [Função]
A função elíptica Jacobiana $dn(u, m)$.

`jacobi_ns (u, m)` [Função]
A função elíptica Jacobiana $ns(u, m) = 1/sn(u, m)$.

`jacobi_sc (u, m)` [Função]
A função elíptica Jacobiana $sc(u, m) = sn(u, m)/cn(u, m)$.

`jacobi_sd (u, m)` [Função]
A função elíptica Jacobiana $sd(u, m) = sn(u, m)/dn(u, m)$.

`jacobi_nc (u, m)` [Função]
A função elíptica Jacobiana $nc(u, m) = 1/cn(u, m)$.

`jacobi_cs (u, m)` [Função]
A função elíptica Jacobiana $cs(u, m) = cn(u, m)/sn(u, m)$.

`jacobi_cd (u, m)` [Função]
A função elíptica Jacobiana $cd(u, m) = cn(u, m)/dn(u, m)$.

`jacobi_nd (u, m)` [Função]
A função elíptica Jacobiana $nc(u, m) = 1/cn(u, m)$.

`jacobi_ds (u, m)` [Função]
A função elíptica Jacobiana $ds(u, m) = dn(u, m)/sn(u, m)$.

`jacobi_dc (u, m)` [Função]
A função elíptica Jacobiana $dc(u, m) = dn(u, m)/cn(u, m)$.

`inverse_jacobi_sn (u, m)` [Função]
A inversa da função elíptica Jacobiana $sn(u, m)$.

`inverse_jacobi_cn (u, m)` [Função]
A inversa da função elíptica Jacobiana $cn(u, m)$.

`inverse_jacobi_dn (u, m)` [Função]
A inversa da função elíptica Jacobiana $dn(u, m)$.

`inverse_jacobi_ns (u, m)` [Função]
A inversa da função elíptica Jacobiana $ns(u, m)$.

`inverse_jacobi_sc (u, m)` [Função]
A inversa da função elíptica Jacobiana $sc(u, m)$.

`inverse_jacobi_sd (u, m)` [Função]
A inversa da função elíptica Jacobiana $sd(u, m)$.

`inverse_jacobi_nc (u, m)` [Função]
A inversa da função elíptica Jacobiana $nc(u, m)$.

`inverse_jacobi_cs (u, m)` [Função]
A inversa da função elíptica Jacobiana $cs(u, m)$.

`inverse_jacobi_cd (u, m)` [Função]
A inversa da função elíptica Jacobiana $cd(u, m)$.

`inverse_jacobi_nd` (*u*, *m*) [Função]
A inversa da função elíptica Jacobiana $nc(u, m)$.

`inverse_jacobi_ds` (*u*, *m*) [Função]
A inversa da função elíptica Jacobiana $ds(u, m)$.

`inverse_jacobi_dc` (*u*, *m*) [Função]
A inversa da função elíptica Jacobiana $dc(u, m)$.

17.3 Definições para Integrais Elípticas

`elliptic_f` (*phi*, *m*) [Função]
A integral elíptica incompleta de primeiro tipo, definida como

$$\int_0^\phi \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}$$

Veja também *elliptic_e* e *elliptic_kc*.

`elliptic_e` (*phi*, *m*) [Função]
A integral elíptica incompleta de segundo tipo, definida como

$$\int_0^\phi \sqrt{1 - m\sin^2\theta}d\theta$$

Veja também *elliptic_e* e *elliptic_ec*.

`elliptic_eu` (*u*, *m*) [Função]
A integral elíptica incompleta de segundo tipo, definida como

$$\int_0^u \mathrm{dn}(v, m)dv = \int_0^\tau \sqrt{\frac{1 - mt^2}{1 - t^2}}dt$$

onde $\tau = \mathrm{sn}(u, m)$
Isso é relacionado a $elliptic_e$ através de

$$E(u, m) = E(\phi, m)$$

onde $\phi = \sin^{-1}\mathrm{sn}(u, m)$ Veja também *elliptic_e*.

`elliptic_pi` (*n*, *phi*, *m*) [Função]
A integral elíptica incompleta de terceiro tipo, definida como

$$\int_0^\phi \frac{d\theta}{(1 - n\sin^2\theta)\sqrt{1 - m\sin^2\theta}}$$

Somente a derivada em relação a *phi* é conhecida pelo Maxima.

`elliptic_kc` (*m*) [Função]

A integral elíptica completa de primeiro tipo, definida como

$$\int_0^{\frac{\pi}{2}} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}$$

Para certos valores de m, o valor da integral é conhecido em termos de funções *Gama*. Use `makegamma` para avaliar esse valor.

`elliptic_ec` (*m*) [Função]

A integral elíptica completa de segundo tipo, definida como

$$\int_0^{\frac{\pi}{2}} \sqrt{1 - m\sin^2\theta} d\theta$$

Para certos valores de m, o valor da integral é conhecido em termos de funções *Gama*. Use `makegamma` para avaliar esse valor.

18 Limites

18.1 Definições para Limites

`lhospitallim` [Variável de Opção]

Valor por omissão: 4

`lhospitallim` é o máximo número de vezes que a regra L'Hospital é usada em `limit`. Isso evita ciclos infinitos em casos como `limit (cot(x)/csc(x), x, 0)`.

`limit` (*expr*, *x*, *val*, *dir*) [Função]
`limit` (*expr*, *x*, *val*) [Função]
`limit` (*expr*) [Função]

Calcula o limite de *expr* com a variável real *x* aproximando-se do valor *val* pela direção *dir*. *dir* pode ter o valor `plus` para um limite pela direita, `minus` para um limite pela esquerda, ou pode ser omitido (implicando em um limite em ambos os lados é para ser computado).

`limit` usa os seguintes símbolos especiais: `inf` (infinito positivo) e `minf` (infinito negativo). Em saídas essa função pode também usar `und` (undefined - não definido), `ind` (indefinido mas associado) e `infinity` (infinito complexo).

`lhospitallim` é o máximo número de vezes que a regra L'Hospital é usada em `limit`. Isso evita ciclos infinitos em casos como `limit (cot(x)/csc(x), x, 0)`.

`tlimswitch` quando `true` fará o pacote `limit` usar série de Taylor quando possível.

`limsubst` evita que `limit` tente substituições sobre formas desconhecidas. Isso é para evitar erros como `limit (f(n)/f(n+1), n, inf)` dando igual a 1. Escolhendo `limsubst` para `true` permitirá tais substituições.

`limit` com um argumento é muitas vezes chamado em ocasiões para simplificar expressões de constantes, por exemplo, `limit (inf-1)`.

`example (limit)` mostra alguns exemplos.

Para saber sobre o método utilizado veja Wang, P., "Evaluation of Definite Integrals by Symbolic Manipulation", tese de Ph.D., MAC TR-92, Outubro de 1971.

`limsubst` [Variável de Opção]

valor padrão: `false` - evita que `limit` tente substituições sobre formas desconhecidas. Isso é para evitar erros como `limit (f(n)/f(n+1), n, inf)` dando igual a 1. Escolhendo `limsubst` para `true` permitirá tais substituições.

`tlimit` (*expr*, *x*, *val*, *dir*) [Função]
`tlimit` (*expr*, *x*, *val*) [Função]
`tlimit` (*expr*) [Função]

Retorna `limit` com `tlimswitch` escolhido para `true`.

`tlimswitch` [Variável de Opção]

Valor por omissão: `false`

Quando `tlimswitch` for `true`, fará o pacote `limit` usar série de Taylor quando possível.

19 Diferenciação

19.1 Definições para Diferenciação

`antid` (*expr*, *x*, *u*(*x*)) [Função]

Retorna uma lista de dois elementos, tais que uma antiderivada de *expr* com relação a *x* pode ser constuída a partir da lista. A expressão *expr* pode conter uma função desconhecida *u* e suas derivadas.

Tome *L*, uma lista de dois elementos, como sendo o valor de retorno de `antid`. Então `L[1] + 'integrate (L[2], x)` é uma antiderivada de *expr* com relação a *x*.

Quando `antid` obtém sucesso inteiramente, o segundo elemento do valor de retorno é zero. De outra forma, o segundo elemento é não zero, e o primeiro elemento não zero ou zero. Se `antid` não pode fazer nenhum progresso, o primeiro elemento é zero e o segundo não zero.

`load ("antid")` chama essa função. O pacote `antid` também define as funções `nonzeroandfreeof` e `linear`.

`antid` está relacionada a `antidiff` como segue. Tome *L*, uma lista de dois elementos, que é o valor de retorno de `antid`. Então o valor de retorno de `antidiff` é igual a `L[1] + 'integrate (L[2], x)` onde *x* é a variável de integração.

Exemplos:

```
(%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*)) [Função]

Retorna uma antiderivada de *expr* com relação a *x*. A expressão *expr* pode conter uma função desconhecida *u* e suas derivadas.

Quando `antidiff` obtém sucesso inteiramente, a expressão resultante é livre do sinal de integral (isto é, livre do substantivo `integrate`). De outra forma, `antidiff` retorna uma expressão que é parcialmente ou inteiramente dentro de um sinal de um sinal de integral. Se `antidiff` não pode fazer qualquer progresso, o valor de retorno é inteiramente dentro de um sinal de integral.

`load ("antid")` chama essa função. O pacote `antid` também define as funções `nonzeroandfreeof` e `linear`.

`antidiff` é relacionada a `antid` como segue. Tome L, uma lista de dois elementos, como sendo o valor de retorno de `antid`. Então o valor de retorno de `antidiff` é igual a `L[1] + 'integrate (L[2], x)` onde x é a variável de integração.

Exemplos:

```
(%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` [propriedade]

`atomgrad` é a propriedade do gradiente atômico de uma expressão. Essa propriedade é atribuída por `gradef`.

`atvalue` (*expr*, [*x_1* = *a_1*, ..., *x_m* = *a_m*], *c*) [Função]

`atvalue` (*expr*, *x_1* = *a_1*, *c*) [Função]

Atribui o valor *c* a *expr* no ponto `x = a`. Tipicamente valores de extremidade são estabelecidos por esse mecanismo.

expr é a função de avaliação, `f(x_1, ..., x_m)`, ou uma derivada, `diff (f(x_1, ..., x_m), x_1, n_1, ..., x_n, n_m)` na qual os argumentos da função explicitamente aparecem. *n_i* é a ordem de diferenciação com relação a *x_i*.

O ponto no qual o `atvalue` é estabelecido é dado pela lista de equações `[x_1 = a_1, ..., x_m = a_m]`. Se existe uma variável simples *x_1*, uma única equação pode ser dada sem ser contida em uma lista.

`printprops ([f_1, f_2, ...], atvalue)` mostra os `atvalues` das funções `f_1`, `f_2`, `...` como especificado por chamadas a `atvalue`. `printprops (f, atvalue)` mostra os `atvalues` de uma função *f*. `printprops (all, atvalue)` mostra os `atvalue`s de todas as funções para as quais `atvalue`s são definidos.

Os simbolos `@1`, `@2`, ... representam as variáveis *x_1*, *x_2*, ... quando `atvalue`s são mostrados.

`atvalue` avalia seus argumentos. `atvalue` retorna *c*, o `atvalue`.

Exemplos:

```
(%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           !
```

```
                                  dx          !
                                              !x = 0, y = 1
```

`cartan` - [Função]

O cálculo exterior de formas diferenciais é uma ferramenta básica de geometria diferencial desenvolvida por Elie Cartan e tem importantes aplicações na teoria das equações diferenciais parciais. O pacote `cartan` implementa as funções `ext_diff` e `lie_diff`, juntamente com os operadores `~` (produto da cunha) e `|` (contração de uma forma com um vector.) Digite `demo (tensor)` para ver uma breve descrição desses comandos juntamente com exemplos.

`cartan` foi implementado por F.B. Estabrook e H.D. Wahlquist.

`del` (*x*) [Função]

`del (x)` representa a diferencial da variável x.

`diff` retorna uma expressão contendo `del` se uma variável independente não for especificada. Nesse caso, o valor de retorno é a então chamada "diferencial total".

Exemplos:

```
(%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*) [Função]

A função Delta de Dirac.

Correntemente somente `laplace` sabe sobre a função `delta`.

Exemplo:

```
(%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` [Variável]

Valor por omissão: `[]`

`dependencies` é a lista de átomos que possuem dependências funcionais, atribuídas por `depends` ou `gradef`. A lista `dependencies` é cumulativa: cada chamada a `depends` ou a `gradef` anexa ítens adicionais.

Veja `depends` e `gradef`.

`depends (f_1, x_1, ..., f_n, x_n)` [Função]

Declara dependêcias funcionais entre variáveis para o propósito de calcular derivadas. Na ausência de dependêcias declaradas, `diff (f, x)` retorna zero. Se `depends (f, x)` for declarada, `diff (f, x)` retorna uma derivada simbólica (isto é, um substantivo `diff`).

Cada argumento *f_1*, *x_1*, etc., pode ser o nome de uma variável ou array, ou uma lista de nomes. Todo elemento de *f_i* (talvez apenas um elemento simples) é declarado para depender de todo elemento de *x_i* (talvez apenas um elemento simples). Se algum *f_i* for o nome de um array ou contém o nome de um array, todos os elementos do array dependem de *x_i*.

`diff` reconhece dependências indirectas estabelecidas por `depends` e aplica a regra da cadeia nesses casos.

`remove (f, dependency)` remove todas as dependências declaradas para *f*.

`depends` retorna uma lista de dependências estabelecidas. As dependências são anexadas à variável global `dependencies`. `depends` avalia seus argumentos.

`diff` é o único comando Maxima que reconhece dependências estabelecidas por `depends`. Outras funções (`integrate`, `laplace`, etc.) somente reconhecem dependências explicitamente representadas por seus argumentos. Por exemplo, `integrate` não reconhece a dependência de `f` sobre `x` a menos que explicitamente representada como `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` [Variável de opção]

Valor por omissão: `false`

Quando `derivabbrev` for `true`, derivadas simbólicas (isto é, substantivos `diff`) são mostradas como subscritos. De outra forma, derivadas são mostradas na notação de Leibniz `dy/dx`.

`derivdegree` (*expr*, *y*, *x*) [Função]

Retorna o maior grau de uma derivada da variável dependente *y* com relação à variável independente *x* ocorrendo em *expr*.

Exemplo:

```
(%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*) [Função]

Causa somente diferenciações com relação às variáveis indicadas, dentro do comando `ev`.

`derivsubst` [Variável de opção]

Valor por omissão: `false`

Quando `derivsubst` for `true`, uma substiruíção não sintática tais como `subst (x, 'diff (y, t), 'diff (y, t, 2))` retorna `'diff (x, t)`.

`diff` (*expr*, *x_1*, *n_1*, ..., *x_m*, *n_m*) [Função]
`diff` (*expr*, *x*, *n*) [Função]
`diff` (*expr*, *x*) [Função]
`diff` (*expr*) [Função]

Retorna uma derivada ou diferencial de *expr* com relação a alguma ou todas as variáveis em *expr*.

`diff (expr, x, n)` retorna a *n*'ésima derivada de *expr* com relação a *x*.

`diff (expr, x_1, n_1, ..., x_m, n_m)` retorna a derivada parcial mista de *expr* com relação a *x_1*, ..., *x_m*. Isso é equivalente a `diff (... (diff (expr, x_m, n_m) ...), x_1, n_1)`.

`diff (expr, x)` retorna a primeira derivada de *expr* com relação a uma variável *x*.

`diff (expr)` retorna a diferencial total de *expr*, isto é, a soma das derivadas de *expr* com relação a cada uma de suas variáveis vezes a diferencial `del` de cada variável. Nenhuma simplificação adicional de `del` é oferecida.

A forma substantiva de `diff` é requerida em alguns contextos, tal como declarando uma equação diferencial. Nesses casos, `diff` pode ser colocado apóstrofo (com `'diff`) para retornar a forma substantiva em lugar da realização da diferenciação.

Quando `derivabbrev` for `true`, derivadas são mostradas como subscritos. De outra forma, derivadas são mostradas na notação de Leibniz, `dy/dx`.

Exemplos:

```
(%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)
```

Para o pacote tensor, as seguintes modificações foram incorporadas:

(1) As derivadas de quaisquer objectos indexados em *expr* terão as variáveis *x_i* anexadas como argumentos adicionais. Então todos os índices de derivada serão ordenados.

(2) As variáveis *x_i* podem ser inteiros de 1 até o valor de uma variável `dimension` [valor padrão: 4]. Isso fará com que a diferenciação seja concluída com relação aos *x_i*'ésimos membros da lista `coordinates` que pode ser escolhida para uma lista de nomes de coordenadas, e.g., `[x, y, z, t]`. Se `coordinates` for associada a uma variável atômica, então aquela variável subscrita por *x_i* será usada para uma variável de diferenciação. Isso permite um array de nomes de coordenadas ou nomes subscritos como `X[1]`, `X[2]`, ... sejam usados. Se `coordinates` não foram atribuídas um valor, então as variáveis seram tratadas como em (1) acima.

`diff` [Símbolo especial]

Quando `diff` está presente como um `evflag` em chamadas para `ev`, Todas as diferenciações indicadas em `expr` são realizdas.

`dscalar (`*f*`)` [Função]

Aplica o d'Alembertiano escalar para a função escalar *f*.

`load ("ctensor")` chama essa função.

`express` (*expr*) [Função]

Expande o substantivo do operador diferencial em expressões em termos de derivadas parciais. `express` reconhece os operadores `grad`, `div`, `curl`, `laplacian`. `express` também expande o produto do X `~`.

Derivadas simbólicas (isto é, substantivos `diff`) no valor de retorno de `express` podem ser avaliadas incluíndo `diff` na chamada à função `ev` ou na linha de comando. Nesse contexto, `diff` age como uma `evfun`.

`load ("vect")` chama essa função.

Exemplos:

```
(%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*) [Função]
`gradef` (`a`, *x*, *expr*) [Função]

Define as derivadas parciais (i.e., os componentes do gradiente) da função *f* ou variável *a*.

`gradef (f(x_1, ..., x_n), g_1, ..., g_m)` define `df/dx_i` como *g_i*, onde *g_i* é uma expressão; *g_i* pode ser uma chamada de função, mas não o nome de uma função. O número de derivadas parciais *m* pode ser menor que o número de argumentos *n*, nesses casos derivadas são definidas com relação a *x_1* até *x_m* somente.

`gradef (a, x, expr)` define uma derivada de variável *a* com relação a *x* como *expr*. Isso também estabelece a dependência de *a* sobre *x* (via `depends (a, x)`).

O primeiro argumento `f(x_1, ..., x_n)` ou *a* é acompanhado de apóstrofo, mas os argumentos restantes *g_1*, ..., *g_m* são avaliados. `gradef` retorna a função ou variável para as quais as derivadas parciais são definidas.

`gradef` pode redefinir as derivadas de funções internas do Maxima. Por exemplo, `gradef (sin(x), sqrt (1 - sin(x)^2))` redefine uma derivada de `sin`.

`gradef` não pode definir derivadas parciais para um função subscrita.

`printprops ([f_1, ..., f_n], gradef)` mostra as derivadas parciais das funções *f_1*, ..., *f_n*, como definidas por `gradef`.

`printprops ([a_n, ..., a_n], atomgrad)` mostra as derivadas parciais das variáveis *a_n*, ..., *a_n*, como definidas por `gradef`.

`gradefs` é a lista de funções para as quais derivadas parciais foram definidas por `gradef`. `gradefs` não inclui quaisquer variáveis para quais derivadas parciais foram definidas por `gradef`.

Gradientes são necessários quando, por exemplo, uma função não é conhecida explicitamente mas suas derivadas primeiras são e isso é desejado para obter derivadas de ordem superior.

`gradefs` [Variável de sistema]

Valor por omissão: `[]`

`gradefs` é a lista de funções para as quais derivadas parciais foram definidas por `gradef`. `gradefs` não inclui quaisquer variáveis para as quais derivadas parciais foram deinidas por `gradef`.

`laplace` (*expr*, *t*, *s*) [Função]

Tenta calcular a transformada de Laplace de *expr* com relação a uma variável *t* e parâmetro de transformação *s*. Se `laplace` não pode achar uma solução, um substantivo `'laplace` é retornado.

`laplace` reconhece em *expr* as funções `delta`, `exp`, `log`, `sin`, `cos`, `sinh`, `cosh`, e `erf`, também `derivative`, `integrate`, `sum`, e `ilt`. Se algumas outras funções estiverem presente, `laplace` pode não ser habilitada a calcular a tranformada.

expr pode também ser uma equação linear, diferencial de coeficiente contante no qual caso o `atvalue` da variável dependente é usado. O requerido `atvalue` pode ser fornecido ou antes ou depois da transformada ser calculada. Uma vez que as condições iniciais devem ser especificadas em zero, se um teve condições de limite impostas em qualquer outro lugar ele pode impor essas sobre a solução geral e eliminar as constantes resolvendo a solução geral para essas e substituindo seus valores de volta.

`laplace` reconhece integrais de convolução da forma `integrate (f(x) * g(t - x), x, 0, t)`; outros tipos de convoluções não são reconhecidos.

Relações funcionais devem ser explicitamente representadas em *expr*; relações implícitas, estabelecidas por `depends`, não são reconhecidas. Isto é, se f depende de x e y, `f (x, y)` deve aparecer em *expr*.

Veja também `ilt`, a transformada inversa de Laplace.

Exemplos:

```
(%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 Integração

20.1 Introdução a Integração

Maxima tem muitas rotinas para realizar integração. A função `integrate` faz uso de muitas dessas. Exite também o pacote `antid`, que manuseia uma função não especificada (e suas derivadas, certamente). Para usos numericos, existe um conjunto de integradores adaptativos de QUADPACK, a saber `quad_qag`, `quad_qags`, etc., os quais são descritos sob o tópico `QUADPACK`. Funções hipergeométricas estão sendo trabalhadas, veja `specint` para detalhes. Geralmente falando, Maxima somente calcula integrais que sejam integráveis em termos de "funções elementares" (funções racionais, trigonometricas, logarítmicas, exponenciais, radicais, etc.) e umas poucas extensões (função de erro, dilogaritmo). Nã consegue calcular integrais em termos de funções desconhecidas tais como `g(x)` e `h(x)`.

20.2 Definições para Integração

`changevar` (*expr*, *f*(*x*,*y*), *y*, *x*) [Função]

Faz a mudança de variável dada por *f*(*x*,*y*) `=` `0` em todos os integrais que existam em *expr* com integração em relação a *x*. A nova variável é *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
```

Uma expressão contendo uma forma substantiva, tais como as instâncias de `'integrate` acima, pode ser avaliada por `ev` com o sinalizador `nouns`. Por exemplo, a expressão retornada por `changevar` acima pode ser avaliada por `ev (%o3, nouns)`.

`changevar` pode também ser usada para alterações nos índices de uma soma ou de um produto. No entanto, é de salientar que quando seja feita uma alteração a uma soma ou produto, essa mudança deverá ser apenas uma deslocação do índice, nomeadamente, `i = j+ ...`, e não uma função de grau superior. Por exemplo,

```
(%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*) [Função]

Esta é uma rotina de integral duplo que foi escrita na linguagem de alto nível do Maxima sendo logo traduzida e compilada para linguagem de máquina. Use `load ("dblint")` para poder usar este pacote. Esta função usa o método da regra de Simpson em ambas as direções x e y para calcular

```
/b /s(x)
|  |
|  |    f(x,y) dy dx
|  |
/a /r(x)
```

A função *f* deve ser uma função traduzida ou compilada de duas variáveis, e *r* e *s* devem cada uma ser uma função traduzida ou compilada de uma variável, enquanto *a* e *b* devem ser números em ponto flutuante. A rotina tem duas variáveis globais que determinam o número de divisões dos intervalos x e y: `dblint_x` e `dblint_y`, ambas as quais são inicialmente 10, e podem ser alteradas independentemente para outros valores inteiros (existem `2*dblint_x+1` pontos calculados na direção x , e `2*dblint_y+1` na direção y). A rotina subdivide o eixo X e então para cada valor de X primeiro calcula `r(x)` e `s(x)`; então o eixo Y entre `r(x)` e `s(x)` é subdividido e o integral ao longo do eixo Y é executado usando a regra de Simpson; então o integral ao longo do eixo X é concluído usando a regra de Simpson com os valores da função sendo os integrais em Y. Esse procedimento pode ser numericamente instável por várias razões, mas razoávelmente rápido: evite usar este progrma sobre funções altamente oscilatórias e funções com singularidades (pólos ou pontos de ramificação na região). Os integrais em Y dependem de quanto fragmentados `r(x)` e `s(x)` sejam; assim, se a distância `s(x) - r(x)` variar rapidamente com X, nesse ponto podrão surgir erros substanciais provenientes de truncação com saltos de diferentes tamanhos nos vários integrais Y. Pode incrementar-se `dblint_x` e `dblint_y` numa tentativa para melhorar a convergência da região, com um aumento no tempo de computação. Os valores da função não são guardados, portanto se a função desperdiçr muito tempo, terá de esperar pela re-computação cada vez que mudar qualquer coisa (pedimos desculpa por esse facto). É necessário que as funções *f*, *r*, e *s* sejam ainda traduzidas ou

compiladas previamente chamando `dblint`. Isso resultará em ordens de magnitude de melhoramentos de velocidade sobre o código interpretado em muitos casos!

`demo (dblint)` executa uma demonstração de `dblint` aplicado a um problema exemplo.

`defint` (*expr*, *x*, `a`, *b*) [Função]

Tenta calcular um integral definido. `defint` é chamada por `integrate` quando limites de integração são especificados, i.e., quando `integrate` é chamado como `integrate (`*expr*`, `*x*`, a, `*b*`)`. Dessa forma do ponto de vista do utilizador, isso é suficiente para chamar `integrate`.

`defint` retorna uma expressão simbólica, e executa um dos dois: ou calcula o integral ou a forma substantiva do integral. Veja `quad_qag` e funções rellacionadas para aproximação numérica de integrais definidos.

`erf` (*x*) [Função]

Representa a função de erro, cuja derivada é: `2*exp(-x^2)/sqrt(%pi)`.

`erfflag` [Variável de opção]

Valor por omissão: `true`

Quando `erfflag` é `false`, previne `risch` da introdução da função `erf` na resposta se não houver nenhum no integrando para começar.

`ilt` (*expr*, *t*, *s*) [Função]

Calcula a transformação inversa de Laplace de *expr* em relação a *t* e parâmetro *s*. *expr* deve ser uma razão de polinómios cujo denominador tem somente factores lineares e quadráticos. Usando a funções `laplace` e `ilt` juntas com as funções `solve` ou `linsolve` o utilizador pode resolver uma diferencial simples ou uma equação integral de convolução ou um conjunto delas.

```
(%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*) [Função]
`integrate` (*expr*, *x*, *a*, *b*) [Função]

Tenta símbolicamente calcular o integral de *expr* em relação a *x*. `integrate (`*expr*`,` *x*`)` é um integral indefinido, enquanto `integrate (`*expr*`,` *x*`,` *a*`,` *b*`)` é um integral definido, com limites de integração *a* e *b*. Os limites não poderam conter *x*, embora `integrate` não imponha essa restrição. *a* não precisa ser menor que *b*. Se *b* é igual a *a*, `integrate` retorna zero.

Veja `quad_qag` e funções relacionadas para aproximação numérica de integrais definidos. Veja `residue` para computação de resíduos (integração complexa). Veja `antid` para uma forma alternativa de calcular integrais indefinidos.

O integral (uma expressão livre de `integrate`) é calculado se `integrate` for bem sucedido. De outra forma o valor de retorno é a forma substantiva do integral (o operador com apóstrofo `'integrate`) ou uma expressão contendo uma ou mais formas substantivas. A forma substantiva de `integrate` é apresentada com um símbolo de integração.

Em algumas circunstâncias isso é útil para construir uma forma substantiva manualmente, colocando em `integrate` um apóstrofo, e.g., `'integrate (`*expr*`,` *x*`)`. Por exemplo, o integral pode depender de alguns parâmetos que não estão ainda calculados. A forma substantiva pode ser aplicada a seus argumentos por `ev (`*i*`, nouns)` onde *i* é a forma substantiva de interesse.

`integrate` calcula integrais definidos separadamente dos indefinidos, e utiliza uma gama de heurísticas para simplificar cada caso. Casos especiais de integrais definidos incluem limites de integração iguais a zero ou infinito (`inf` ou `minf`), funções trigonométricas com limites de integração iguais a zero e `%pi` ou `2 %pi`, funções racionais, integrais relacionados com as definições das funções `beta` e `psi`, e alguns integrais logarítmicos e trigonométricos. O processamento de funções racionais pode incluir cálculo de resíduos. Se um caso especial aplicável não for encontrado, será feita uma tentativa para calcular o integral indefinido e avaliá-lo nos limites de integração. Isso pode incluir o cálculo de um limite nos casos em que um dos limites do integral for para infinito ou menos infinito; veja também `ldefint`.

Casos especiais de integrais indefinidos incluem funções trigonométricas, exponenciais e funções logarítmicas, e funções racionais. `integrate` pode também fazer uso de uma pequena tabela de integais elementares.

`integrate` pode realizar uma mudança de variável se o integrando tiver a forma `f(g(x)) * diff(g(x), x)`. `integrate` tenta achar uma subexpressão `g(x)` de forma que a derivada de `g(x)` divida o integrando. Essa busca pode fazer uso de derivadas definidas pela função `gradef`. Veja também `changevar` e `antid`.

Se nenhum dos procedimentos heurísticos conseguir calcular o integral indefinido, o algoritmo de Risch é executado. O sinalizador `risch` pode ser utilizado como um parâmetro para `ev`, ou na linha de comando, nomeadamente, `ev (integrate (`*expr*`, `*x*`), risch)` ou `integrate (`*expr*`, `*x*`), risch`. Se `risch` estiver presente, `integrate` chamará a função `risch` sem tentar heurísticas primeiro. Veja também `risch`.

`integrate` trabalha somente com relações funcionais representadas explicitamente com a notação `f(x)`. `integrate` não respeita dependências implicitas estabelecidas pela função `depends`. `integrate` pode necessitar conhecer alguma propriedade de um parâmetro no integrando. `integrate` irá primeiro consultar a base de dados do `assume`, e , se a variável de interesse não está lá, `integrate` perguntará ao utilizador. Dependendo da pergunta, respostas adequadas são `yes;` ou `no;`, ou `pos;`, `zero;`, ou `neg;`.

`integrate` não é, por padrão, declarada ser linear. Veja `declare` e `linear`.

`integrate` tenta integração por partes somente em uns poucos casos especiais.

Exemplos:

- Integrais definidos e indefinidos elementares.

```
(%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
```

- Uso de `assume` e dúvida interativa.

```
(%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
```

- Mudança de variável. Existem duas mudanças de variável nesse exemplo: uma usando a derivada estabelecida por `gradef`, e uma usando a derivação `diff(r(x))` de uma função não especificada `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)))
```

- O resultado contém a forma substantiva `'integrate`. Neste exemplo, Maxima pode extrair um factor do denominador de uma função racional, mas não pode factorizar o restante ou de outra forma achar o seu integral. `grind` mostra a forma substantiva `'integrate` no resultado. Veja também `integrate_use_rootsof` para mais informaçes sobre integrais de funções racionais.

```
(%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$
```

- Definindo uma função em termos de um integral. O corpo de uma função não é avaliado quando a função é definida. Dessa forma o corpo de `f_1` nesse exemplo contém a forma substantiva de `integrate`. O operador de doi apóstrofos seguidos `''` faz com que o integral seja avaliado, e o resultado se transforme-se no corpo de `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_counter` [Variável de sistema]

Valor por omissão: 0

`integration_constant_counter` é um contador que é actualizado a cada vez que uma constante de integração (nomeada pelo Maxima, por exemplo, `integrationconstant1`) é introduzida numa expressão obtida após a integração indefinida de uma equação.

`integrate_use_rootsof` [Variável de opção]

Valor por omissão: `false`

Quando `integrate_use_rootsof` é `true` e o denominador de uma função racional não pode ser factorizado, `integrate` retorna o integral em uma forma que é uma soma sobre as raízes (não conhecidas ainda) do denominador.

Por exemplo, com `integrate_use_rootsof` escolhido para `false`, `integrate` retorna um integral não resolvido de uma função racional na forma substantiva:

```
(%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)
```

Agora vamos escolher o sinalizador para ser true e a parte não resolvida do integral será escrito como uma soma sobre as raízes do denominador da função racional:

```
(%i3) integrate_use_rootsof: true$
(%i4) integrate (1/(1+x+x^5), x);
      ====       2
      \       (%r4  - 4 %r4 + 5) log(x - %r4)
       >      ------------------------------
      /                    2
      ====           3 %r4  - 2 %r4
                       3      2
```

```
      %r4 in rootsof(%r4  - %r4  + 1, %r4)
(%o4) ------------------------------------------------------------
                7

                                                          2 x + 1
                                              2        5 atan(-------)
                                        log(x  + x + 1)         sqrt(3)
                                      - --------------- + ---------------
                                             14             7 sqrt(3)
```

Alternativamente o utilizador pode calcular as raízes do denominador separadamente, e então expressar o integrando em termos dessas raízes, e.g., `1/((x - a)*(x - b)*(x - c))` ou `1/((x^2 - (a+b)*x + a*b)*(x - c))` se o denominador for um polinómio cúbico. Algumas vezes isso ajudará Maxima a obter resultados mais úteis.

`ldefint` (*expr*, *x*, *a*, *b*) [Função]

Tenta calcular o integral definido de *expr* pelo uso de `limit` para avaliar o integral indefinido *expr* em relação a *x* no limite superior *b* e no limite inferior *a*. Se isso falha para calcular o integral definido, `ldefint` retorna uma expressão contendo limites como formas substantivas.

`ldefint` não é chamada por `integrate`, então executando `ldefint (`*expr*`, `*x*`, a, `*b*`)` pode retornar um resultado diferente de `integrate (`*expr*`, `*x*`, a, `*b*`)`. `ldefint` sempre usa o mesmo método para avaliar o integral definido, enquanto `integrate` pode utilizar várias heurísticas e pode reconhecer alguns casos especiais.

`potential` (*givengradient*) [Função]

O cálculo faz uso da variável global `potentialzeroloc[0]` que deve ser `nonlist` ou da forma

```
[indeterminatej=expressãoj, indeterminatek=expressãok, ...]
```

O formador sendo equivalente para a expressão nonlist para todos os lados direitos-manuseados mais tarde. Os lados direitos indicados são usados como o limite inferior de integração. O sucesso das integrações pode depender de seus valores e de sua ordem. `potentialzeroloc` é inicialmente escolhido para 0.

`residue` (*expr*, *z*, *z_0*) [Função]

Calcula o resíduo no plano complexo da expressão *expr* quando a variável *z* assumes o valor *z_0*. O resíduo é o coeficiente de `(z - z_0)^(-1)` nas séries de Laurent para *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*) [Função]

Integra *expr* em relação a *x* usando um caso transcendental do algoritmo de Risch. (O caso algébrico do algoritmo de Risch foi implementado.) Isso actualmente manuseia os casos de exponenciais aninhadas e logaritmos que a parte principal de `integrate` não pode fazer. `integrate` irá aplicar automaticamente `risch` se dados esses casos.

`erfflag`, se `false`, previne `risch` da introdução da função `erf` na resposta se não for achado nenhum no integrando para começar.

```
(%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*) [Função]

Equivalente a `ldefint` com `tlimswitch` escolhido para `true`.

20.3 Introdução a QUADPACK

QUADPACK é uma colecção de funções para aálculo numérico de integrais definidos unidimensionais. O pacote QUADPACK resultou da junção de um projeto de R. Piessens[1], E. de Doncker[2], C. Ueberhuber[3], e D. Kahaner[4].

A biblioteca QUADPACK inclída no Maxima é uma tradução automática (feita através do programa `f2cl`) do código fonte em de QUADPACK como aparece na SLATEC Common Mathematical Library, Versão 4.1[5]. A biblioteca Fortran SLATEC é datada de Julho de 1993, mas as funções QUADPACK foram escritas alguns anos antes. Existe outra versão de QUADPACK em Netlib[6]; não está claro no que aquela versão difere da versão existente em SLATEC.

As funções QUADPACK incluídas no Maxima são toda automáticas, no sentido de que essas funções tentam calcular um resultado para uma precisão específica, requerendo um número não especificado de avaliações de função. A tradução do Lisp do Maxima da iblioteca QUADPACK também inclui algumas funçẽs não automáticas, mas elas não são expostas a nível de Maxima.

Informação adicionalsobre a bilioteca QUADPACK pode ser encontrada no livro do QUADPACK[7].

---

[1] Applied Mathematics and Programming Division, K.U. Leuven

[2] Applied Mathematics and Programming Division, K.U. Leuven

[3] Institut fur 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, e D.K. Kahaner. *QUADPACK: A Subroutine Package for Automatic Integration.* Berlin: Springer-Verlag, 1983, ISBN 0387125531.

20.3.1 Overview

`quad_qag` Integração de uma função genérica sobre um intervalo finito. `quad_qag` implementa um integrador adaptativo globalmente simples usando a estratégia de Aind (Piessens, 1973). O chamador pode escolher entre 6 pares de formulas da quadratura de Gauss-Kronrod para a componente de avaliação da regra. As regras de alto grau são adequadas para integrandos fortemente oscilantes.

`quad_qags`
Integração de uma função genérica sob um intervalo finito. `quad_qags` implementa subdivisão de intervalos globalmente adaptativos com extrapolação (de Doncker, 1978) por meio do algoritmo de Epsilon (Wynn, 1956).

`quad_qagi`
Integração de uma função genérica sobre um intervalo finito ou semi-finito. O intervalo é mapeado sobre um intervalo finito e então a mesma estratégia de `quad_qags` é aplicada.

`quad_qawo`
Integração de $cos(omegax)f(x)$ ou $sin(omegax)f(x)$ sobre um intervalo finito, onde *omega* é uma constante. A componente de avaliação da regra é baseada na técnica modificada de Clenshaw-Curtis. `quad_qawo` aplica subdivisão adaptativa com extrapolação, similar a `quad_qags`.

`quad_qawf`
Calcula uma transformação de co-seno de Fourier ou de um seno de Fourier sobre um intervalo semi-finito. O mesmo aproxima como `quad_qawo` aplicado sobre intervalos finitos sucessivos, e aceleração de convergência por meio d algorítimo de Epsilon (Wynn, 1956) aplicado a séries de contribuições de integrais.

`quad_qaws`
Integraçào de $w(x)f(x)$ sobre um intervalo finito $[a, b]$, onde w é uma função da forma $(x-a)^a lpha(b-x)^b etav(x)$ e $v(x)$ é 1 ou $log(x-a)$ ou $log(b-x)$ ou $log(x-a)log(b-x)$, e $alpha > -1$ e $beta > -1$. Auma estratégia de subdivisão adaptativa é aplicada, com integração modificada de Clenshaw-Curtis sobre os subintervalos que possuem a ou b.

`quad_qawc`
Calcula o valor principal de Cauchy de $f(x)/(x-c)$ sobre um intervalo finito (a, b) e um c especificado. A estratégia é globalmente adaptativa, e a integração modificada de Clenshaw-Curtis é usada sobre subamplitudes que possuírem o ponto $x = c$.

20.4 Definições para QUADPACK

`quad_qag` (*f*(*x*), *x*, *a*, *b*, *chave*, *epsrel*, *limite*) [Função]
`quad_qag` (*f*, *x*, *a*, *b*, *chave*, *epsrel*, *limite*) [Função]
Integração de uma função genérica sobre um intervalo finito. `quad_qag` implementa um integrador adaptativo globalmente simples usando a estratégia de Aind (Piessens, 1973). O chamador pode escolher entre 6 pares de fórmulas da quadratura de Gauss-

Kronrod para a componente de avaliação da regra. As regras de alto nível são adequadas para integrandos fortemente oscilatórios.

`quad_qag` calcula o integral

$$\int_a^b f(x)dx$$

A função a ser integrada é *f(x)*, com variável dependente *x*, e a função é para ser integrada entre os limites *a* e *b*. *chave* é o integrador a ser usado e pode ser um inteiro entre 1 e 6, inclusive. O valor de *chave* selecciona a ordem da regra de integração de Gauss-Kronrod. Regra de alta ordem são adequadas para integrandos fortemente oscilatórios.

O integrando pode ser especidficado como o nome de uma função Maxima ou uma função Lisp ou um operador, uma expressão lambda do Maxima, ou uma expressão geral do Maxima.

A integração numérica é concluída adaptativamente pela subdivisão a região de integração até que a precisão desejada for completada.

Os argumentos opcionais *epsrel* e *limite* são o erro relativo desejado e o número máximo de subintervalos respectivamente. *epsrel* padrão em 1e-8 e *limite* é 200.

`quad_qag` retorna uma lista de quatro elementos:

- uma aproximação para o integral,
- o erro absoluto estimado da aproximação,
- o número de avaliações do integrando,
- um código de erro.

O código de erro (quarto elemento do valor de retorno) pode ter os valores:

`0` se nenhum problema foi encontrado;

`1` se foram utilizados muitos subintervalos;

`2` se for detectato um erro de arredondamento excessivo;

`3` se o integrando se comportar muito mal;

`6` se a entrada não for válida.

Exemplos:

```
(%i1) quad_qag (x^(1/2)*log(1/x), x, 0, 1, 3);
(%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*, *limite*) [Função]
`quad_qags` (*f*, *x*, `a`, *b*, *epsrel*, *limite*) [Função]

Integração de uma função geral sobre um intervalo finito. `quad_qags` implementa subdivisão de intervalo globalmente adaptativa com extrapolação (de Doncker, 1978) através do algoritmo de (Wynn, 1956).

`quad_qags` calcula o integral

$$\int_a^b f(x)dx$$

A função a ser integrada é *f(x)*, com variável dependente *x*, e a função é para ser integrada entre os limites *a* e *b*.

O integrando pode ser especidficado como o nome de uma função Maxima ou uma função Lisp ou um operador, uma expressão lambda do Maxima, ou uma expressão geral do Maxima.

Os argumentos opcionais *epsrel* e *limite* são o erro relativo desejado e o número máximo de subintervalos, respectivamente. *epsrel* padrão em 1e-8 e *limite* é 200.

`quad_qags` retorna uma lista de quatro elementos:

- uma aproximação para o integral,
- o erro absoluto estimado da aproximação,
- o número de avaliações do integrando,
- um código de erro.

O código de erro (quarto elemento do valor de retorno) pode ter os valores:

`0`	nenhum problema foi encontrado;
`1`	foram utilizados muitos subintervalos;
`2`	foi detectato um erro de arredondamento excessivo;
`3`	o integrando comporta-se muito mal;
`4`	não houve convergência
`5`	o integral provavelmente é divergente, o converge lentamente
`6`	a entrada não foi válida.

Exemplos:

```
(%i1) quad_qags (x^(1/2)*log(1/x), x, 0 ,1);
(%o1)   [.4444444444444448, 1.11022302462516E-15, 315, 0]
```

Note que `quad_qags` é mais preciso e eficiente que `quad_qag` para esse integrando.

`quad_qagi` (*f*(*x*), *x*, `a`, *inftype*, *epsrel*, *limite*) [Função]
`quad_qagi` (*f*, *x*, `a`, *inftype*, *epsrel*, *limite*) [Função]

Integração de uma função genérica sobre um intervalo finito ou semi-finito. O intervalo é mapeado sobre um intervalo finito e então a mesma estratégia que em `quad_qags` é aplicada.

`quad_qagi` avalia um dos seguintes integrais

$$\int_a^\infty f(x)dx$$

$$\int_\infty^a f(x)dx$$

$$\int_{-\infty}^{\infty} f(x)dx$$

usando a rotina Quadpack QAGI. A função a ser integrada é *f(x)*, com variável dependente *x*, e a função é para ser integrada sobre um intervalo infinito.

O integrando pode ser especidficado como o nome de uma função Maxima ou uma função Lisp ou um operador, uma expressão lambda do Maxima, ou uma expressão geral do Maxima.

O parâmetro *inftype* determina o intervalo de integração como segue:

`inf` O intervalo vai de *a* ao infinito positivo.

`minf` O intervalo vai do infinito negativo até *a*.

`both` O intervalo corresponde a toda reta real.

Os argumentos opcionais *epsrel* e *limite* são o erro relativo desejado e o número maximo de subintervalos, respectivamente. *epsrel* padrão para 1e-8 e *limite* é 200.

`quad_qagi` retorna uma lista de quatro elementos:

- uma aproximação para o integral,
- o erro absoluto estimado da aproximação,
- o número de avaliações do integrando,
- um código de erro.

O código de erro (quarto elemento do valor de retorno) pode ter os valores:

`0` nenhum problema foi encontrado;

`1` foram utilizados muitos subintervalos;

`2` foi detectato um erro de arredondamento excessivo;

`3` o integrando comporta-se muito mal;

`4` não houve convergência

`5` o integral provavelmente é divergente, o converge lentamente

`6` a entrada não foi válida.

Exemplos:

```
(%i1) quad_qagi (x^2*exp(-4*x), x, 0, inf);
(%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*, *limite*) [Função]
`quad_qawc` (*f*, *x*, *c*, *a*, *b*, *epsrel*, *limite*) [Função]

Calcula o valor principal de Cauchy de $f(x)/(x-c)$ over a finite interval. A estratégia é globalmente adaptativa, e a integração de Clenshaw-Curtis modificada é usada sobre as subamplitudes que possuírem o ponto $x = c$.

`quad_qawc` calcula o valor principal de Cauchy de

$$\int_a^b \frac{f(x)}{x-c}\,dx$$

usando a rotina Quadpack QAWC. A função a ser integrada é `f(x)/(x - c)`, com variável dependente *x*, e a função é para ser integrada sobre o intervalo que vai de *a* até *b*.

O integrando pode ser especidficado como o nome de uma função Maxima ou uma função Lisp ou um operador, uma expressão lambda do Maxima, ou uma expressão geral do Maxima.

Os argumentos opcionais *epsrel* e *limite* são o erro relativo desejado e o máximo número de subintervalos, respectivamente. *epsrel* padrão para 1e-8 e *limite* é 200.

`quad_qawc` retorna uma lista de quatro elementos:

- uma aproximação para o integral,
- o erro absoluto estimado da aproximação,
- o número de avaliações do integrando,
- um código de erro.

O código de erro (quarto elemento do valoor de retorno) pode ter os valores:

`0` nenhum problema foi encontrado;

`1` foram utilizados muitos subintervalos;

`2` foi detectato um erro de arredondamento excessivo;

`3` o integrando comporta-se muito mal;

`6` a entrada não foi válida.

Exemplos:

```
(%i1) quad_qawc (2^(-5)*((x-1)^2+4^(-5))^(-1), x, 2, 0, 5);
(%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*) [Função]
quad_qawf (*f*, *x*, a, *omega*, *trig*, *epsabs*, *limit*, *maxp1*, *limlst*) [Função]

Calcula uma transformação de co-seno de Fourier ou de um seno de Fourier sobre um intervalo semi-finito. usando a função QAWF do pacote Quadpack. A mesma aproxima como em `quad_qawo` quando aplicada sobre intervalos finitos sucessivos, e aceleração de convergência por meio d algorítimo de Epsilon (Wynn, 1956) aplicado a séries de contribuições de integrais.

`quad_qawf` calcula o integral

$$\int_a^\infty f(x)w(x)dx$$

A função peso w é seleccionada por *trig*:

`cos` $w(x) = cos(omegax)$

`sin` $w(x) = sin(omegax)$

O integrando pode ser especidficado como o nome de uma função Maxima ou uma função Lisp ou um operador, uma expressão lambda do Maxima, ou uma expressão geral do Maxima.

Os argumentos opcionais são:

`epsabs` Erro absoluto de aproximação desejado. Padrão é 1d-10.

`limit` Tamanho de array interno de trabalho. (*limit* - *limlst*)/2 é o maximo número de subintervalos para usar. O Padrão é 200.

`maxp1` O número máximo dos momentos de Chebyshev. Deve ser maior que 0. O padrão é 100.

`limlst` Limite superior sobre número de ciclos. Deve ser maior ou igual a 3. O padrão é 10.

epsabs e *limit* são o erro relativo desejado e o número maximo de subintervalos, respectivamente. *epsrel* padrão para 1e-8 e *limit* é 200.

`quad_qawf` retorna uma lista de quatro elementos:

- uma aproximação para o integral,
- o erro absoluto estimado da aproximação,
- o número de avaliações do integrando,
- um código de erro.

O código de erro (quarto elemento do valor de retorno) pode ter os valores:

`0` nenhum problema foi encontrado;

`1` foram utilizados muitos subintervalos;

`2` foi detectato um erro de arredondamento excessivo;

`3` o integrando comporta-se muito mal;

`6` a entrada não foi válida.

Exemplos:

```
(%i1) quad_qawf (exp(-x^2), x, 0, 1, 'cos);
(%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*, *epsabs*, *limite*, *maxp1*, *limlst*) [Função]
`quad_qawo` (*f*, *x*, *a*, *b*, *omega*, *trig*, *epsabs*, *limite*, *maxp1*, *limlst*) [Função]

Integração de $cos(omegax)f(x)$ ou $sin(omegax)f(x)$ sobre um intervalo finito, onde *omega* é uma constante. A componente de avaliação da regra é baseada na técnica modificada de Clenshaw-Curtis. `quad_qawo` aplica subdivisão adaptativa com extrapolação, similar a `quad_qags`.

`quad_qawo` calcula o integral usando a rotina Quadpack QAWO:

$$\int_a^b f(x)w(x)dx$$

A função peso w é seleccionada por *trig*:

`cos` $w(x) = cos(omegax)$

`sin` $w(x) = sin(omegax)$

O integrando pode ser especidficado como o nome de uma função Maxima ou uma função Lisp ou um operador, uma expressão lambda do Maxima, ou uma expressão geral do Maxima.

Os argumentos opcionais são:

`epsabs` Erro absoluto desejado de aproximação. O Padrão é 1d-10.

`limite` Tamanho do array interno de trabalho. (*limite* - *limlst*)/2 é o número máximo de subintervalos a serem usados. Default é 200.

`maxp1` Número máximo dos momentos de Chebyshev. Deve ser maior que 0. O padrão é 100.

`limlst` Limite superior sobre o número de ciclos. Deve ser maior que ou igual a 3. O padrão é 10.

epsabs e *limite* são o erro relativo desejado e o número máximo de subintervalos, respectivamente. *epsrel* o padrão é 1e-8 e *limite* é 200.

`quad_qawo` retorna uma lista de quatro elementos:

- uma aproximação para o integral,
- o erro absoluto estimado da aproximação,
- o número de avaliações do integrando,
- um código de erro.

O código de erro (quarto elemento do valor de retorno) pode ter os valores:

`0` nenhum problema foi encontrado;

`1` foram utilizados muitos subintervalos;

`2` foi detectato um erro de arredondamento excessivo;

`3` o integrando comporta-se muito mal;

`6` a entrada não foi válida.

Exemplos:

```
(%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*, *epsabs*, *limite*) [Função]
`quad_qaws` (*f*, *x*, *a*, *b*, *alpha*, *beta*, *wfun*, *epsabs*, *limite*) [Função]

Integração de $w(x)f(x)$ sobre um intervalo finito, onde $w(x)$ é uma certa função algébrica ou logarítmica. Uma estratégia de subdivisão globalmente adaptativa é aplicada, com integração modificada de Clenshaw-Curtis sobre os subintervalos que possuírem os pontos finais dos intervalos de integração.

`quad_qaws` calcula o integral usando a rotina Quadpack QAWS:

$$\int_a^b f(x)w(x)dx$$

A função peso w é seleccionada por *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` $w(x) = (x-a)^a lpha(b-x)^b etalog(x-a)log(b-x)$

O integrando pode ser especificado como o nome de uma função Maxima ou uma função Lisp ou um operador, uma expressão lambda do Maxima, ou uma expressão geral do Maxima.

O argumentos opcionais são:

`epsabs` Erro absoluto desejado de aproximação. O padrão é 1d-10.

`limite` Tamanho do array interno de trabalho. (*limite* - *limlst*)/2 é o número máximo de subintervalos para usar. O padrão é 200.

epsabs e *limit* são o erro relativo desejado e o número máximo de subintervalos, respectivamente. *epsrel* o padrão é 1e-8 e *limite* é 200.

`quad_qaws` retorna uma lista de quatro elementos:

- uma aproximação para o integral,
- o erro absoluto estimado da aproximação,
- o número de avaliações do integrando,
- um código de erro.

O código de erro (quarto elemento do valor de retorno) pode ter os valores:

`0` nenhum problema foi encontrado;

`1` foram utilizados muitos subintervalos;

`2` foi detectato um erro de arredondamento excessivo;

`3` o integrando comporta-se muito mal;

`6` a entrada não foi válida.

Exemplos:

```
(%i1) quad_qaws (1/(x+1+2^(-4)), x, -1, 1, -0.5, -0.5, 1);
(%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 Equações

21.1 Definições para Equações

`%rnum_list` [Variável]

Valor por omissão: `[]`

`%rnum_list` é a lista de variáveis introduzidas em soluções por `algsys`. `%r` variáveis São adicionadas a `%rnum_list` na ordem em que forem criadas. Isso é conveniente para fazer substituições dentro da solução mais tarde. É recomendado usar essa lista em lugar de fazer `concat ('%r, j)`.

`algexact` [Variável]

Valor por omissão: `false`

`algexact` afecta o comportamento de `algsys` como segue:

Se `algexact` é `true`, `algsys` sempre chama `solve` e então usa `realroots` sobre falhas de `solve`.

Se `algexact` é `false`, `solve` é chamada somente se o eliminante não for de uma variável, ou se for uma quadrática ou uma biquadrada.

Dessa forma `algexact: true` não garante soluções exactas, apenas que `algsys` tentará primeiro pegar soluções exactas, e somente retorna aproximações quando tudo mais falha.

`algsys` ([*expr_1*, ..., *expr_m*], [*x_1*, ..., *x_n*]) [Função]
`algsys` ([*eqn_1*, ..., *eqn_m*], [*x_1*, ..., *x_n*]) [Função]

Resolve polinómios simultâneos *expr_1*, ..., *expr_m* ou equações polinômiais *eqn_1*, ..., *eqn_m* para as variáveis *x_1*, ..., *x_n*. Uma expressão *expr* é equivalente a uma equação *expr* = 0. Pode existir mais equações que variáveis ou vice-versa.

`algsys` retorna uma lista de soluções, com cada solução dada com uma lista de valores de estado das equações das variáveis *x_1*, ..., *x_n* que satisfazem o sistema de equações. Se `algsys` não pode achar uma solução, uma lista vazia `[]` é retornada.

Os símbolos `%r1`, `%r2`, ..., são introduzidos tantos quantos forem necessários para representar parâmetros arbitrários na solução; essas variáveis são também anexadas à lista `%rnum_list`.

O método usado é o seguinte:

(1) Primeiro as equações são factorizaadas e quebradas em subsistemas.

(2) Para cada subsistema *S_i*, uma equação *E* e uma variável *x* são seleccionados. A variável é escolhida para ter o menor grau não zero. Então a resultante de *E* e *E_j* em relação a *x* é calculada para cada um das equações restantes *E_j* nos subsistemas *S_i*. Isso retorna um novo subsistema *S_i'* em umas poucas variáveis, como *x* tenha sido eliminada. O processo agora retorna ao passo (1).

(3) Eventualmente, um subsistema consistindo de uma equação simples é obtido. Se a equação é de várias variáveis e aproximações na forma de números em ponto flutuante nã tenham sido introduzidas, então `solve` é chamada para achar uma solução exacta.

Em alguns casos, `solve` não está habilitada a achar uma solução, ou se isso é feito a solução pode ser uma expressão expressão muito larga.

Se a equação é de uma única variável e é ou linear, ou quadrática, ou biquadrada, então novamente `solve` é chamada se aproximações não tiverem sido introduzidas. Se aproximações tiverem sido introduzidas ou a equação não é de uma única variável e nem tão pouco linear, quadratica, ou biquadrada, então o comutador `realonly` é `true`, A função `realroots` é chamada para achar o valor real das soluções. Se `realonly` é `false`, então `allroots` é chamada a qual procura por soluções reais e complexas.

Se `algsys` produz uma solução que tem poucos digitos significativos que o requerido, o utilizador pode escolher o valor de `algepsilon` para um valor maior.

Se `algexact` é escolhido para `true`, `solve` será sempre chamada.

(4) Finalmente, as soluções obtidas no passo (3) são substituídas dentro dos níveis prévios e o processo de solução retorna para (1).

Quando `algsys` encontrar uma equação de várias variáveis que contém aproximações em ponto flutuante (usualmente devido a suas falhas em achar soluções exactas por um estágio mais fácil), então não tentará aplicar métodos exatos para tais equações e em lugar disso imprime a mensagem: "`algsys` cannot solve - system too complicated."

Interações com `radcan` podem produzir expressões largas ou complicadas. Naquele caso, pode ser possível isolar partes do resultado com `pickapart` ou `reveal`.

Ocasionalmente, `radcan` pode introduzir uma unidade imaginária `%i` dentro de uma solução que é actualmente avaliada como real.

Exemplos:

```
++
     (%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*) [Função]
`allroots` (*eqn*) [Função]

Calcula aproximações numéricas de raízes reais e complexas do polinómio *expr* ou equação polinômial *eqn* de uma variável.

O sinalizador `polyfactor` quando `true` faz com que `allroots` factore o polinómio sobre os números reais se o polinómio for real, ou sobre os números complexos, se o polinómio for complexo.

`allroots` pode retornar resultados imprecisos no caso de múltiplas raízes. Se o polinómio for real, `allroots (%i*p)`) pode retornar aproximações mais precisas que `allroots (p)`, como `allroots` invoca um algoritmo diferente naquele caso.

`allroots` rejeita expressoões que não sejam polinómios. Isso requer que o numerador após a classificação (`rat`'ing) poderá ser um polinómio, e isso requer que o denominador seja quando muito um número complexo. Com esse tipo resultado `allroots` irá sempre produzir uma expressão equivalente (mas factorizada), se `polyfactor` for `true`.

Para polinómios complexos um algoritmo por Jenkins e Traub é usado (Algorithm 419, *Comm. ACM*, vol. 15, (1972), p. 97). Para polinómios reais o algoritmo usado é devido a Jenkins (Algorithm 493, *ACM TOMS*, vol. 1, (1975), p.178).

Exemplos:

```
(%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` [Variável]

Valor por omissão: `true`

Quando `backsubst` é `false`, evita substituições em expressões anteriores após as equações terem sido triangularizadas. Isso pode ser de grande ajuda em problemas muito grandes onde substituição em expressões anteriores pode vir a causar a geração de expressões extremamente largas.

`breakup` [Variável]

Valor por omissão: `true`

Quando `breakup` é `true`, `solve` expressa soluções de equações cúbicas e quárticas em termos de subexpressões comuns, que são atribuídas a rótulos de expressões intermédias (`%t1`, `%t2`, etc.). De outra forma, subexpressões comuns não são identificadas.

`breakup: true` tem efeito somente quando `programmode` é `false`.

Exemplos:

```
(%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*) [Função]

`dimension` (*eqn_1*, ..., *eqn_n*) [Função]

`dimen` é um pacote de análise dimensional. `load ("dimen")` chama esse pacote. `demo ("dimen")` mostra uma cura demostração.

`dispflag` [Variável]

Valor por omissão: `true`

Se escolhida para `false` dentro de um `block` inibirá a visualização da saída gerada pelas funções solve chamadas de dentro de `block`. Terminando `block` com um sinal de dolar, $, escolhe `dispflag` para `false`.

`funcsolve` (*eqn*, *g*(*t*)) [Função]

Retorna [*g*(*t*) = ...] ou [], dependendo de existir ou não uma função racional *g*(*t*) satisfazendo *eqn*, que deve ser de primeira ordem, polinómio linear em (para esse caso) *g*(*t*) e *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));

Equações dependentes eliminadas:  (4 3)
                                    n
(%o2)                  f(n) = ---------------
                                (n + 1) (n + 2)
```

Atenção: essa é uma implementação muito rudimentar – muitas verificações de segurança e obviamente generalizações estão ausêntes.

`globalsolve` [Variável]

Valor por omissão: `false`

When `globalsolve` for `true`, variáveis para as quais as equações são resolvidas são atribuidas aos valores da solução encontrados por `linsolve`, e por `solve` quando resolvendo duas ou mais equações lineares. Quando `globalsolve` for `false`, soluções encontradas por `linsolve` e por `solve` quando resolvendo duas ou mais equações lineares são espressas como equações, e as variáveis para as quais a equação foi resolvida não são atribuidas.

Quando resolvendo qualquer coisa outra que não duas equações lineares ou mais, `solve` ignora `globalsolve`. Outras funções que resolvem equações (e.g., `algsys`) sempre ignoram `globalsolve`.

Exemplos:

```
(%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*) [Função]

`inteqn` é um pacote para resolver equações integrais. `load ("inteqn")` carrega esse pacote.

ie é a equação integral; *unk* é a função desconhecida; *tech* é a técnica a ser tentada nesses dados acima (*tech* = `first` significa: tente a primeira técnica que achar uma solução; *tech* = `all` significa: tente todas a técnicas aplicáveis); *n* é o número máximo de termos a serem usados de `taylor`, `neumann`, `firstkindseries`, ou `fredseries` (isso é também o número máximo de ciclos de recurssão para o método de diferenciação); *guess* é o inicial suposto para `neumann` ou `firstkindseries`.

Valores padrão do segundo até o quinto parâmetro são:

unk: `p(x)`, onde *p* é a primeira função encontrada em um integrando que é desconhecida para Maxima e *x* é a variável que ocorre como um argumento para a primeira ocorrência de *p* achada fora de uma integral no caso de equações `secondkind` , ou é somente outra variável ao lado da variável de integração em equações `firstkind`. Se uma tentativa de procurar por *x* falha, o utilizador será perguntado para suprir a variável independente.

tech: `first`

n: 1

guess: `none` o que fará com que `neumann` e `firstkindseries` use `f(x)` como uma suposição inicial.

`ieqnprint` [Variável de opção]

Valor por omissão: `true`

`ieqnprint` governa o comportamento do resultado retornado pelo comando `ieqn`. Quando `ieqnprint` é `false`, as listas retornadas pela função `ieqn` são da forma

[*solução*, *tecnica usada*, *nterms*, *sinalizador*]

onde *sinalizador* é retirado se a solução for exacta.

De outra forma, isso é a palavra `approximate` ou `incomplete` correspondendo à forma inexacta ou forma aberta de solução, respectivamente. Se um método de série foi usado, *nterms* fornece o número de termos usados (que poderá ser menor que os n dados para `ieqn` se ocorrer um erro evita a geração de termos adicionais).

`lhs` (*expr*) [Função]

Retorna o lado esquerdo (isto é, o primeiro argumento) da expressão *expr*, quando o operador de *expr* for um dos operadores relacionais `< <= = # equal notequal >= >`, um dos operadores de atribuição `:= ::= : ::`, ou um operadro infixo definido pelo utilizador, como declarado por meio de `infix`.

Quando *expr* for um átomo ou seu operador for alguma coisa que não esses listados acima, `lhs` retorna *expr*.

Veja também `rhs`.

Exemplos:

```
(%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*]) [Função]

Resolve a lista de equações lineares simultâneas para a lista de variáveis. As expressões devem ser cada uma polinómios nas variáveis e podem ser equações.

Quando `globalsolve` é `true` então variáveis que foram resolvidas serão escolhidas para a solução do conjunto de equações simultâneas.

Quando `backsubst` é `false`, `linsolve` não realiza substituição em equações anteriores após as equações terem sido triangularizadas. Isso pode ser necessário em problemas muito grandes onde substituição em equações anteriores poderá causar a geração de expressões extremamente largas.

Quando `linsolve_params` for `true`, `linsolve` também gera símbolos `%r` usados para representar parâmetros arbitrários descritos no manual sob `algsys`. De outra forma, `linsolve` resolve um menor-determinado sistema de equações com algumas variáveis expressas em termos de outras.

Quando `programmode` for `false`, `linsolve` mostra a solução com expressões intermédias com rótulos (`%t`), e retorna a lista de rótulos.

```
(%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` [Variável]

Valor por omissão: `true`

Quando `linsolvewarn` é `true`, `linsolve` imprime uma mensagem "Dependent equações eliminated".

`linsolve_params` [Variável]

Valor por omissão: `true`

Quando `linsolve_params` é `true`, `linsolve` também gera os símbolos `%r` usados para representar parâmetros arbitrários descritos no manual sob `algsys`. De outra forma, `linsolve` resolve um menor-determinado sistema de equações com algumas variáveis expressas em termos e outras.

`multiplicities` [Variável]

Valor por omissão: `not_set_yet`

`multiplicities` é escolhida para uma lista de multiplicidades das soluções individuais retornadas por `solve` ou `realroots`.

`nroots` (*p*, *low*, *high*) [Função]

Retorna o número de raízes reais do polinómio real de uma única variável *p* no intervalo semi-aberto (*low*, *high*]. Uma extremidade do intervalo podem ser `minf` ou `inf`. infinito e mais infinito.

`nroots` usa o método das sequuências de Sturm.

```
(%i1) p: x^10 - 2*x^4 + 1/2$
(%i2) nroots (p, -6, 9.1);
(%o2)                          4
```

`nthroot` (*p*, *n*) [Função]

Onde p é um polinómio com coeficientes inteiros e n é um inteiro positivo retorna q, um polinómio sobre os inteiros, tal que q^n=p ou imprime uma mensagem de erro indicando que p não é uma potência n-ésima perfeita. Essa rotina é mais rápida que `factor` ou mesmo `sqfr`.

programmode [Variável]

Valor por omissão: true

Quando programmode é true, solve, realroots, allroots, e linsolve retornam soluções como elementos em uma lista. (Exceto quando backsubst é escolhido para false, nesse caso programmode: false é assumido.)

Quando programmode é false, solve, etc. cria rótulos de expressões intermédias %t1, t2, etc., e atribui as soluções para eles.

realonly [Variável]

Valor por omissão: false

Quando realonly é true, algsys retorna somente aquelas soluções que estão livres de %i.

realroots (*expr*, *bound*) [Função]
realroots (*eqn*, *bound*) [Função]
realroots (*expr*) [Função]
realroots (*eqn*) [Função]

Calcula aproximações racionais das raízes reais da expressão polinomial *expr* ou da equação polinomial *eqn* de uma variável, dentro de uma tolerância de *bound*. coeficientes de *expr* ou de *eqn* devem ser números literais; constantes símbolo tais como %pi são rejeitadas.

realroots atribui as multiplicidades das raízes que encontrar para a variável global multiplicities.

realroots constrói uma sequência de Sturm para delimitar cada raíz, e então palica a bisecção para redefinir as aproximações. Todos os coeficientes são convertidos para os equivalentes racionais antes da busca por raízes, e cálculos são realizados por meio de aritmética racional exacta. Mesmo se alguns coeficientes forem números em ponto flutuante, os resultados são racionais (a menos que forçados a números em ponto flutuante por float ou por numer flags).

Quando *bound* for menor que 1, todas as raízes inteiras são encontradas exactamente. Quando *bound* não for especificado, será assumido como sendo igual à variável globa rootsepsilon.

Quando a varável global programmode for true, realroots retorna uma lista da forma [x = *x_1*, x = *x_2*, ...]. Quando programmode for false, realroots cria rótulos de expressões intermédias %t1, %t2, ..., atribui os resultados a eles, e retorna a lista de rótulos.

Exemplos:

```
(%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*) [Função]

Retorna o lado direito (isto é, o segundo argumento) da expressão *expr*, quando o operador de *expr* for um dos operadores relacionais `< <= = # equal notequal >= >`, um dos operadores de atribuição `:= ::= : ::`, ou um operador binário infixo definido pelo utilizador, como declarado por meio de `infix`.

Quando *expr* for um étomo ou seu operadro for alguma coisa que não esses listados acima, `rhs` retorna 0.

Veja também `lhs`.

Exemplos:

```
(%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` [Variável de opção]

Valor por omissão: `true`

`rootsconmode` governa o comportamento do comando `rootscontract`. Veja `rootscontract` para detalhes.

`rootscontract` (*expr*) [Função]

Converte produtos de raízes em raízes de produtos. Por exemplo, `rootscontract (sqrt(x)*y^(3/2))` retorna `sqrt(x*y^3)`.

Quando `radexpand` é `true` e `domain` é `real`, `rootscontract` converte `abs` em `sqrt`, e.g., `rootscontract (abs(x)*sqrt(y))` retorna `sqrt(x^2*y)`.

Existe uma opção `rootsconmode` afectando `rootscontract` como segue:

```
Problem            Value of        Result of applying
                  rootsconmode        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)
```

Quando `rootsconmode` é `false`, `rootscontract` contrai somente como relação a expoentes de número racional cujos denominadores são os mesmos. A chave para os exemplos `rootsconmode: true` é simplesmente que 2 divides 4 mas não divide 3. `rootsconmode: all` envolve pegar o menor múltiplo comum dos denominadores dos expoentes.

`rootscontract` usa `ratsimp` em uma maneira similar a `logcontract`.

Exemplos:

```
(%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` [Variável de opção]

Valor por omissão: 1.0e-7

`rootsepsilon` é a tolerância que estabelece o intervalo de confiabilidade para as raízes achadas pela função `realroots`.

`solve (`*`expr`*`, `*`x`*`)` [Função]
`solve (`*`expr`*`)` [Função]
`solve ([`*`eqn_1`*`, ..., `*`eqn_n`*`], [`*`x_1`*`, ..., `*`x_n`*`])` [Função]

Resolve a equação algébrica *expr* para a variável *x* e retorna uma lista de equações solução em *x*. Se *expr* não é uma equação, a equação *expr* `= 0` é assumida em seu lugar. *x* pode ser uma função (e.g. `f(x)`), ou outra expressão não atômica excepto uma adição ou um produto. *x* pode ser omitido se *expr* contém somente uma variável. *expr* pode ser uma expressão racional, e pode conter funções trigonométricas, exponenciais, etc.

O seguinte método é usado:

Tome *E* sendo a expressão e *X* sendo a variável. Se *E* é linear em *X* então isso é trivialmente resolvido para *X*. De outra forma se *E* é da forma `A*X^N + B` então o resultado é `(-B/A)^1/N)` vezes as `N`'ésimas raízes da unidade.

Se *E* não é linear em *X* então o máximo divisor comum (mdc) dos expoentes de *X* em *E* (digamos *N*) é dividido dentro dos expoentes e a multiplicidade das raízes é multiplicada por *N*. Então `solve` é chamada novamente sobre o resultado. Se *E* for dada em factores então `solve` é chamada sobre cada um dos factores. Finalmente `solve` usará as fórmulas quadráticas, cúbicas, ou quárticas onde necessário.

No caso onde *E* for um polinómio em alguma função de variável a ser resolvida, digamos `F(X)`, então isso é primeiro resolvida para `F(X)` (chama o resultado *C*), então a equação `F(X)=C` pode ser resolvida para *X* fornecendo o inverso da função *F* que é conhecida.

`breakup` se `false` fará com que `solve` expresse as soluções de equações cúbicas ou quárticas como expressões simples ao invés de como feito em cima de várias subexpressões comuns que é o padrão.

`multiplicities` - será escolhido para uma lista de multiplicidades de soluções individuais retornadas por `solve`, `realroots`, ou `allroots`. Tente `apropos (solve)` para os comutadores que afectam `solve`. `describe` pode então ser usada sobre o nome do comutador individual se seu propósito não é claro.

`solve ([`*`eqn_1`*`, ..., `*`eqn_n`*`], [`*`x_1`*`, ..., `*`x_n`*`])` resolve um sistema de equações polinomiais (lineares ou não-lineares) simultâneas por chamada a `linsolve` ou `algsys` e retorna uma lista de listas solução nas variáveis. No caso de `linsolve` essa lista conterá uma lista simples de soluções. Isso pega duas listas como argumentos. A primeira lista representa as equações a serem resolvidas; a segunda lista é a lista de desconhecidos a ser determinada. Se o número total de variáveis nas equações é igual ao número de equações, a segunda lista-argumento pode ser omitida. Para sistemas lineares se as dadas equações não são compatíveis, a mensagem `inconsistent` será mostrada (veja o comutador `solve_inconsistent_error`); se não existe solução única, então `singular` será mostrado.

Exemplos:

```
(%i1) solve (asin (cos (3*x))*(f(x) - 1), x);
```

```
SOLVE is using arc-trig functions to get a solution.
Some soluções 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` [Variável de opção]

Valor por omissão: `true`

Quando `solvedecomposes` é `true`, `solve` chama `polydecomp` se perguntado para resolver polinómios.

`solveexplicit` [Variável de opção]

Valor por omissão: `false`

Quando `solveexplicit` é `true`, inibe `solve` de retornar soluções implícitas, isto é, soluções da forma `F(x) = 0` onde `F` é alguma função.

`solvefactors` [Variável de opção]

Valor por omissão: `true`

Quando `solvefactors` é `false`, `solve` não tenta factorizar a expressão. A escolha do `false` poderá ser útil em alguns casos onde a factorização não é necessária.

`solvenullwarn` [Variável de opção]

Valor por omissão: `true`

Quando `solvenullwarn` é `true`, `solve` imprime uma mensagem de alerta se chamada com ou uma lista equação ou uma variável lista nula. Por exemplo, `solve ([], [])` imprimirá duas mensagens de alerta e retorna `[]`.

`solveradcan` [Variável de opção]

Valor por omissão: `false`

Quando `solveradcan` é `true`, `solve` chama `radcan` que faz `solve` lento mas permitirá certamente que problemas contendo exponeniais e logaritmos sejam resolvidos.

`solvetrigwarn` [Variável de opção]

Valor por omissão: `true`

Quando `solvetrigwarn` é `true`, `solve` pode imprimir uma mensagem dizendo que está usando funções trigonométricas inversas para resolver a equação, e desse modo perdendo soluções.

`solve_inconsistent_error` [Variável de opção]

Valor por omissão: `true`

Quando `solve_inconsistent_error` é `true`, `solve` e `linsolve` resultam em erro se as equações a serem resolvidas são inconsistentes.

Se `false`, `solve` e `linsolve` retornam uma lista vazia `[]` se as equações forem inconsistentes.

Exemplo:

```
(%i1) solve_inconsistent_error: true$
(%i2) solve ([a + b = 1, a + b = 2], [a, b]);
Inconsistent equações:  (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 Equações Diferenciais

22.1 Introdução às Equações Diferenciais

Esta secção descreve as funções disponíveis no Maxima para obter a solução analítica de alguns tipos específicos de equações diferencias de primeira e segunda ordem. Para obter a solução numérica dum sistema de equações diferenciais, consulte o pacote adicional `dynamics`. Para obter representações gráficas no espaço de fase, consulte o pacote adicional `plotdf`.

22.2 Definições para Equações Diferenciais

`bc2` (*solução*, *xval1*, *yval1*, *xval2*, *yval2*) [Função]

Resolve um problema de valores fronteira para uma equação diferencial de segunda ordem. Aqui: *solução* é uma solução geral para a equação, calculada por `ode2`; *xval1* define o valor da variável independente, num primeiro ponto, na forma *x = x1*, e *yval1* define o valor da variável dependente, no mesmo ponto, na forma *y = y1*. As expressões *xval2* e *yval2* definem os valores das mesmas variáveis, num segundo ponto, usando a mesma forma.

Veja um exemplo da sua utilização na documentação de `ode2`.

`desolve` (*eqn*, *x*) [Função]
`desolve` ([*eqn_1*, ..., *eqn_n*], [*x_1*, ..., *x_n*]) [Função]

A função `dsolve` resolve sistemas de equações diferenciais ordinárias lineares usando transformada de Laplace. Aqui as expressões *eqn* são equações diferenciais nas variáveis dependentes *x_1*, ..., *x_n*. A relação funcional de *x_1*, ..., *x_n* na variável independente deve ser indicada explicitamente nas variáveis e nas suas derivadas. Por exemplo, esta forma de definir as equações não seria correcta:

```
eqn_1: 'diff(f,x,2) = sin(x) + 'diff(g,x);
eqn_2: 'diff(f,x) + x^2 - f = 2*'diff(g,x,2);
```

A forma correcta seria:

```
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);
```

Assim, a chamada à função `desolve` seria:

```
desolve([eqn_1, eqn_2], [f(x),g(x)]);
```

Se as condições iniciais em `x=0` forem conhecidas, poderão ser fornecidas antes de usar `desolve`, através de `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)]
```

Se `desolve` não pode obter uma solução, retorna `false`.

`ic1` (*solução*, *xval*, *yval*) [Função]

Resolve problemas de valor inicial para equações diferenciais de primeira ordem. Aqui *solução* é uma solução geral para a equação, na forma dada por `ode2`, *xval* dá um valor inicial para a variável independente, na forma *x* = *x0*, e *yval* dá o valor inicial para a variável dependente, na forma *y* = *y0*.

Veja um exemplo da sua utilização na documentação de `ode2`.

`ic2` (*solução*, *xval*, *yval*, *dval*) [Função]

Resolve problemas de valores iniciais para equações diferenciais de segunda ordem. Aqui *solução* é uma solução geral para a equação, na forma dada por `ode2`, *xval* dá um valor inicial para a variável independente, na forma *x* = *x0*, *yval* dá o valor inicial para a variável dependente, na forma *y* = *y0* e *dval* dá o valor inicial para a primeira derivada da variável dependente, em função da variável independente, na forma `diff(`*y*`,`*x*`)` = *dy0* (`diff` não tem que ser precedido por apóstrofo).

Veja um exemplo da sua utilização na documentação de `ode2`.

`ode2` (*eqn*, *dvar*, *ivar*) [Função]

A função `ode2` resolve uma equação diferencial ordinária (EDO) de primeira ou de segunda ordem. Precisa de três argumentos: uma EDO dada por *eqn*, a variável dependente *dvar*, e a variável independente *ivar*. Quando conseguir, retorna uma solução para a variável dependente, na forma explícita ou implícita. `%c` é usado para representar a constante de integração no caso de equações de primeira ordem, e `%k1` e `%k2` as constantes para equações de segunda ordem. A dependência da variável dependente na variável independente não tem que ser escrita em forma explícita, como no caso de `desolve`, mas a variável independente deverá ser indicada sempre no terceiro argumento.

Se por alguma razão `ode2` não conseguir encontrar a solução, retornará `false`, após talvez mostrar uma mensagem de erro. Os métodos implementados para equações diferenciais de primeira ordem, na ordem em que serão testados, são: linear, separável,

exacta - talvez requerendo um factor de integração, homogénea, equação de Bernoulli, homogénea generalizada. Os tipos de equações de segunda ordem que podem ser resolvidas são: coeficientes constantes, exactas, linear homogéneas com coeficientes não-constantes que possam ser transformados para constates, equação de Euler ou equi-dimensional, equações que possam ser resolvidas pelo método de variação dos parâmetros, e equações que não dependam ou da variável independente ou da variável dependente de modo que possam ser reduzidas a duas equações lineares de primeira ordem a serem resolvidas sequêncialmente.

Durante o processo de resolução da EDO, serão dados valores a várias variáveis locais, com fins puramente informativos: `método` denota o método de solução usado (por exemplo, `linear`), `intfactor` denota qualquer factor integrante utilizado, `odeindex` denota o índice para o método de Bernoulli ou para o método homogéneo generalizado, e `yp` denota a solução particular no método de variação dos parâmetros.

Para resolver problemas de valores iniciais (PVI) estão disponíveis as funções `ic1` e `ic2`e, para equações de primeira e segunda ordem, e para resolver problemas de valores fronteira (PVF) de segunda ordem pode usar-se a função `bc2`.

Exemplo:

```
(%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 Numérico

23.1 Introdução a Numérico

23.2 Pacotes de Fourier

O pacote `fft` compreende funções para computação numérica (não simbólica) das transformações rápidas de Fourier. `load ("fft")` chama esse pacote. Veja `fft`.

O pacote `fourie` compreende funções para computação simbólica de séries de Fourier. `load ("fourie")` chama esse pacote. Existem funções no pacote `fourie` para calcular coeficientes da integral de Fourier e algumas funções para manipulação de expressões. Veja `Definições para Séries`.

23.3 Definições para Numérico

`polartorect` (*magnitude_array*, *phase_array*) [Função]

Traduz valores complexos da forma `r %e^(%i t)` para a forma `a + b %i`. `load ("fft")` chama essa função dentro do Maxima. Veja também `fft`.

O módulo e a fase, `r` e `t`, São tomados de *magnitude_array* e *phase_array*, respectivamente. Os valores originais de arrays de entrada são substituídos pelas partes real e emaginária, `a` e `b`, no retorno. As saídas são calculadas como

```
a: r cos (t)
b: r sin (t)
```

Os arrays de entrada devem ter o mesmo tamanho e ser unidimensionais. O tamanho do array não deve ser uma potência de 2.

`polartorect` é a função inversa de `recttopolar`.

`recttopolar` (*real_array*, *imaginary_array*) [Função]

Traduz valores complexos da forma `a + b %i` para a forma `r %e^(%i t)`. `load ("fft")` chama essa função dentro do Maxima. Veja também `fft`.

As partes real e imaginária, `a` e `b`, são tomadas de *real_array* e *imaginary_array*, respectivamente. Os valores originais dos arrays de entrada são substituídos pelo módulo e pelo ângulo, `r` e `t`, no retorno. As saídas são calculadas como

```
r: sqrt (a^2 + b^2)
t: atan2 (b, a)
```

O ângulo calculado encontra-se no intervalo de `-%pi` a `%pi`.

Os arrays de entrada devem ter o mesmo tamanho e ser unidimensionais. O tamanho do array não deve ser uma potência de 2.

`recttopolar` é a função inversa de `polartorect`.

`ift` (*real_array*, *imaginary_array*) [Função]

Transformação rápida inversa discreta de Fourier . `load ("fft")` chama essa função dentro do Maxima.

`ift` realiza a transformação rápida complexa de Fourier sobre arrays em ponto flutuante unidimensionais. A transformação inversa é definida como

```
x[j]: sum (y[j] exp (+2 %i %pi j k / n), k, 0, n-1)
```

Veja `fft` para maiores detalhes.

`fft` (*real_array*, *imaginary_array*) [Função]
`ift` (*real_array*, *imaginary_array*) [Função]
`recttopolar` (*real_array*, *imaginary_array*) [Função]
`polartorect` (*magnitude_array*, *phase_array*) [Função]

Transformação rápidada de Fourier e funções relacionadas. `load ("fft")` chama essas funções dentro do Maxima.

`fft` e `ift` realiza transformação rápida complexa de Fourier e a transformação inversa, respectivamente, sobre arrays em ponto flutuante unidimensionais. O tamanho de *imaginary_array* deve ser igual ao tamanho de *real_array*.

`fft` e `ift` operam in-loco. Isto é, sobre o retorno de `fft` ou de `ift`, O conteúdo original dos arrays de entrada é substituído pela saída. A função `fillarray` pode fazer uma cópia de um array, isso pode ser necessário.

A transformação discreta de Fourier e sua transformação inversa são definidas como segue. Tome `x` sendo os dados originais, com

```
x[i]: real_array[i] + %i imaginary_array[i]
```

Tome `y` sendo os dados transformados. A transformação normal e sua transformação inversa são

```
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)
```

Arrays adequadas podem ser alocadas pela função `array`. Por exemplo:

```
array (my_array, float, n-1)$
```

declara um array unidimensional com n elementos, indexado de 0 a n-1 inclusive. O número de elementos n deve ser igual a 2^m para algum m.

`fft` pode ser aplicada a dados reais (todos os arrays imaginários são iguais a zero) para obter coeficientes seno e co-seno. Após chamar `fft`, os coeficientes seno e co-seno, digamos `a` e `b`, podem ser calculados como

```
a[0]: real_array[0]
b[0]: 0
```

e

```
a[j]: real_array[j] + real_array[n-j]
b[j]: imaginary_array[j] - imaginary_array[n-j]
```

para j variando de 1 a n/2-1, e

```
a[n/2]: real_array[n/2]
b[n/2]: 0
```

`recttopolar` traduz valores complexos da forma `a + b %i` para a forma `r %e^(%i t)`. Veja `recttopolar`.

`polartorect` traduz valores complexos da forma `r %e^(%i t)` para a forma `a + b %i`. Veja `polartorect`.

`demo ("fft")` exibe uma demonstração do pacote `fft`.

`fortindent` [Variável de opção]

Valor por omissão: 0

`fortindent` controla a margem esquerda de indentação de expressões mostradas pelo comando `fortran`. 0 fornece indentação normal (i.e., 6 espaços), e valores positivos farão com que expressões sejam mostrados mais além para a direita.

`fortran (`*expr*`)` [Função]

Mostra *expr* como uma declaração Fortran. A linha de saída é indentada com espaços. Se a linha for muito longa, `fortran` imprime linhas de continuação. `fortran` mostra o operador de exponenciação `^` como `**`, e mostra um número complexo `a + b %i` na forma `(a,b)`.

expr pode ser uma equação. Nesse caso, `fortran` mostra uma declaração de atribuição, atribuindo o primeiro membro (esquerda) da equação ao segundo membro (direita). Em particular, se o primeiro membro *expr* é um nome de uma matriz, então `fortran` mostra uma declaração de atribuição para cada elemento da matriz.

Se *expr* não for alguma coisa reconhecida por `fortran`, a expressão é mostrada no formato `grind` sem reclamação. `fortran` não conhece listas, arrays ou funções.

`fortindent` controla o margem esquerda das linhas mostradas. 0 é a margem normal (i.e., indentada 6 espaços). Incrementando `fortindent` faz com que expressões sejam mostradas adiante para a direita.

quando `fortspaces` for `true`, `fortran` preenche cada linha mostrada com espaços em branco até completar 80 columas.

`fortran` avalia seus argumentos; colocando um apóstrofo em um argumento evita avaliação. `fortran` sempre retorna `done`.

Exemplos:

```
(%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` [Variável de opção]

Valor por omissão: `false`

Quando `fortspaces` for `true`, `fortran` preenche cada linha mostrada com espaços em branco até completar 80 columas.

`horner` (*expr*, *x*) [Função]
`horner` (*expr*) [Função]

Retorna uma representação rearranjada de *expr* como na regra de Horner, usando *x* como variável principal se isso for especificado. `x` pode ser omitido e nesse caso a variável principal da forma de expressão racional canónica de *expr* é usada.

`horner` algumas vezes melhora a estabilidade se `expr` for ser numericamente avaliada. Isso também é útil se Maxima é usado para gerar programas para rodar em Fortran. Veja também `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` (*f*(*x*), *x*, *a*, *b*) [Função]
`find_root` (*f*, *a*, *b*) [Função]

Encontra a raíz da função *f* com a variável *x* percorrendo o intervalo `[a, b]`. A função deve ter um sinal diferente em cada ponto final. Se essa condição não for alcançada, a action of the function is governed by `find_root_error`. If `find_root_error` is `true` then an error occurs, otherwise the value of `find_root_error` is returned (thus for plotting `find_root_error` might be set to 0.0). De outra forma (dado que Maxima pode avaliar o primeiro argumento no intervalo especificado, e que o intervalo é contínuo) `find_root` é garantido vir para cima com a raíz (ou um deles se existir mais que uma raíz). A precisão de `find_root` é governada por `intpolabs` e `intpolrel` os quais devem ser números em ponto flutuante não negativos. `find_root` encerrará quando o primeiro argumento avaliar para alguma coisa menor que ou

igual a `intpolabs` ou se sucessivas aproximações da raíz diferirem por não mais que `intpolrel * <um dos aproximandos>`. O valor padrão de `intpolabs` e `intpolrel` são 0.0 de forma que `find_root` pega como boa uma resposta como for possível com a precisão aritmética simples que tivermos. O primeiro argumento pode ser uma equação. A ordem dos dois últimos argumentos é irrelevante. Dessa forma

```
find_root (sin(x) = x/2, x, %pi, 0.1);
```

é equivalente a

```
find_root (sin(x) = x/2, x, 0.1, %pi);
```

O método usado é uma busca binária no intervalo especificado pelos últimos dois argumentos. Quando o resultado da busca for encontrado a função é fechada o suficiente para ser linear, isso inicia usando interpolação linear.

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
```

`find_root_abs` [Variável de opção]

Valor por omissão: 0.0

`find_root_abs` é a precisão do comando `find_root`. A precisão é governada por `find_root_abs` e `find_root_rel` que devem ser números não negativos em ponto flutuante. `find_root` terminará quando o primeiro argumento avaliar para alguma coisa menor que ou igual a `find_root_abs` ou se sucessivos aproximandos para a raíz diferirem por não mais que `find_root_rel * <um dos aproximandos>`. Os valores padrão de `find_root_abs` e `find_root_rel` são 0.0 de forma que `find_root` tome como boa uma resposta que for possível com a precisão aritmética simples que tivermos.

`find_root_error` [Variável de opção]

Valor por omissão: `true`

`find_root_error` governa o comportamento de `find_root`. Quando `find_root` for chamada, ela determina se a função a ser resolvida satisfaz ou não a condição que os valores da função nos pontos finais do intervalo de interpolação são opostos em sinal. Se eles forem de sinais opostos, a interpolação prossegue. Se eles forem de mesmo sinal, e `find_root_error` for `true`, então um erro é sinalizado. Se eles forem de mesmo sinal e `find_root_error` não for `true`, o valor de `find_root_error` é retornado. Dessa forma para montagem de gráfico, `find_root_error` pode ser escolhida para 0.0.

`find_root_rel` [Variável de opção]

Valor por omissão: 0.0

`find_root_rel` é a precisão do comando `find_root` e é governada por `find_root_abs` e `find_root_rel` que devem ser números não negativos em ponto flutuante. `find_root` terminará quando o primeiro argumento avaliar para alguma coisa menor que ou igual a `find_root_abs` ou se sucessivos aproximandos para a raíz diferirem de não mais que `find_root_rel * <um dos aproximandos>`. Os valores padrão de `find_root_labs` e `find_root_rel` é 0.0 de forma que `find_root` toma como boa uma resposta que for possível com a precisão aritmética simples que tivermos.

`newton` (*expr*, *x*, *x_0*, *eps*) [Função]

Retorna uma solução aproximada de *expr* = 0 através do método de Newton, considerando *expr* como sendo uma função de uma variável, *x*. A busca pela solução começa com *x* = *x_0* e prossegue até `abs(`*expr*`) <` *eps* (com *expr* avaliada para o valor corrente de *x*).

`newton` permite que variáveis indefinidas apareçam em *expr*, contanto que o teste de terminação `abs(`*expr*`) <` *eps* avalie para `true` ou `false`. Dessa forma não é necessário que *expr* avalie para um número.

`load("newton1")` chama essa função.

Veja também `realroots`, `allroots`, `find_root`, e `mnewton`.

Exemplos:

```
(%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
```

23.4 Definições para Séries de Fourier

`equalp` (*x*, *y*) [Função]

Retorna `true` se `equal (x, y)` de outra forma `false` (não fornece uma mensagem de erro como `equal (x, y)` poderia fazer nesse caso).

`remfun` (*f*, *expr*) [Função]

`remfun` (*f*, *expr*, *x*) [Função]

`remfun (`*f*`, `*expr*`)` substitue todas as ocorrências de *f* `(`*arg*`)` por *arg* em *expr*.

`remfun (`*f*`, `*expr*`, `*x*`)` substitue todas as ocorrências de *f* `(`*arg*`)` por *arg* em *expr* somente se *arg* contiver a variável *x*.

`funp` (*f*, *expr*) [Função]
`funp` (*f*, *expr*, *x*) [Função]

`funp (f, expr)` retorna `true` se *expr* contém a função *f*.

`funp (f, expr, x)` retorna `true` se *expr* contém a função *f* e a variável *x* em algum lugar no argumento de uma das instâncias de *f*.

`absint` (*f*, *x*, *halfplane*) [Função]
`absint` (*f*, *x*) [Função]
`absint` (*f*, *x*, *a*, *b*) [Função]

`absint (f, x, halfplane)` retorna a integral indefinida de *f* com relação a *x* no dado semi-plano (`pos`, `neg`, ou `both`). *f* pode conter expressões da forma `abs (x)`, `abs (sin (x))`, `abs (a) * exp (-abs (b) * abs (x))`.

`absint (f, x)` é equivalente a `absint (f, x, pos)`.

`absint (f, x, a, b)` retorna a integral definida de *f* com relação a *x* de *a* até *b*. *f* pode incluir valores absolutos.

`fourier` (*f*, *x*, *p*) [Função]

Retorna uma lista de coeficientes de Fourier de `f(x)` definidos sobre o intervalo `[-p, p]`.

`foursimp` (*l*) [Função]

Simplifica `sin (n %pi)` para 0 se `sinnpiflag` for `true` e `cos (n %pi)` para `(-1)^n` se `cosnpiflag` for `true`.

`sinnpiflag` [Variável de opção]

Valor por omissão: `true`

Veja `foursimp`.

`cosnpiflag` [Variável de opção]

Valor por omissão: `true`

Veja `foursimp`.

`fourexpand` (*l*, *x*, *p*, *limit*) [Função]

Constrói e retorna a série de Fourier partindo da lista de coeficientes de Fourier *l* até (up through) *limit* termos (*limit* pode ser `inf`). *x* e *p* possuem o mesmo significado que em `fourier`.

`fourcos` (*f*, *x*, *p*) [Função]

Retorna os coeficientes do co-seno de Fourier para `f(x)` definida sobre `[0, %pi]`.

`foursin` (*f*, *x*, *p*) [Função]

Retorna os coeficientes do seno de Fourier para `f(x)` definida sobre `[0, p]`.

`totalfourier` (*f*, *x*, *p*) [Função]

Retorna `fourexpand (foursimp (fourier (f, x, p)), x, p, 'inf)`.

`fourint` (*f*, *x*) [Função]

Constrói e retorna uma lista de coeficientes de integral de Fourier de `f(x)` definida sobre `[minf, inf]`.

`fourintcos` (*f*, *x*) [Função]

Retorna os coeficientes da integral do co-seno de Fourier para `f(x)` on `[0, inf]`.

`fourintsin` (*f*, *x*) [Função]

Retorna os coeficientes da integral do seno de Fourier para `f(x)` on `[0, inf]`.

24 Arrays

24.1 Definições para Arrays

`array` (*name*, *dim_1*, ..., *dim_n*) [Função]
`array` (*name*, *type*, *dim_1*, ..., *dim_n*) [Função]
`array` ([*nome_1*, ..., *nome_m*], *dim_1*, ..., *dim_n*) [Função]

Cria um array n-dimensional. n pode ser menor ou igual a 5. Os subscritos para a i'ésima dimensão são inteiros no intervalo de 0 a *dim_i*.

`array (name, dim_1, ..., dim_n)` cria um array genérico.

`array (name, type, dim_1, ..., dim_n)` cria um array, com elementos de um tipo especificado. *type* pode ser `fixnum` para inteiros de tamanho limitado ou `flonum` para números em ponto flutuante.

`array ([nome_1, ..., nome_m], dim_1, ..., dim_n)` cria m arrays, todos da mesma dimensão.

Se o utilizador atribui a uma variável subscrita antes de declarar o array correspondente, um array não declarado é criado. Arrays não declarados, também conhecidos como array desordenado (porque o codigo desordenado termina nos subscritos), são mais gerais que arrays declarados. O utilizador não declara seu tamanho máximo, e ele cresce dinamicamente e desordenadamente à medida que são atribuídos valores a mais elementos. Os subscritos de um array não declarado não precisam sempre ser números. Todavia, excepto para um array um tanto quanto esparso, é provavelmente mais eficiente declarar isso quando possível que deixar não declarado. A função `array` pode ser usada para transformar um array não declarado em um array declarado.

`arrayapply` (*A*, [*i_1*, ..., *i_n*]) [Função]

Avalia `A [i_1, ..., i_n]`, quando A for um array e *i_1*, ..., *i_n* são inteiros.

Ela é remanescente de `apply`, excepto o primeiro argumento que é um array ao invés de uma função.

`arrayinfo` (*A*) [Função]

Retorna informações sobre o array A. O argumento A pode ser um array declarado, uma array não declarado (que sofreu um hash), uma função de array, ou uma função que possui subscrito.

Para arrays declarados, `arrayinfo` retorna uma lista compreendendo o átomo `declared`, o n;umero de dimensões, e o tamanho de cada dimensão. Os elementos do array, ambos associados e não associados, são retornados por `listarray`.

Para arrays não declarados (arrays que sofreram um hash), `arrayinfo` retorna uma lista compreendendo o átomo `hashed`, o número de subscritos, e os subscritos de de todo elemento que tiver um valor. Os valores são retornados por meio de `listarray`.

Para funções de array, `arrayinfo` retretorna uma lista compreendendo o átomo `hashed`, o número de subscritos, e quaisquer valores de subscritos para os quais exista valores funcionais armazenados. Os valores funcionais armazenados são retornados através de `listarray`.

Para funções que possuem subscritos, `arrayinfo` retorna uma lista compreendendo o átomo `hashed`, o número de subscritos, e qualquer valores subscritos para os quais existe uma expressões lambda. As expressões lambda são retornadas por `listarray`.

Examples:

`arrayinfo` e `listarray` aplicado a um array declarado.

```
(%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` e `listarray` aplicado a um array não declarado (no qual foi aplicado um hash).

```
(%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` e `listarray` aplicado a uma função de array.

```
(%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` e `listarray` aplicadas a funções com subscritos.

```
(%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*]) [Função]

Retorna a expressão *A*[*i_1*, ..., *i_n*]. O resultado é uma referência a um array não avaliado.

`arraymake` é remanicência de `funmake`, excepto o valor retornado é um array de referência não avaliado ao invés de uma chamada de função não avaliada.

Exemplos:

```
(%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` [Variável de sistema]

Valor por omissão: `[]`

`arrays` é uma lista dos arrays que tiverem sido alocados. Essa lista compreende arrays declarados através de `array`, arrays desordenados (hashed) construídos através de definição implícita (atribuindo alguma coisa a um elemento de array), e funções de array definidas por meio de `:=` e `define`. Arrays definidos por meio de `make_array` não estão incluídos.

Veja também `array`, `arrayapply`, `arrayinfo`, `arraymake`, `fillarray`, `listarray`, e `rearray`.

Exemplos:

```
(%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*) [Função]

Transforma a expressão *expr* dando a cada somatório e a cada produto um único índice. Isso dá a `changevar` grande precisão quando se está trabalhando com somatórios e produtos. A forma do único índice é *`jnumber`*. A quantidade *number* é determindad por referência a `gensumnum`, que pode ser alterada pelo utilizador. Por exemplo, `gensumnum:0$` reseta isso.

`fillarray` (*A*, *B*) [Função]

Preenche o array *A* com *B*, que é uma lista ou um array.

Se um tipo específico for declarado para A no momento de sua criação, A somente porde ser preenchido com elementos do tipo especificado; Constitui um erro alguma tentativa feita para copiar um um elemento de um tipo diferente.

Se as dimensões dos arrays A e B forem diferents, A é preenchido no ordem de maior fileira. Se não existirem elementos livres em B o último elemento é usado para preencher todo o resto de A. Se existirem muitos , esses restantes seram ignorados.

`fillarray` retorna esse primeiro argumento.

Exemplos:

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]
```

Quando existirem poucos elementos para preencher o array, o último elemento é repetido. Quando houverem muitos elementos, os elementos extras são ignorados.

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

Arrays multi-dimensionais são preenchidos em ordem de maior fileira.

```
(%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)` [Função]

Retorna uma lista dos elementos do array A. O argumento A pode ser um array declarado, um array não declarado (desordenado - hashed), uma função de array, ou uma função com subscritos.

Elementos são listados em ordem de linha maior. Isto é, elementos são ordenados conforme o primeiro índice, en seguida conforme o segundo índice, e assim sucessivamente. A sequuência de ordenação por meio dos valores dos índices é a mesma ordem estabelecida por meio de `orderless`.

Para arrays não declarados , funções de arrays, e funções com subscritos, os elementos correspondem aos valores de índice retornados através de `arrayinfo`.

Elemetos não associados de arrays genéricos declarados (isto é, não `fixnum` e não `flonum`) são retornados como `#####`. Elementos não associados de arrays declarados `fixnum` ou `flonum` são retornados como 0 ou 0.0, respectivamente. Elementos não associados de arrays não declarados, funções de array, e funções subscritas não são retornados.

Exemplos:

`listarray` e `arrayinfo` aplicados a um array declarado.

```
(%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` e `arrayinfo` aplicadas a arrays não declarados (hashed - desordenados).

```
(%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` e `arrayinfo` aplicada a uma função de array.

```
(%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` e `arrayinfo` aplicadas a funções com subscritos.

```
(%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*) [Função]

Cria e retorna um array de Lisp. *type* pode ser `any`, `flonum`, `fixnum`, `hashed` ou `functional`. Existem n indices, e o i'enésimo indice está no intervalo de 0 a $dim\_i - 1$.

A vantagem de `make_array` sobre `array` é que o valor de retorno não tem um nome, e uma vez que um ponteiro a ele vai, ele irá também. Por exemplo, se `y: make_array (...)` então `y` aponta para um objecto que ocupa espaço, mas depois de `y: false`, `y` não mais aponta para aquele objecto, então o objecto pode ser descartado.

Exemplos:

```
(%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*) [Função]

Altera as dimenções de um array. O novo array será preenchido com os elementos do antigo em ordem da maior linha. Se o array antigo era muito pequeno, os elementos restantes serão preenchidos com `false`, `0.0` ou `0`, dependendo do tipo do array. O tipo do array não pode ser alterado.

`remarray` (*A_1*, ..., *A_n*) [Função]

`remarray` (*all*) [Função]

Remove arrays e funções associadas a arrays e libera o espaço ocupado. Os argumentos podem ser arrays declarados, arrays não declarados (dsordenados - hashed), funções de array functions, e funções com subscritos.

`remarray (all)` remove todos os ítens na lista global `arrays`.

Isso pode ser necessário para usar essa função se isso é desejado para redefinir os valores em um array desordenado.

`remarray` retorna a lista dos arrays removidos.

`subvar` (*x*, *i*) [Função]

Avalia a expressão subscrita `x[i]`.

`subvar` avalia seus argumentos.

`arraymake (x, [i]` constrói a expressão `x[i]`, mas não a avalia.

Exemplos:

```
(%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` [Variável de pção]

- Se `true` somente dois tipos de arrays são reconhecidos.

1) O array art-q (t no Lisp Comum) que pode ter muitas dimensões indexadas por inteiros, e pode aceitar qualquer objecto do Lisp ou do Maxima como uma entrada. Para construir assim um array, insira `a:make_array(any,3,4);` então `a` terá como valor, um array com doze posições, e o índice é baseado em zero.

2) O array Hash_table que é o tipo padrão de array criado se um faz `b[x+1]:y^2` (e `b` não é ainda um array, uma lista, ou uma matriz – se isso ou um desses ocorrer um erro pode ser causado desde `x+1` não poderá ser um subscrito válido para um array art-q, uma lista ou uma matriz). Esses índices (também conhecidos como chaves) podem ser quaisquer objectos. Isso somente pega uma chave por vez a cada vez (`b[x+1,u]:y` ignorará o `u`). A referência termina em `b[x+1] ==> y^2`. Certamente a chave poe ser uma lista , e.g. `b[[x+1,u]]:y` poderá ser válido. Isso é incompatível com os arrays antigos do Maxima, mas poupa recursos.

Uma vantagem de armazenar os arrays como valores de símbolos é que as convenções usuais sobre variáveis locais de uma função aplicam-se a arrays também. O tipo Hash_table também usa menos recursos e é mais eficiente que o velho tipo hashar do Maxima. Para obter comportamento consistente em códigos traduzidos e compilados posicione `translate_fast_arrays` para ser `true`.

25 Matrizes e Álgebra Linear

25.1 Introdução a Matrizes e Álgebra Linear

25.1.1 Ponto

O operador `.` representa multiplicação não comutativa e produto escalar. Quando os operandos são matrizes 1-coluna ou 1-linha `a` e `b`, a expresão `a.b` é equivalente a `sum (a[i]*b[i], i, 1, length(a))`. Se `a` e `b` não são complexos, isso é o produto escalar, também chamado produto interno ou produto do ponto, de `a` e `b`. O produto escalar é definido como `conjugate(a).b` quando `a` e `b` são complexos; `innerproduct` no pacote `eigen` fornece o produto escalar complexo.

Quando os operandos são matrizes mais gerais, o produto é a matriz produto `a` e `b`. O número de linhas de `b` deve ser igual ao número de colunas de `a`, e o resultado tem número de linhas igual ao número de linhas de `a` e número de colunas igual ao número de colunas de `b`.

Para distinguir `.` como um operador aritmético do ponto decimal em um número em ponto flutuante, pode ser necessário deixar espaços em cada lado. Por exemplo, `5.e3` é `5000.0` mas `5 . e3` é `5` vezes `e3`.

Existem muitos sinalizadores que governam a simplificação de expresões envolvendo `.`, a saber `dot`, `dot0nscsimp`, `dot0simp`, `dot1simp`, `dotassoc`, `dotconstrules`, `dotdistrib`, `dotexptsimp`, `dotident`, e `dotscrules`.

25.1.2 Vetores

`vect` é um pacote de funções para análise vectorial. `load ("vect")` chama esse pacote, e `demo ("vect")` permite visualizar uma demonstração.

O pacote de análise vectorial pode combinar e simplificar expresões simbólicas incluindo produtos dos pontos e productos dos x, juntamente com o gradiente, divergencia, torção, e operadores Laplacianos. A distribuição desses operadores sobre adições ou produtos é governada por muitos sinalizadores, como são várias outras expansões, incluindo expansão dentro de componentes em qualquer sistema de coordenadas ortogonais. Existem também funções para derivar o escalar ou vector potencial de um campo.

O pacote `vect` contém essas funções: `vectorsimp`, `scalefactors`, `express`, `potential`, e `vectorpotential`.

Atenção: o pacote `vect` declara o operador ponto `.` como sendo um operador comutativo.

25.1.3 auto

O pacote `eigen` contém muitas funções devotadas para a computação simbólica de autovalores e autovectores. Maxima chama o pacote automaticamente se uma das funções `eigenvalues` ou `eigenvectors` é invocada. O pacote pode ser chamado explicitamente com `load ("eigen")`.

`demo ("eigen")` mostra uma demonstração das compatibilidades desse pacote. `batch ("eigen")` executa a mesma demonstração, mas sem lembretes de utilizador entre sucessivas computações.

As funções no pacote `eigen` são `innerproduct`, `unitvector`, `columnvector`, `gramschmidt`, `eigenvalues`, `eigenvectors`, `uniteigenvectors`, e `similaritytransform`.

25.2 Definições para Matrizes e Álgebra Linear

`addcol` (*M*, *list_1*, ..., *list_n*) [Função]
Anexa a(s) coluna(s) dadas por uma ou mais listas (ou matrizes) sobre a matriz *M*.

`addrow` (*M*, *list_1*, ..., *list_n*) [Função]
Anexa a(s) linha(s) dadas por uma ou mais listas (ou matrizes) sobre a matriz *M*.

`adjoint` (*M*) [Função]
Retorna a matriz adjunta da matriz *M*. A matriz adjunta é a transposta da matriz dos cofactores de *M*.

`augcoefmatrix` ([*eqn_1*, ..., *eqn_m*], [*x_1*, ..., *x_n*]) [Função]
Retorna a matriz dos coeficientes aumentada para as variáveis *x_1*, ..., *x_n* do sistema de equações lineares *eqn_1*, ..., *eqn_m*. Essa é a matriz dos coeficientes com uma coluna anexada para os termos independentes em cada equação (i.e., esses termos não dependem de *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*) [Função]
Retorna um polinómio característico para a matriz *M* em relação à variável *x*. Que é, `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*]) [Função]

Retorna a matriz dos coeficientes para as variáveis *x_1*, ..., *x_n* do sistema de equações lineares *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*) [Função]

Reorna a *i*'ésima coluna da matriz *M*. O valor de retorno é uma matriz.

`columnvector` (*L*) [Função]

`covect` (*L*) [Função]

Retorna uma matriz de uma coluna e `length` (*L*) linhas, contendo os elementos da lista *L*.

`covect` é um sinônimo para `columnvector`.

`load ("eigen")` chama essa função.

Isso é útil se quiser usar partes das saídas das funções nesse pacote em cálculos matriciais.

Exemplo:

```
(%i1) load ("eigen")$
Warning - you are redefining the Macsyma function autovalores
Warning - you are redefining the Macsyma function autovectores
(%i2) columnvector ([aa, bb, cc, dd]);
                                [ aa ]
                                [    ]
                                [ bb ]
(%o2)                           [    ]
                                [ cc ]
                                [    ]
                                [ dd ]
```

`conjugate` (*x*) [Função]

Retorna o conjugado complexo de *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*) [Função]

Retorna uma cópia da matriz *M*. Esse é o único para fazer uma copia separada copiando *M* elemento a elemento.

Note que uma atribuição de uma matriz para outra, como em `m2: m1`, não copia `m1`. Uma atribuição `m2 [i,j]: x` ou `setelmx (x, i, j, m2` também modifica `m1 [i,j]`. criando uma cópia com `copymatrix` e então usando atribução cria uma separada e modificada cópia.

`determinant` (*M*) [Função]

Calcula o determinante de *M* por um método similar à eliminação de Gauss.

A forma do resultado depende da escolha do comutador `ratmx`.

Existe uma rotina especial para calcular determinantes esparsos que é chamada quando os comutadores `ratmx` e `sparse` são ambos `true`.

`detout` [Variável]

Valor por omissão: `false`

Quando `detout` é `true`, o determinante de uma matriz cuja inversa é calculada é factorado fora da inversa.

Para esse comutador ter efeito `doallmxops` e `doscmxops` deveram ambos serem `false` (veja suas transcrições). Alternativamente esses comutadores podem ser dados para `ev` o que faz com que os outros dois sejam escolhidos correctamente.

Exemplo:

```
(%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*`)` [Função]

Retorna uma matriz diagonal de tamanho *n* por *n* com os elementos da diagonal todos iguais a *x*. `diagmatrix (`*n*`, 1)` retorna uma matriz identidade (o mesmo que `ident (`*n*`)`).

n deve avaliar para um inteiro, de outra forma `diagmatrix` reclama com uma mensagem de erro.

x pode ser qualquer tipo de expresão, incluindo outra matriz. Se *x* é uma matriz, isso não é copiado; todos os elementos da diagonal referem-se à mesma instância, *x*.

`doallmxops` [Variável]

Valor por omissão: `true`

Quando `doallmxops` é `true`, todas as operações relacionadas a matrizes são realizadas. Quando isso é `false` então a escolha de comutadores individuais `dot` governam quais operações são executadas.

`domxexpt` [Variável]

Valor por omissão: `true`

Quando `domxexpt` é `true`, uma matriz exponencial, `exp (`*M*`)` onde *M* é a matriz, é interpretada como uma matriz com elementos `[i,j` iguais a `exp (m[i,j])`. de outra forma `exp (`*M*`)` avalia para `exp (`*ev(M)*`)`.

`domxexpt` afecta todas as expresões da forma *`base^expoente`* onde *base* é uma expresão assumida escalar ou constante, e *expoente* é uma lista ou matriz.

Exemplo:

```
(%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` [Variável de opção]

Valor por omissão: `true`

Quando `domxmxops` é `true`, todas as operações matriz-matriz ou matriz-lista são realizadas (mas não operações escalar-matriz); se esse comutador é `false` tais operações não são.

`domxnctimes` [Variável de opção]

Valor por omissão: `false`

Quando `domxnctimes` é `true`, produtos não comutativos de matrizes são realizados.

`dontfactor` [Variável de opção]

Valor por omissão: `[]`

`dontfactor` pode ser escolhido para uma lista de variáveis em relação a qual factoração não é para ocorrer. (A lista é inicialmente vazia.) Factoração também não pegará lugares com relação a quaisquer variáveis que são menos importantes, conforme a hierarquía de variável assumida para a forma expresão racional canónica (CRE), que essas na lista `dontfactor`.

`doscmxops` [Variável de opção]

Valor por omissão: `false`

Quando `doscmxops` é `true`, operações escalar-matriz são realizadas.

`doscmxplus` [Variável de opção]

Valor por omissão: `false`

Quando `doscmxplus` é `true`, operações escalar-matriz retornam uma matriz resultado. Esse comutador não é subsomado sob `doallmxops`.

`dot0nscsimp` [Variável de opção]

Valor por omissão: `true`

Quando `dot0nscsimp` é `true`, um produto não comutativo de zero e um termo não escalar é simplificado para um produto comutativo.

`dot0simp` [Variável de opção]

Valor por omissão: `true`

Quando `dot0simp` é `true`, um produto não comutativo de zero e um termo escalar é simplificado para um produto não comutativo.

`dot1simp` [Variável de opção]

Valor por omissão: `true`

Quando `dot1simp` é `true`, um produto não comutativo de um e outro termo é simplificado para um produto comutativo.

`dotassoc` [Variável de opção]

Valor por omissão: `true`

Quando `dotassoc` é `true`, uma expresão `(A.B).C` simplifica para `A.(B.C)`.

`dotconstrules` [Variável de opção]

Valor por omissão: `true`

Quando `dotconstrules` é `true`, um produto não comutativo de uma constante e outro termo é simplificado para um produto comutativo. Ativando esse sinalizador efectivamente activamos `dot0simp`, `dot0nscsimp`, e `dot1simp` também.

`dotdistrib` [Variável de opção]

Valor por omissão: `false`

Quando `dotdistrib` é `true`, uma expresão `A.(B + C)` simplifica para `A.B + A.C`.

`dotexptsimp` [Variável de opção]

Valor por omissão: `true`

Quando `dotexptsimp` é `true`, uma expresão `A.A` simplifica para `A^^2`.

`dotident` [Variável de opção]

Valor por omissão: 1

`dotident` é o valor retornado por `X^^0`.

`dotscrules` [Variável de opção]

Valor por omissão: `false`

Quando `dotscrules` é `true`, uma expresão `A.SC` ou `SC.A` simplifica para `SC*A` e `A.(SC*B)` simplifica para `SC*(A.B)`.

`echelon` (*M*) [Função]

Retorna a forma escalonada da matriz *M*, como produzido através da eliminação de Gauss. A forma escalonada é calculada de *M* por operações elementares de linha tais que o primeiro elemento não zero em cada linha na matriz resultante seja o número um e os elementos da coluna abaixo do primeiro número um em cada linha sejam todos zero.

`triangularize` também realiza eliminação de Gaussian, mas não normaliza o elemento líder não nulo em cada linha.

`lu_factor` e `cholesky` são outras funções que retornam matrizes triangularizadas.

```
(%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*) [Função]
`eivals` (*M*) [Função]

Retorna uma lista de duas listas contendo os autovalores da matriz *M*. A primeira sublista do valor de retorno é a lista de autovalores da matriz, e a segunda sublista é a lista de multiplicidade dos autovalores na ordem correspondente.

`eivals` é um sinônimo de `eigenvalues`.

`eigenvalues` chama a função `solve` para achar as raízes do polinómio característico da matriz. Algumas vezes `solve` pode não estar habilitado a achar as raízes do polinómio; nesse caso algumas outras funções nesse pacote (except `innerproduct`, `unitvector`, `columnvector` e `gramschmidt`) não irão trabalhar.

Em alguns casos os autovalores achados por `solve` podem ser expresões complicadas. (Isso pode acontecer quando `solve` retorna uma expresão real não trivial para um autovalor que é sabidamente real.) Isso pode ser possível para simplificar os autovalores usando algumas outras funções.

O pacote `eigen.mac` é chamado automaticamente quando `eigenvalues` ou `eigenvectors` é referenciado. Se `eigen.mac` não tiver sido ainda chamado, `load ("eigen")` chama-o. Após ser chamado, todas as funções e variáveis no pacote estarão disponíveis.

`eigenvectors` (*M*) [Função]
`eivects` (*M*) [Função]

pegam uma matriz *M* como seu argumento e retorna uma lista de listas cuja primeira sublista é a saída de `eigenvalues` e as outras sublistas são os autovectores da matriz correspondente para esses autovalores respectivamente.

`eivects` é um sinônimo para `eigenvectors`.

O pacote `eigen.mac` é chamado automaticamente quando `eigenvalues` ou `eigenvectors` é referenciado. Se `eigen.mac` não tiver sido ainda chamado, `load ("eigen")` chama-o. Após ser chamado, todas as funções e variáveis no pacote estarão disponíveis.

Os sinalizadores que afectam essa função são:

`nondiagonalizable` é escolhido para `true` ou `false` dependendo de se a matriz é não diagonalizável ou diagonalizável após o retorno de `eigenvectors`.

`hermitianmatrix` quando `true`, faz com que os autovectores degenerados da matriz Hermitiana sejam ortogonalizados usando o algoritmo de Gram-Schmidt.

`knowneigvals` quando `true` faz com que o pacote `eigen` assumir que os autovalores da matriz são conhecidos para o utilizador e armazenados sob o nome global `listeigvals`. `listeigvals` poderá ser escolhido para uma lista similar à saída de `eigenvalues`.

A função `algsys` é usada aqui para resolver em relação aos autovectores. Algumas vezes se os autovalores estão ausêntes, `algsys` pode não estar habilitado a achar uma solução. Em alguns casos, isso pode ser possível para simplificar os autovalores por primeiro achando e então usando o comando `eigenvalues` e então usando outras funções para reduzir os autovalores a alguma coisa mais simples. Continuando a simplificação, `eigenvectors` pode ser chamada novamente com o sinalizador `knowneigvals` escolhido para `true`.

`ematrix` (*m*, *n*, *x*, *i*, *j*) [Função]

Retorna uma matriz *m* por *n*, todos os elementos da qual são zero excepto para o elemento `[i, j]` que é *x*.

`entermatrix` (*m*, *n*) [Função]

Retorna uma matriz *m* por *n*, lendo os elementos interativamente.

Se *n* é igual a *m*, Maxima pergunta pelo tipo de matriz (diagonal, simétrica, anti-simétrica, ou genérica) e por cada elemento. Cada resposta é terminada por um ponto e vírgula `;` ou sinal de dólar `$`.

Se *n* não é igual a *m*, Maxima pergunta por cada elemento.

Os elementos podem ser quaisquer expressões, que são avaliadas. `entermatrix` avalia seus argumentos.

```
(%i1) n: 3$
(%i2) m: entermatrix (n, n)$

Is the matriz  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)$

Matriz 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*) [Função]
`genmatrix` (a, *i_2*, *j_2*, *i_1*) [Função]
`genmatrix` (a, *i_2*, *j_2*) [Função]

Retorna uma matriz gerada de *a*, pegando o elemento `a[i_1,j_1]` como o elemento do canto superior esquerdo e `a[i_2,j_2]` como o elemento do canto inferior directo da matriz. Aqui *a* é um array declarado (criado através de `array` mas não por meio de `make_array`) ou um array não declarado, ou uma função array, ou uma expressão lambda de dois argumentos. (Uma funçãO array é criado como outras funções com `:=` ou `define`, mas os argumentos são colocados entre colchêtes em lugar de parêntesis.)

Se *j_1* é omitido, isso é assumido ser igual a *i_1*. Se ambos *j_1* e *i_1* são omitidos, ambos são assumidos iguais a 1.

Se um elemento seleccionado `i,j` de um array for indefinido, a matriz conterá um elemento simbólico `a[i,j]`.

Exemplos:

```
(%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*) [Função]
`gschmit` (*x*) [Função]

Realiza o algoritmo de ortonalização de Gram-Schmidt sobre *x*, seja ela uma matriz ou uma lista de listas. *x* não é modificado por `gramschmidt`.

Se x é uma matriz, o algoritmo é aplicado para as linhas de x. Se x é uma lista de listas, o algoritmo é aplicado às sublistas, que devem ter igual números de elementos. Nos dois casos, o valor de retorno é uma lista de listas, as sublistas das listas são ortogonais e gera o mesmo spaço que x. Se a dimensão do conjunto gerador de x é menor que o número de linhas ou sublistas, algumas sublistas do valor de retorno são zero.

`factor` é chamada a cada estágio do algoritmo para simplificar resultados intermédios. Como uma consequência, o valor de retorno pode conter inteiros factorados.

`gschmit` (nota ortográfica) é um sinônimo para `gramschmidt`.

`load ("eigen")` chama essa função.

Exemplo:

```
(%i1) load ("eigen")$
Warning - you are redefining the Macsyma function autovalores
Warning - you are redefining the Macsyma function autovectores
(%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) i: innerproduct$
(%i5) [i (y[1], y[2]), i (y[2], y[3]), i (y[3], y[1])];
(%o5)                       [0, 0, 0]
```

`ident` (*n*) [Função]

Retorna uma matriz identidade n por n.

`innerproduct` (*x*, *y*) [Função]

`inprod` (*x*, *y*) [Função]

Retorna o produto interno (também chamado produto escalar ou produto do ponto) de x e y, que são listas de igual comprimento, ou ambas matrizes 1-coluna ou 1-linha de igual comprimento. O valor de retorno é `conjugate (x) . y`, onde `.` é o operador de multiplicação não comutativa.

`load ("eigen")` chama essa função.

`inprod` é um sinônimo para `innerproduct`.

`invert` (*M*) [Função]

Retorna a inversa da matriz M. A inversa é calculada pelo método adjunto.

Isso permite a um utilizador calcular a inversa de uma matriz com entradas bfloat ou polinómios com coeficientes em ponto flutuante sem converter para a forma CRE.

Cofactores são calculados pela função `determinant`, então se `ratmx` é `false` a inversa é calculada sem mudar a representação dos elementos.

A implementação corrente é ineficiente para matrizes de alta ordem.

Quando `detout` é `true`, o determinante é factorado fora da inversa.

Os elementos da inversa não são automaticamente expandidos. Se M tem elementos polinomiais, melhor aparência de saída pode ser gerada por `expand (invert (m)), detout`. Se isso é desejável para ela divisão até pelo determinante pode ser excelente por `xthru (%)` ou alternativamente na unha por

```
expe (adjoint (m)) / expand (determinant (m))
invert (m) := adjoint (m) / determinant (m)
```

Veja `^^` (expoente não comutativo) para outro método de inverter uma matriz.

`lmxchar` [Variável de opção]

Valor por omissão: `[`

`lmxchar` é o caractere mostrado como o delimitador esquerdo de uma matriz. Veja também `rmxchar`.

Exemplo:

```
(%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*) [Função]

Retorna uma matriz retangular que tem as linhas *row_1*, ..., *row_n*. Cada linha é uma lista de expressões. Todas as linhas devem ter o mesmo comprimento.

As operações `+` (adição), `-` (subtração), `*` (multiplicação), e `/` (divisão), são realizadas elemento por elemento quando os operandos são duas matrizes, um escalar e uma matriz, ou uma matriz e um escalar. A operação `^` (exponenciação, equivalentemente `**`) é realizada elemento por elemento se os operandos são um escalar e uma matriz ou uma matriz e um escalar, mas não se os operandos forem duas matrizes. Todos as operações são normalmente realizadas de forma completa, incluindo `.` (multiplicação não comutativa).

Multiplicação de matrizes é representada pelo operador de multiplicação não comutativa `..` O correspondente operador de exponenciação não comutativa é `^^`. Para uma matriz `A`, `A.A` = `A^^2` e `A^^-1` é a inversa de A, se existir.

Existem comutadores para controlar a simplificação de expresões envolvendo operações escalar e matriz-lista. São eles `doallmxops`, `domxexpt domxmxops`, `doscmxops`, e `doscmxplus`.

Existem opções adicionais que são relacionadas a matrizes. São elas: `lmxchar`, `rmxchar`, `ratmx`, `listarith`, `detout`, `scalarmatrix`, e `sparse`.

Existe um número de funções que pegam matrizes como argumentos ou devolvem matrizes como valor de retorno. Veja `eigenvalues`, `eigenvectors`, `determinant`, `charpoly`, `genmatrix`, `addcol`, `addrow`, `copymatrix`, `transpose`, `echelon`, e `rank`.

Exemplos:

- Construção de matrizes de listas.

```
(%i1) x: matrix ([17, 3], [-8, 11]);
                             [ 17   3  ]
(%o1)                        [         ]
                             [ - 8  11 ]
(%i2) y: matrix ([%pi, %e], [a, b]);
                             [ %pi  %e ]
(%o2)                        [         ]
                             [  a   b  ]
```

- Adição, elemento por elemento.

```
(%i3) x + y;
                        [ %pi + 17  %e + 3 ]
(%o3)                   [                  ]
                        [  a - 8    b + 11 ]
```

- Subtração, elemento por elemento.

```
(%i4) x - y;
                        [ 17 - %pi  3 - %e ]
(%o4)                   [                  ]
                        [ - a - 8   11 - b ]
```

- Multiplicação, elemento por elemento.

```
(%i5) x * y;
                          [ 17 %pi  3 %e ]
(%o5)                     [              ]
                          [ - 8 a   11 b ]
```

- Divisão, elemento por elemento.

```
(%i6) x / y;
                          [ 17       - 1 ]
                          [ ---  3 %e    ]
                          [ %pi          ]
(%o6)                     [              ]
                          [   8    11    ]
                          [ - -    --    ]
                          [   a    b     ]
```

- Matriz para um expoente escalar, elemento por elemento.

```
(%i7) x ^ 3;
                           [ 4913    27  ]
(%o7)                      [             ]
                           [ - 512  1331 ]
```

- Base escalar para um expoente matriz, elemento por elemento.

```
(%i8) exp(y);
                          [   %pi    %e ]
                          [ %e     %e   ]
(%o8)                     [              ]
                          [    a     b  ]
                          [  %e    %e   ]
```

- Base matriz para um expoente matriz. Essa não é realizada elemento por elemento.

```
(%i9) x ^ y;
                                  [ %pi  %e ]
                                  [         ]
                                  [  a   b  ]
                       [ 17   3  ]
(%o9)                  [         ]
                       [ - 8  11 ]
```

- Multiplicação não comutativa de matrizes.

```
(%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   ]
```

- Exponenciação não comutativa de matrizes. Uma base escalar b para uma potência matriz M é realizada elemento por elemento e então `b^^m` é o mesmo que `b^m`.

```
(%i12) x ^^ 3;
                         [  3833   1719 ]
(%o12)                   [              ]
                         [ - 4584  395  ]
(%i13) %e ^^ y;
                          [   %pi    %e ]
                          [ %e     %e   ]
(%o13)                    [              ]
                          [    a     b  ]
                          [  %e    %e   ]
```

- A matriz elevada a um expoente -1 com exponenciação não comutativa é a matriz inversa, se existir.

```
(%i14) x ^^ -1;
                          [ 11      3  ]
                          [ ---  - --- ]
                          [ 211    211 ]
(%o14)                    [            ]
                          [  8    17   ]
```

```
                              [ ---   ---  ]
                              [ 211   211  ]
(%i15) x . (x ^^ -1);
                                 [ 1  0 ]
(%o15)                           [      ]
                                 [ 0  1 ]
```

`matrixmap` (*f*, *M*) [Função]

Retorna uma matriz com elemento `i,j` igual a *f*(*M*`[i,j]`).

Veja também `map`, `fullmap`, `fullmapl`, e `apply`.

`matrixp` (*expr*) [Função]

Retorna `true` se *expr* é uma matriz, de outra forma retorna `false`.

`matrix_element_add` [Variável de opção]

Valor por omissão: `+`

`matrix_element_add` é a operação invocada em lugar da adição em uma multiplicação de matrizes. A `matrix_element_add` pode ser atribuído qualquer operador n-ário (que é, uma função que manuseia qualquer número de argumentos). Os valores atribuídos podem ser o nome de um operador entre aspas duplas, o nome da função, ou uma expressão lambda.

Veja também `matrix_element_mult` e `matrix_element_transpose`.

Exemplo:

```
(%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^u  b^v  c^w  a^x b^y c^z ]
(%o5)                    [                           ]
                         [  u   v   w    x   y   z  ]
                         [ d^u  e^v  f^w  d^x e^y f^z ]
```

`matrix_element_mult` [Variável de opção]

Valor por omissão: `*`

`matrix_element_mult` é a operação invocada em lugar da multiplicação em uma multiplicação de matrizes. A `matrix_element_mult` pode ser atribuído qualquer operador binário. O valor atribuído pode ser o nome de um operador entre aspas duplas, o nome de uma função, ou uma expressão lambda.

O operador do ponto `.` é uma escolha útil em alguns contextos.

Veja também `matrix_element_add` e `matrix_element_transpose`.

Exemplo:

```
(%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` [Variável de opção]

Valor por omissão: `false`

`matrix_element_transpose` é a operação aplicada a cada elemento de uma matriz quando for uma transposta. A `matrix_element_mult` pode ser atribuído qualquer operador unário. O valor atribuído pode ser nome de um operador entre aspas duplas, o nome de uma função, ou uma expressão lambda.

Quando `matrix_element_transpose` for igual a `transpose`, a função `transpose` é aplicada a todo elemento. Quando `matrix_element_transpose` for igual a `nonscalars`, a função `transpose` é aplicada a todo elemento não escalar. Se algum elemento é um átomo, a opção `nonscalars` aplica `transpose` somente se o átomo for declarado não escalar, enquanto a opção `transpose` sempre aplica `transpose`.

O valor padrão, `false`, significa nenhuma operação é aplicada.

Veja também `matrix_element_add` e `matrix_element_mult`.

Exemplos:

```
(%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*) [Função]

Retorna o traço (que é, a soma dos elementos sobre a diagonal principal) da matriz quadrada *M*.

`mattrace` é chamada por `ncharpoly`, uma alternativa para `charpoly` do Maxima.

`load ("nchrpl")` chama essa função.

`minor` (*M*, *i*, *j*) [Função]

Retorna o *i*, *j* menor do elemento localizado na linha *i* coluna *j* da matriz *M*. Que é *M* com linha *i* e coluna *j* ambas removidas.

`ncexpt` (*a*, *b*) [Função]

Se uma expressão exponencial não comutativa é muito alta para ser mostrada como `a^^b` aparecerá como `ncexpt (a,b)`.

`ncexpt` não é o nome de uma função ou operador; o nome somente aparece em saídas, e não é reconhecido em entradas.

`ncharpoly` (*M*, *x*) [Função]

Retorna o polinómio característico da matriz *M* com relação a *x*. Essa é uma alternativa para `charpoly` do Maxima.

`ncharpoly` trabalha pelo cálculo dos traços das potências na dada matriz, que são sabidos serem iguais a somas de potências das raízes do polinómio característico. Para essas quantidade a função simétrica das raízes pode ser calculada, que nada mais são que os coeficientes do polinómio característico. `charpoly` trabalha formatando o determinante de *x* `* ident [n] - a`. Dessa forma `ncharpoly` é vencedor, por exemplo, no caso de largas e densas matrizes preencidas com inteiros, desde que isso evite inteiramente a aritmética polinomial.

`load ("nchrpl")` loads this file.

`newdet (M, n)` [Função]
Calcula o determinante de uma matriz ou array M pelo algoritmo da árvore menor de Johnson-Gentleman. O argumento n é a ordem; isso é opcional se M for uma matriz.

`nonscalar` [Declaração]
Faz átomos ser comportarem da mesma forma que uma lista ou matriz em relação ao operador do ponto.

`nonscalarp (expr)` [Função]
Retorna `true` se *expr* é um não escalar, i.e., isso contém átomos declarados como não escalares, listas, ou matrizes.

`permanent (M, n)` [Função]
Calcula o permanente da matriz M. Um permanente é como um determinante mas sem mudança de sinal.

`rank (M)` [Função]
Calcula o posto da matriz M. Que é, a ordem do mais largo determinante não singular de M.

rank pode retornar uma resposta ruim se não puder determinar que um elemento da matriz que é equivalente a zero é realmente isso.

`ratmx` [Variável de opção]
Valor por omissão: `false`

Quando `ratmx` é `false`, adição, subtração, e multiplicação para determinantes e matrizes são executados na representação dos elementos da matriz e fazem com que o resultado da inversão de matrizes seja esquerdo na representação geral.

Quando `ratmx` é `true`, as 4 operações mencionadas acima são executadas na forma CRE e o resultado da matriz inversa é dado na forma CRE. Note isso pode fazer com que os elementos sejam expandidos (dependendo da escolha de `ratfac`) o que pode não ser desejado sempre.

`row (M, i)` [Função]
retorna a i'ésima linha da matriz M. O valor de retorno é uma matriz.

`scalarmatrixp` [Variável de opção]
Valor por omissão: `true`

Quando `scalarmatrixp` é `true`, então sempre que uma matriz 1 x 1 é produzida como um resultado de cálculos o produto do ponto de matrizes é simplificado para um escalar, a saber o elemento solitário da matriz.

Quando `scalarmatrixp` é `all`, então todas as matrizes 1 x 1 serão simplificadas para escalares.

Quando `scalarmatrixp` é `false`, matrizes 1 x 1 não são simplificadas para escalares.

`scalefactors` (*coordinatetransform*) [Função]

Aqui coordinatetransform avalia para a forma [[expresão1, expresão2, ...], indeterminação1, indeterminação2, ...], onde indeterminação1, indeterminação2, etc. são as variáveis de coordenadas curvilíneas e onde a escolha de componentes cartesianas retangulares é dada em termos das coordenadas curvilíneas por [expresão1, expresão2, ...]. `coordinates` é escolhida para o vector [indeterminação1, indeterminação2,...], e `dimension` é escolhida para o comprimento desse vector. SF[1], SF[2], ..., SF[DIMENSION] são escohidos para factores de escala de coordenada, e `sfprod` é escohido para o produto desse factores de escala. Inicialmente, `coordinates` é [X, Y, Z], `dimension` é 3, e SF[1]=SF[2]=SF[3]=SFPROD=1, correspondendo a coordenadas Cartesianas retangulares 3-dimensional. Para expandir uma expresão dentro de componentes físicos no sistema de coordenadas corrente , existe uma função com uso da forma

`setelmx` (*x*, *i*, *j*, *M*) [Função]

Atribue *x* para o (*i*, *j*)'ésimo elemento da matriz *M*, e retorna a matriz alterada.

M `[`*i*`,` *j*`]:` *x* tem o mesmo efeito, mas retorna *x* em lugar de *M*.

`similaritytransform` (*M*) [Função]
`simtran` (*M*) [Função]

`similaritytransform` calcula uma transformação homotética da matriz `M`. Isso retorna uma lista que é a saída do comando `uniteigenvectors`. Em adição se o sinalizador `nondiagonalizable` é `false` duas matrizes globais `leftmatrix` e `rightmatrix` são calculadas. Essas matrizes possuem a propriedade de `leftmatrix` `.` *M* `.` `rightmatrix` é uma matriz diagonal com os autovalores de *M* sobre a diagonal. Se `nondiagonalizable` é `true` as matrizes esquerda e direita não são computadas.

Se o sinalizador `hermitianmatrix` é `true` então `leftmatrix` é o conjugado complexo da transposta de `rightmatrix`. De outra forma `leftmatrix` é a inversa de `rightmatrix`.

`rightmatrix` é a matriz cujas colunas são os autovectores unitários de *M*. Os outros sinalizadores (veja `eigenvalues` e `eigenvectors`) possuem o mesmo efeito desde que `similaritytransform` chama as outras funções no pacote com o objectivo de estar habilitado para a forma `rightmatrix`.

`load ("eigen")` chama essa função.

`simtran` é um sinônimo para `similaritytransform`.

`sparse` [Variável de opção]

Valor por omissão: `false`

Quando `sparse` é `true`, e se `ratmx` é `true`, então `determinant` usará rotinas especiais para calcular determinantes esparsos.

`submatrix` (*i_1*, ..., *i_m*, *M*, *j_1*, ..., *j_n*) [Função]
`submatrix` (*i_1*, ..., *i_m*, *M*) [Função]
`submatrix` (*M*, *j_1*, ..., *j_n*) [Função]

Retorna uma nova matriz formada pela matrix *M* com linhas *i_1*, ..., *i_m* excluídas, e colunas *j_1*, ..., *j_n* excluídas.

`transpose` (*M*) [Função]

Retorna a transposta de *M*.

Se *M* é uma matriz, o valor de retorno é outra matriz *N* tal que `N[i,j] = M[j,i]`.

Se *M* for uma lista, o valor de retorno é uma matrix *N* de `length (m)` linhas e 1 coluna, tal que `N[i,1] = M[i]`.

De outra forma *M* é um símbolo, e o valor de retorno é uma expressão substantiva `'transpose (M)`.

`triangularize` (*M*) [Função]

Retorna a maior forma triangular da matriz `M`, como produzido através da eliminação de Gauss. O valor de retorno é o mesmo que `echelon`, excepto que o o coeficiente lider não nulo em cada linha não é normalizado para 1.

`lu_factor` e `cholesky` são outras funções que retornam matrizes triangularizadas.

```
(%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*) [Função]
`ueivects` (*M*) [Função]

Calcula autovectores unitários da matriz *M*. O valor de retorno é uma lista de listas, a primeiro sublista é a saída do comando `eigenvalues`, e as outras sublistas são os autovectores unitários da matriz correspondente a esses autovalores respectivamente.

Os sinalizadores mencionados na descrição do comando `eigenvectors` possuem o mesmo efeito aqui também.

Quando `knowneigvects` é `true`, o pacote `eigen` assume que os autovectores da matriz são conhecidos para o utilizador são armazenados sob o nome global `listeigvects`. `listeigvects` pode ser ecolhido para uma lista similar à saída do comando `eigenvectors`.

Se `knowneigvects` é escolhido para `true` e a lista de autovectores é dada a escolha do sinalizador `nondiagonalizable` pode não estar correcta. Se esse é o caso por favor ecolha isso para o valor correcto. O autor assume que o utilizador sabe o que está fazendo e que não tentará diagonalizar uma matriz cujos autovectores não geram o mesmo espaço vectorial de dimensão apropriada.

`load ("eigen")` chama essa função.

`ueivects` é um sinônimo para `uniteigenvectors`.

`unitvector (x)` [Função]
`uvect (x)` [Função]

Retorna $x/norm(x)$; isso é um vector unitário na mesma direção que x.

`load ("eigen")` chama essa função.

`uvect` é um sinônimo para `unitvector`.

`vectorsimp (expr)` [Função]

Aplica simplificações e expansões conforme os seguintes sinalizadores globais:

`expandall`, `expanddot`, `expanddotplus`, `expandcross`, `expandcrossplus`, `expandcrosscross`, `expandgrad`, `expandgradplus`, `expandgradprod`, `expanddiv`, `expanddivplus`, `expanddivprod`, `expandcurl`, `expandcurlplus`, `expandcurlcurl`, `expandlaplacian`, `expandlaplacianplus`, e `expandlaplacianprod`.

Todos esses sinalizadores possuem valor padrão `false`. O sufixo `plus` refere-se a utilização aditivamente ou distribuitivamente. O sufixo `prod` refere-se a expansão para um operando que é qualquer tipo de produto.

`expandcrosscross`
: Simplifica $p\ (q\ r)$ para $(p.r) * q - (p.q) * r$.

`expandcurlcurl`
: Simplifica *curlcurlp* para *graddivp* + *divgradp*.

`expandlaplaciantodivgrad`
: Simplifica *laplacianp* para *divgradp*.

`expandcross`
: Habilita `expandcrossplus` e `expandcrosscross`.

`expandplus`
: Habilita `expanddotplus`, `expandcrossplus`, `expandgradplus`, `expanddivplus`, `expandcurlplus`, e `expandlaplacianplus`.

`expandprod`
: Habilita `expandgradprod`, `expanddivprod`, e `expandlaplacianprod`.

Esses sinalizadores foram todos declarados `evflag`.

`vect_cross` [Variável de opção]

Valor por omissão: `false`

Quando `vect_cross` é `true`, isso permite DIFF(X~Y,T) trabalhar onde ~ é definido em SHARE;VECT (onde VECT_CROSS é escolhido para `true`, de qualqeur modo.)

`zeromatrix (m, n)` [Função]

Retorna um matriz m por n, com todos os elementos sendo zero.

`[` [Símbolo especial]
`]` [Símbolo especial]

`[` e `]` marcam o omeço e o fim, respectivamente, de uma lista.

`[` e `]` também envolvem os subscritos de uma lista, array, array desordenado, ou função array.

Exemplos:

```
(%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 Funções Afins

26.1 Definições para Funções Afins

`fast_linsolve` (*[expr_1, ..., expr_m]*, *[x_1, ..., x_n]*) [Função]

Resolve equações lineares simultâneas *expr_1*, ..., *expr_m* para as variáveis *x_1*, ..., *x_n*. Cada *expr_i* pode ser uma equação ou uma expressão geral; se for dada como uma expressão geral, será tratada como uma equação na forma *expr_i* `= 0`.

O valor de retorno é uma lista de equações da forma `[x_1 = a_1, ..., x_n = a_n]` onde *a_1*, ..., *a_n* são todas livres de *x_1*, ..., *x_n*.

`fast_linsolve` é mais rápido que `linsolve` para sistemas de equações que são esparsas.

Para usar essa função escreva primeiramente `load("affine")`.

`grobner_basis` (*[expr_1, ..., expr_m]*) [Função]

Retorna uma base de Groebner para as equações *expr_1*, ..., *expr_m*. A funçã `polysimp` pode então ser usada para simplificar outras funções relativas às equações.

```
grobner_basis ([3*x^2+1, y*x])$

polysimp (y^2*x + x^3*9 + 2) ==> -3*x + 2
```

`polysimp(f)` produz 0 se e somente se *f* está no ideal gerado por *expr_1*, ..., *expr_m*, isto é, se e somente se *f* for uma combinação polinomial dos elementos de *expr_1*, ..., *expr_m*.

Para usar essa função escreva primeiramente `load("affine")`.

`set_up_dot_simplifications` (*eqns*, *check_through_degree*) [Função]

`set_up_dot_simplifications` (*eqns*) [Função]

As *eqns* são equações polinomiais em variáveis não comutativas. O valor de `current_variables` é uma lista de variáveis usadas para calcular graus. As equações podem ser homogéneas, de forma a que o procedimento termine.

Se tiver optado por sobreposição de simplificações em `dot_simplifications` acima do grau de *f*, então o seguinte é verdadeiro: `dotsimp (f)` retorna 0 se, e somente se, *f* está no ideal gerado pelas equações, i.e., se e somente se *f* for uma combinação polinomial dos elementos das equações.

O grau é aquele retornado por `nc_degree`. Isso por sua vez é influenciado pelos pesos das variáveis individuais.

Para usar essa função escreva primeiramente `load("affine")`.

`declare_weights` (*x_1*, *w_1*, ..., *x_n*, *w_n*) [Função]

Atribui pesos *w_1*, ..., *w_n* to *x_1*, ..., *x_n*, respectivamente. Esses são pesos usados em cálculos `nc_degree`.

Para usar essa função escreva primeiramente `load("affine")`.

`nc_degree` (*p*) [Função]

Retorna o grau de um polinómio não comutativo *p*. Veja `declare_weights`.

Para usar essa função escreva primeiramente `load("affine")`.

`dotsimp` (*f*) [Função]

Retorna 0 se e somente se f for um ideal gerado pelas equações, i.e., se e somente se f for uma combinação polinomial dos elementos das equações.

Para usar essa função escreva primeiramente `load("affine")`.

`fast_central_elements` ([*x_1*, ..., *x_n*], *n*) [Função]

Se `set_up_dot_simplifications` tiver sido feito previamente, ache o polinómio central nas variáveis *x_1*, ..., *x_n* no grau dado, *n*.

Por exemplo:

```
set_up_dot_simplifications ([y.x + x.y], 3);
fast_central_elements ([x, y], 2);
[y.y, x.x];
```

Para usar essa função escreva primeiramente `load("affine")`.

`check_overlaps` (*n*, *add_to_simps*) [Função]

Verifica as sobreposições através do grau n, garantindo que tem regras de simplificação suficientes em cada grau, para `dotsimp` trabalhar correctamente. Esse processo pode ser mais rápido se souber de antemão a dimensão do espaço de monómios. Se for de dimensão global finita, então `hilbert` pode ser usada. Se não conhece as dimensões monomiais, não especifique nenhum `rank_function`. Um terceiro argumento opcional, `reset`, `false` diz para não se incomodar em perguntar sobre reiniciar coisas.

Para usar essa função escreva primeiramente `load("affine")`.

`mono` ([*x_1*, ..., *x_n*], *n*) [Função]

Retorna a lista de monómios independentes relativamente à simplificação actual do grau n nas variáveis *x_1*, ..., *x_n*.

Para usar essa função escreva primeiramente `load("affine")`.

`monomial_dimensions` (*n*) [Função]

Calcula a série de Hilbert através do grau n para a algebra corrente.

Para usar essa função escreva primeiramente `load("affine")`.

`extract_linear_equations` ([*p_1*, ..., *p_n*], [*m_1*, ..., *m_n*]) [Função]

Faz uma lista dos coeficientes dos polinómios não comutativos *p_1*, ..., *p_n* dos monomios não comutatvos *m_1*, ..., *m_n*. Os coeficientes podem ser escalares. Use `list_nc_monomials` para construir a lista dos monómios.

Para usar essa função escreva primeiramente `load("affine")`.

`list_nc_monomials` ([*p_1*, ..., *p_n*]) [Função]

`list_nc_monomials` (*p*) [Função]

Retorna uma lista de monómios não comutativos que ocorrem em um polinómio p ou em uma lista de polinómios *p_1*, ..., *p_n*.

Para usar essa função escreva primeiramente `load("affine")`.

`all_dotsimp_denoms` [Variável de opção]

Valor por omissão: `false`

Quando `all_dotsimp_denoms` é uma lista, os denominadores encontrados por `dotsimp` são adicionados ao final da lista. `all_dotsimp_denoms` pode ser iniciado como uma lista vazia `[]` antes chamando `dotsimp`.

Por padrão, denominadores não são colectados por `dotsimp`.

27 itensor

27.1 Introdução a itensor

Maxima implementa a manipulação de tensores simbólicos d dois tipos distintos: manipulação de componentes de tensores (pacote `ctensor`) e manipulação de tensores indiciais (pacote `itensor`).

Note bem: Por favor veja a nota sobre 'nova notação de tensor' abaixo.

Manipulação de componentes de tensores significa que objectos do tipo tensor geométrico são representados como arrays ou matrizes. Operações com tensores tais com contração ou diferenciação covariante são realizadas sobre índices (que ocorrem exactamente duas vezes) repetidos com declarações `do`. Isto é, se executa explicitamente operações sobre as componentes apropriadas do tensor armazenadas em um array ou uma matriz.

Manipulação tensorial de índice é implementada através da representação de tensores como funções e suas covariantes, contravariantes e índices de derivação. Operações com tensores como contração ou diferenciação covariante são executadas através de manipulação dos índices em si mesmos em lugar das componentes para as quais eles correspondem.

Esses dois métodos aproximam-se do tratamento de processos diferenciais, algébricos e analíticos no contexto da geometria de Riemannian possuem várias vantagens e desvantagens as quais se revelam por si mesmas somente apesar da natureza particular e dificuldade dos problemas de utilizador. Todavia, se pode ter em mente as seguintes características das duas implementações:

As representações de tensores e de operações com tensores explicitamente em termos de seus componntes tornam o pacote `ctensor` fácil de usar. Especificação da métrica e o cálculo de tensores induzidos e invariantes é directo. Embora todas a capacidade de simplificação poderosa do Maxima está em manusear, uma métrica complexa com intrincada dependência funcional e de coordenadas pode facilmente conduzir a expressões cujo tamanho é excessivo e cuja estrutura está escondida. Adicionalmente, muitos cálculos envolvem expressões intermédias cujo crescimento fazem com que os programas terminem antes de serem completados. Através da experiência, um utilizador pode evitar muitas dessas dificuldade.

O motivo de caminhos especiais através dos quais tensores e operações de tensores são representados em termos de operações simbólicas sobre seus índices, expressões cujas representação de componentes podem ser não gerenciaveis da forma comum podem algumas vezes serem grandemente simplificadas através do uso das rotinas especiais para objectos simétricos em `itensor`. Nesse caminho a estrutura de uma expressão grande pode ser mais transparente. Por outro lado, o motivo da representação indicial especial em `itensor`, faz com que em alguns casos o utilizador possa encontrar dificuldade com a especificação da métrica, definição de função, e a avaliação de objectos "indexados" diferenciados.

27.1.1 Nova notação d tensores

Até agora, o pacote `itensor` no Maxima tinha usado uma notação que algumas vezes conduzia a ordenação incorrecta de índices. Considere o seguinte, por exemplo:

```
(%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
```

O resultado está incorrecto a menos que ocorra ser `a` um tensor simétrico. A razão para isso é que embora `itensor` mantenha correctamente a ordem dentro do conjunto de índices covariantes e contravariantes, assim que um índice é incrementado ou decrementado, sua posição relativa para o outro conjunto de índices é perdida.

Para evitar esse problema, uma nova notação tem sido desenvolvida que mantém total compatibilidade com a notação existente e pode ser usada intercambiavelmente. Nessa notação, índices contravariantes são inseridos na posição apropriada na lista de índices covariantes, mas com um sinal de menos colocado antes. Funções como `contract` e `ishow` estão agora consciente dessa nova notação de índice e podem processar tensores apropriadamente.

Nessa nova notação, o exemplo anterior retorna um resultado correcto:

```
(%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
```

Presentemente, o único código que faz uso dessa notação é a função `lc2kdt`. Através dessa notação, a função `lc2kdt` encontra com êxito resultados consistentes como a aplicação do tensor métrico para resolver os símbolos de Levi-Civita sem reordenar para índices numéricos.

Uma vez que esse código é um tipo novo, provavelmente contém erros. Enquanto esse tipo novo não tiver sido testado para garantir que ele não interrompe nada usando a "antiga" notação de tensor, existe uma considerável chance que "novos" tensores irão falhar em interoperar com certas funções ou recursos. Essas falhas serão corrigidas à medida que forem encontradas... até então, seja cuidadoso!

27.1.2 Manipulação de tensores indiciais

o pacote de manipulação de tensores indiciais pode ser chamado através de `load("itensor")`. Demonstações estão também disponíveis: tente `demo(tensor)`. Em `itensor` um tensor é representado como um "objecto indexado" . Um "objecto indexado" é uma função de 3 grupos de índices os quais representam o covariante, o contravariante e o índice de derivação. Os índices covariantes são especificados através de uma lista com o primeiro argumento para o objecto indexado, e os índices contravariantes através de uma lista como segundo argumento. Se o objecto indexado carece de algum desses grupos de índices então a lista vazia `[]` é fornecida como o argumento correspondente. Dessa forma, `g([a,b],[c])` representa um objecto indexado chamado `g` o qual tem dois índices covariantes `(a,b)`, um índice contravariante (`c`) e não possui índices de derivação.

Os índices de derivação, se estiverem presente, são anexados ao final como argumentos adicionais para a função numérica representando o tensor. Eles podem ser explicitamente especificado pelo utilizador ou serem criados no processo de diferenciação com relação a alguma variável coordenada. Uma vez que diferenciação ordinária é comutativa, os índices de derivação são ordenados alfanumericamente, a menos que `iframe_flag` seja escolhida para `true`, indicando que um referencial métrico está a ser usado. Essa ordenação canónica torna possível para Maxima reconhecer que, por exemplo, `t([a],[b],i,j)` é o mesmo que `t([a],[b],j,i)`. Diferenciação de um objecto indexado com relação a alguma coordenada cujos índices não aparecem como um argumento para o objecto indexado podem normalmente retornar zero. Isso é porque Maxima pode não saber que o tensor representado através do objecto indexado possívelmente depende implicitamente da respectiva coordenada. Pela modificação da função existente no Maxima, `diff`, em `itensor`, Maxima sabe assumir que todos os objectos indexados dependem de qualquer variável de diferenciação a menos que seja declarado de outra forma. Isso torna possível para a convençào de somatório ser extendida para índices derivativos. Pode ser verificado que `itensor` não possui a compatibilidade de incrementar índices derivativos, e então eles são sempre tratados como covariantes.

As seguintes funções estão disponíveis no pacote tensor para manipulação de objectos. Actualmente, com relação às rotinas de simplificação, é assumido que objectos indexados não possuem por padrão propriedades simétricas. Isso pode ser modificado através da escolha da variável `allsym[false]` para `true`, o que irá resultar no tratamento de todos os objectos indexados completamente simétricos em suas listas de índices covariantes e simétricos em suas listas de índices contravariantes.

O pacote `itensor` geralmente trata tensores como objectos opacos. Equações tensoriais são manipuladas baseadas em regras algébricas, especificamente simetria e regras de contração. Adicionalmente, o pacote `itensor` não entende diferenciação covariante, curvatura, e torsão. Cálculos podem ser executados relativamente a um métrica de referenciais de movimento, dependendo da escolha para a variável `iframe_flag`.

Uma sessão demonstrativa abaixo mostra como chamar o pacote `itensor`, especificando o nome da métrica, e executando alguns cálculos simples.

```
(%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 %8                     %6 %8
(%t16) - (ifg      v   ifb       + ifg      v   ifb       - 2 v
                    %6    j %8 i             %6    i j %8      i,j

                                                     %6 %8
                                               - ifg      v   ifb      )/2
                                                           %6    %8 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 Definições para itensor

27.2.1 Gerenciando objectos indexados

`entertensor` (*nome*) [Função]

É uma função que, através da linha de comando, permite criar um objecto indexado chamado *nome* com qualquer número de índices de tensores e derivativos. Ou um índice simples ou uma lista de índices (às quais podem ser nulas) são entradas aceitáveis (veja o exemplo sob `covdiff`).

`changename` (*antigo*, *novo*, *expr*) [Função]

Irá mudar o nome de todos os objectos indexados chamados *antigo* para *novo* em *expr*. *antigo* pode ser ou um símbolo ou uma lista da forma `[`*nome*`,` *m*`,` *n*`]` nesse caso somente esses objectos indexados chamados *nome* com índice covariante *m* e índice contravariante *n* serão renomeados para *novo*.

`listoftens` [Função]

Lista todos os tensores em uma expressão tensorial, incluindo seus índices. 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*) [Função]

Mostra *expr* com os objectos indexados tendo seus índices covariantes como subscritos e índices contravariantes como sobrescritos. Os índices derivativos são mostrados como subscritos, separados dos índices covariantes por uma vírgula (veja os exemplos através desse documento).

`indices` (*expr*) [Função]

Retorna uma lista de dois elementos. O primeiro é uma lista de índices livres em *expr* (aqueles que ocorrem somente uma vez). O segundo é uma lista de indices

que ocorrem exactamente duas vezes em *expr* (dummy) como demonstra o seguinte exemplo.

```
(%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]]
```

Um produto de tensores contendo o mesmo índice mais que duas vezes é sintaticamente ilegal. `indices` tenta lidar com essas expressões de uma forma razoável; todavia, quando `indices` é chamada para operar sobre tal uma expressão ilegal, seu comportamento pode ser considerado indefinido.

`rename` (*expr*) [Função]
`rename` (*expr*, *contador*) [Função]

Retorna uma expressão equivalente para *expr* mas com índices que ocorrem exactamente duas vezes em cada termo alterado do conjunto `[%1, %2,...]`, se o segundo argumento opcional for omitido. De outra forma, os índices que ocorrem exactamente duas vezes são indexados começando no valor de *contador*. Cada índice que ocorre exactamente duas vezes em um produto será diferente. Para uma adição, `rename` irá operar sobre cada termo na a adição zerando o contador com cada termo. Nesse caminho `rename` pode servir como um simplificador tensorial. Adicionalmente, os índices serão ordenados alfanumericamente (se `allsym` for `true`) com relação a índices covariantes ou contravariantes dependendo do valor de `flipflag`. Se `flipflag` for `false` então os índices serão renomeados conforme a ordem dos índices contravariantes. Se `flipflag` for `true` a renomeação ocorrerá conforme a ordem dos índices covariantes. Isso muitas vezes ajuda que o efeito combinado dos dois restantes sejam reduzidos a uma expressão de valor um ou mais que um por si mesma.

```
(%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` [Variável de Opção]

Valor por omissão: `false`. Se `false` então os índices irão ser renomeados conforme a ordem dos índices contravariantes, de outra forma serão ordenados conforme a ordem dos índices covariantes.

Se `flipflag` for `false` então `rename` forma uma lista de índices contravariantes na ordem em que forem encontrados da esquerda para a direita (se `true` então de índices contravariantes). O primeiro índice que ocorre exactamente duas vezes na lista é renomeado para `%1`, o seguinte para `%2`, etc. Então a ordenação ocorre após a ocorrência do `rename` (veja o exemplo sob `rename`).

`defcon` (*tensor_1*) [Função]
`defcon` (*tensor_1*, *tensor_2*, *tensor_3*) [Função]

Dado *tensor_1* a propriedade que a contração de um produto do *tensor_1* e do *tensor_2* resulta em *tensor_3* com os índices apropriados. Se somente um argumento, *tensor_1*, for dado, então a contração do produto de *tensor_1* com qualquer objecto indexado tendo os índices apropriados (digamos `my_tensor`) irá retornar como resultado um objecto indexado com aquele nome, i.e. `my_tensor`, e com uma nova escolha de índices refletindo as contrações executadas. Por exemplo, se `imetric:g`, então `defcon(g)` irá implementar o incremento e decremento de índices através da contração com o tensor métrico. Mais de uma `defcon` pode ser dada para o mesmo objecto indexado; o último fornecido que for aplicado a uma contração particular será usado. `contractions` é

uma lista de objectos indexados que tenham fornecido propriedades de contrações com `defcon`.

`remcon` (*tensor_1*, ..., *tensor_n*) [Função]
`remcon` (*all*) [Função]

Remove todas as propriedades de contração de *tensor_1*, ..., *tensor_n*). `remcon(all)` remove todas as propriedades de contração de todos os objectos indexados.

`contract` (*expr*) [Função]

Realiza contrações tensoriais em *expr* a qual pode ser qualquer combinação de adições e produtos. Essa função usa a informação dada para a função `defcon`. Para melhores resultados, `expr` pode ser completamente expandida. `ratexpand` é o meio mais rápido para expandir produtos e expoentes de adições se não existirem variáveis nos denominadores dos termos. O comutador `gcd` pode ser `false` se cancelamentos de máximo divisor comum forem desnecessários.

`indexed_tensor` (*tensor*) [Função]

Deve ser executada antes de atribuir componentes para um *tensor* para o qual um valor interno já existe como com `ichr1`, `ichr2`, `icurvature`. Veja o exemplo sob `icurvature`.

`components` (*tensor*, *expr*) [Função]

Permite que se atribua um valor indicial a uma expressão *expr* dando os valores das componentes do *tensor*. Esses são automaticamente substituídos para o tensor mesmo que isso ocorra com todos os seus índices. O tensor deve ser da forma `t([...],[...])` onde qualquer lista pode ser vazia. *expr* pode ser qualquer expressão indexada envolvendo outros objectos com os mesmos índices livres que *tensor*. Quando usada para atribuir valores a um tensor métrico no qual as componentes possuem índices que ocorrem exactamente duas vezes se deve ser cuidadoso para definir esses índices de forma a evitar a geração de índices que ocorrem exactamente duas vezes e que são múltiplos. a remoção dessas atribuições é dada para a função `remcomps`.

É importante ter em mente que `components` cuida somente da valência de um tensor, e que ignora completamente qualquer ordenação particular de índices. Dessa forma atribuindo componentes a, digamos, `x([i,-j],[])`, `x([-j,i],[])`, ou `x([i],[j])` todas essas atribuições produzem o mesmo resultado, a saber componentes sendo atribuidas a um tensor chamado `x` com valência `(1,1)`.

Componentes podem ser atribuidas a uma expressão indexada por quatro caminhos, dois dos quais envolvem o uso do comando `components`:

1) Como uma expressão indexada. Por exemplo:

```
(%i2) components(g([],[i,j]),e([],[i])*p([],[j]))$
(%i3) ishow(g([],[i,j]))$
                                        i  j
(%t3)                                  e  p
```

2) Como uma matriz:

```
(%i6) components(g([i,j],[]),lg);
(%o6)                                done
(%i7) ishow(g([i,j],[]))$
(%t7)                                g
                                      i j
(%i8) g([3,3],[]);
(%o8)                                 1
(%i9) g([4,4],[]);
(%o9)                                 - 1
```

3) Como uma função. Pode usar uma função Maxima para especificar as componentes de um tensor baseado nesses índices. Por exemplo, os seguintes códigos atribuem `kdelta` a `h` se `h` tiver o mesmo número de índices covariantes e índices contravariantes e nenhum índice derivativo, e atribui `kdelta` a `g` caso as condições anteriores não sejam atendidas:

```
(%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) Usando a compatibilidade dos modelos de coincidência do Maxima, especificamente os comandos `defrule` e `applyb1`:

```
(%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*) [Função]

Desassocia todos os valores de *tensor* que foram atribuídos com a função `components`.

showcomps (*tensor*) [Função]

Mostra atribuições de componentes de um tensor, feitas usando o comando `components`. Essa função pode ser particularmente útil quando uma matriz é atribuída a um tensor indicial usando `components`, como demonstrado através do seguinte exemplo:

```
(%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
```

O comando `showcomps` pode também mostrar componentes de um tensor de categoria maior que 2.

`idummy` () [Função]

Incrementos `icounter` e retorno como seu valor um índice da forma `%n` onde n é um inteiro positivo. Isso garante que índices que ocorrem exactamente duas vezes e que são necessários na formação de expressões não irão conflitar com índices que já estiverem sendo usados (veja o exemplo sob `indices`).

`idummyx` [Variável de opção]

Valor por omissão: `%`

É o prefixo para índices que ocorrem exactamente duas vezes (veja o exemplo sob índices `indices`).

`icounter` [Variável de Opção]

Valor por omissão: `1`

Determina o sufixo numérico a ser usado na geração do próximo índice que ocorre exactamente duas vezes no pacote tensor. O prefixo é determinado através da opção `idummy` (padrão: `%`).

`kdelta` (*L1*, *L2*) [Função]

é a função delta generalizada de Kronecker definida no pacote `itensor` com *L1* a lista de índices covariantes e *L2* a lista de índices contravariantes. `kdelta([i],[j])` retorna o delta de Kronecker comum. O comando `ev(`*expr*`,kdelta)` faz com que a avaliação de uma expressão contendo `kdelta([],[])` se dê para a dimensão de multiplicação.

No que conduzir a um abuso dessa notação, `itensor` também permite `kdelta` ter 2 covariantes e nenhum contravariante, ou 2 contravariantes e nenhum índice covariante, com efeito fornecendo uma compatibilidade para "matriz unitária" covariante ou contravariante. Isso é estritamente considerado um recurso de programação e não significa implicar que `kdelta([i,j],[])` seja um objecto tensorial válido.

`kdels` (*L1*, *L2*) [Função]

Delta de Kronecker simetrizado, usado em alguns cálculos. Por exemplo:

```
(%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*) [Função]

é o tensor de permutação (ou de Levi-Civita) que retorna 1 se a lista L consistir de uma permutação par de inteiros, -1 se isso consistir de uma permutação ímpar, e 0 se alguns índices em L forem repetidos.

lc2kdt (*expr*) [Função]

Simplifica expressões contendo os símbolos de Levi-Civita, convertendo esses para expressões delta de Kronecker quando possível. A principal diferença entre essa função e simplesmente avaliar os simbolos de Levi-Civita é que a avaliação directa muitas vezes resulta em expressões Kronecker contendo índices numéricos. Isso é muitas vezes indesejável como na prevenção de simplificação adicional. A função `lc2kdt` evita esse problema, retornando expressões que são mais facilmente simplificadas com `rename` ou `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
```

A função `lc2kdt` algumas vezes faz uso de tensores métricos. Se o tensor métrico não tiver sido definido previamente com `imetric`, isso resulta em um erro.

```
(%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       k
(%t10) (g     kdelta   g     kdelta   - g     kdelta   g     kdelta  ) a
                    %3             %4              %3             %4   j k
(%i11) ishow(contract(expand(%)))$
                                  l i      l i
(%t11)                           a    - a g
```

`lc_l` [Função]

Regra de simplificação usada para expressões contendo símbolos não avaliados de Levi-Civita (`levi_civita`). Juntamente com `lc_u`, pode ser usada para simplificar muitas expressões mais eficientemente que a avaliação de `levi_civita`. Por exemplo:

```
(%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) ishow(canform(contract(expand(applyb1(el1,lc_l,lc_u)))))$
(%t4)                                  0
(%i5) ishow(canform(contract(expand(applyb1(el2,lc_l,lc_u)))))$
(%t5)                                  0
```

`lc_u` [Função]

Regra de simplificação usada para expressões contendo símbolos não avaliados de Levi-Civita (`levi_civita`). Juntamente com `lc_u`, pode ser usada para simplificar muitas

expressões mais eficientemente que a avaliação de `levi_civita`. Para detalhes, veja `lc_l`.

`canten` (*expr*) [Função]
Simplifica *expr* por renomeação (veja `rename`) e permutando índices que ocorrem exactamente duas vezes. `rename` é restrito a adições de produto de tensores nos quais nenhum índice derivativo estiver presente. Como tal isso é limitado e pode somente ser usado se `canform` não for capaz de realizar a simplificação requerida.

A função `canten` retorna um resultado matematicamente correcto somente se seu argumento for uma expressão que é completamente simétrica em seus índices. Por essa razão, `canten` retorna um erro se `allsym` não for posicionada em `true`.

`concan` (*expr*) [Função]
Similar a `canten` mas também executa contração de índices.

27.2.2 Simetrias de tensores

`allsym` [Variável de Opção]
Valor por omissão: `false`. Se `true` então todos os objectos indexados são assumidos simétricos em todos os seus índices covariantes e contravariantes. Se `false` então nenhum simétrico de qualquer tipo é assumidos nesses índices. Índices derivativos são sempre tomados para serem simétricos a menos que `iframe_flag` seja escolhida para `true`.

`decsym` (*tensor*, *m*, *n*, [*cov_1*, *cov_2*, ...], [*contr_1*, *contr_2*, ...]) [Função]
Declara propriedades de simetria para *tensor* de covariante *m* e *n* índices contravariantes. As *cov_i* e *contr_i* são pseudofunções expressando relações de simetrias em meio a índices covariante e índices contravariantes respectivamente. Esses são da forma `symoper(`*index_1*`,` *index_2*`,...)` onde `symoper` é um entre `sym`, `anti` ou `cyc` e os *index_i* são inteiros indicando a posição do índice no *tensor*. Isso irá declarar *tensor* para ser simétrico, antisimétrico ou cíclico respectivamente nos *index_i*. `symoper(all)` é também forma permitida que indica todos os índices obedecem à condição de simetria. Por exemplo, dado um objecto `b` com 5 índices covariantes, `decsym(b,5,3,[sym(1,2),anti(3,4)],[cyc(all)])` declara `b` simétrico no seu primeiro e no seu segundo índices e antisimétrico no seu terceiro e quarto índices covariantes, e cíclico em todos de seus índices contravariantes. Qualquer lista de declarações de simetria pode ser nula. A função que executa as simplificações é `canform` como o exemplo abaixo ilustra.

```
(%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*) [Função]

Remove todas as propriedades de simetria de *tensor* que tem *m* índices covariantes e *n* índices contravariantes.

`canform` (*expr*) [Função]

Simplifica *expr* através de mudança de nome de índices que ocorrem exactamente duas vezes e reordenação de todos os índices como ditados pelas condições de simetria impostas sobre eles. Se `allsym` for `true` então todos os índices são assumidos simétricos, de outra forma a informação de simetria fornecida pelas declarações `decsym` irão ser usadas. Os índices que ocorrem exactamente duas vezes são renomeados da mesma maneira que na função `rename`. Quando `canform` é aplicada a uma expressão larga o cálculo pode tomar um considerável montante de tempo. Esse tempo pode ser diminuído através do uso de `rename` sobre a expressão em primeiro lugar. Também veja o exemplo sob `decsym`. Nota: `canform` pode não estar apta a reduzir um expressão completamente para sua forma mais simples embora retorne sempre um resultado matemáticamente correcto.

27.2.3 Cálculo de tensores indiciais

`diff` (*expr*, *v_1*, [*n_1*, [*v_2*, *n_2*] ...]) [Função]

É a função usual de diferenciação do Maxima que tem sido expandida nessas habilidades para `itensor`. `diff` toma a derivada de *expr n_1* vezes com relação a *v_1*, *n_2* vezes com relação a *v_2* , etc. Para o pacote `tensor`, a função tem sido modificada de forma que os *v_i* possam ser inteiros de 1 até o valor da variável `dim`. Isso causará a conclusão da diferenciação com relação ao *v_i*ésimo membro da lista `vect_coords`. Se `vect_coords` for associado a uma variável atômica, então aquela variável subscrita através de *v_i* será usada para a variável de diferenciação. Isso permite que um array de nomes de coordenadas ou nomes subscritos como `x[1]`, `x[2]`, ... sejam usados.

idiff (*expr*, *v_1*, [*n_1*, [*v_2*, *n_2*] ...]) [Função]

Diferenciação indicial. A menos que `diff`, que diferencia com relação a uma variável independente, `idiff` possa ser usada para diferenciar com relação a uma coordenada. Para um objecto indexado, isso equivale a anexar ao final os *v_i* como índices derivativos. Subsequêntemente, índices derivativos irão ser ordenados, a menos que `iframe_flag` seja escolhida para `true`.

`idiff` pode também ser o determinante de um tensor métrico. Dessa forma, se `imetric` tiver sido associada a `G` então `idiff(determinant(g),k)` irá retornar `2*determinant(g)*ichr2([%i,k],[%i])` onde o índice que ocorre exactamente duas vezes `%i` é escolhido apropriadamente.

liediff (*v*, *ten*) [Função]

Calcula a derivada de Lie da expressão tensorial *ten* com relação ao campo vectorial *v*. *ten* pode ser qualquer expressão tensorial indexada; *v* pode ser o nome (sem índices) de um campo vectorial. Por exemplo:

```
(%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*) [Função]

Avalia todas as ocorrências do comando `idiff` na expressão tensorial *ten*.

undiff (*expr*) [Função]

Retorna uma expressão equivalente a *expr* mas com todas as derivadas de objectos indexados substituídas pela forma substantiva da função `idiff`. Seu argumento pode retornar aquele objecto indexado se a diferenciação for concluída. Isso é útil quando for desejado substituir um objecto indexado que sofreu diferenciação com alguma definição de função resultando em *expr* e então concluir a diferenciação através de digamos `ev(`*expr*`, idiff)`.

evundiff (*expr*) [Função]

Equivalente à execução de `undiff`, seguida por `ev` e `rediff`.

O ponto dessa operação é facilmente avaliar expressões que não possam ser directamente avaliadas na forma derivada. Por exemplo, o seguinte causa um erro:

```
(%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.
```

Todavia, se `icurvature` é informado em sua forma substantiva, pode ser avaliado usando `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
```

Nota: Em versões anteriores do Maxima, formas derivadas dos símbolos de Christoffel também não podiam ser avaliadas. Isso foi corrigido actualmente, de forma que `evundiff` não mais é necessária para expressões como essa:

```
(%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*, ...) [Função]

Escolhe para zero, em *expr*, todas as ocorrências de *tensor_i* que não tiverem índices derivativos.

`flushd` (*expr*, *tensor_1*, *tensor_2*, ...) [Função]

Escolhe para zero, em *expr*, todas as ocorrências de *tensor_i* que tiverem índices derivativos.

flushnd (*expr*, *tensor*, *n*) [Função]

Escolhe para zero, em *expr*, todas as ocorrências do objecto diferenciado *tensor* que tem *n* ou mais índices derivativos como demonstra o seguinte exemplo.

```
(%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*, ...) [Função]

Dados os *tensor_i* a propriedade de diferenciação da coordenada que a derivada do vector contravariante cujo nome é um dos *tensor_i* retorna um delta de Kronecker. Por exemplo, se `coord(x)` tiver sido concluída então `idiff(x([],[i]),j)` fornece `kdelta([i],[j])`. `coord` que é uma lista de todos os objectos indexados tendo essa propriedade.

remcoord (*tensor_1*, *tensor_2*, ...) [Função]

remcoord (*all*) [Função]

Remove a propriedade de coordenada de diferenciação dos `tensor_i` que foram estabelecidos através da função `coord`. `remcoord(all)` remove essa propriedade de todos os objectos indexados.

makebox (*expr*) [Função]

Mostra *expr* da mesma maneira que `show`; todavia, qualquer tensor d'Alembertiano ocorrendo em *expr* será indicado usando o símbolo `[]`. Por exemplo, `[]p([m],[n])` representa `g([],[i,j])*p([m],[n],i,j)`.

conmetderiv (*expr*, *tensor*) [Função]

Simplifica expressões contendo derivadas comuns de ambas as formas covariantes e contravariantes do tensor métrico (a restrição corrente). Por exemplo, `conmetderiv` pode relatar a derivada do tensor contravariante métrico com símbolos de Christoffel como visto adiante:

```
(%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*) [Função]
`simpmetderiv` (*expr*[, *stop*]) [Função]

Simplifica expressões contendo produtos de derivadas de tensores métricos. Especificamente, `simpmetderiv` reconhece duas identidades:

```
 ab        ab           ab                  a
g   g   + g   g     = (g   g  )   = (kdelta )   = 0
 ,d  bc        bc,d         bc ,d          c ,d
```

consequêntemente

```
 ab          ab
g   g   = - g   g
 ,d  bc          bc,d
```

e

```
 ab          ab
g   g     = g   g
 ,j  ab,i    ,i  ab,j
```

que seguem de simetrias de símbolos de Christoffel.

A função `simpmetderiv` toma um parâmetro opcional que, quando presente, faz com que a função pare após a primeira substituição feita com sucesso em uma expressão produto. A função `simpmetderiv` também faz uso da variável global *flipflag* que determina como aplicar uma ordenação "canonica" para os índices de produto.

Colocados juntos, essas compatibilidades podem ser usadas poderosamente para encontrar simplificações que são difíceis ou impossíveis de realizar de outra forma. Isso é demonstrado através do seguinte exemplo que explicitamente usa o recurso de simplificação parcial de `simpmetderiv` para obter uma expressão contractível:

```
(%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
```

Veja também `weyl.dem` para um exemplo que usa `simpmetderiv` e `conmetderiv` juntos para simplificar contrações do tensor de Weyl.

`flush1deriv` (*expr*, *tensor*) [Função]

Escolhe para zero, em `expr`, todas as ocorrências de `tensor` que possuem exactamente um índice derivativo.

27.2.4 Tensores em espaços curvos

`imetric` (*g*) [Função]

`imetric` [Variável de sistema]

Especifica a métrica através de atribuição à variável `imetric`:*g* adicionalmente, as propriedades de contração da métrica *g* são escolhidas através da execução dos comandos `defcon(`*g*`)`,`defcon(`*g*`,`*g*`,kdelta)`. A variável `imetric` (desassociada por padrão), é associada à métrica, atribuida pelo comando `imetric(`*g*`)`.

`idim` (*n*) [Função]

Escolhe as dimensões da métrica. Também inicializa as propriedades de antisimetria dos símbolos de Levi-Civita para as dimensões dadas.

`ichr1` ([*i*, *j*, *k*]) [Função]

Retorna o símbolo de Christoffel de primeiro tipo via definição

```
(g      + g      - g     )/2 .
  ik,j     jk,i     ij,k
```

Para avaliar os símbolos de Christoffel para uma métrica particular, à variável `imetric` deve ser atribuída um nome como no exemplo sob `chr2`.

`ichr2` ([*i*, *j*], [*k*]) [Função]

Retorna o símbolo de Christoffel de segundo tipo definido pela relação

```
                          ks
ichr2([i,j],[k]) = g    (g      + g      - g     )/2
                           is,j     js,i     ij,s
```

`icurvature` ([*i*, *j*, *k*], [*h*]) [Função]

Retorna o tensor da curvatura de Riemann em termos de símbolos de Christoffel de segundo tipo (`ichr2`). A seguinte notação é usada:

```
                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*, ...) [Função]

Retorna a derivada da covariante de *expr* com relação às variáveis *v_i* em termos de símbolos de Christoffel de segundo tipo (`ichr2`). Com o objectivo de avaliar esses, se pode usar `ev(`*expr*`,ichr2)`.

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) entertensor()$
Enter tensor name: a;
Enter a list of the índices covariantes: [i,j];
Enter a list of the índices contravariantes: [k];
Enter a list of the derivative indices: [];
                                     k
(%t2)                               a
                                     i j
(%i3) ishow(covdiff(%,s))$
             k         %1     k         %1     k            k     %1
(%t3)     - a     ichr2    - a     ichr2    + a      + ichr2     a
             i %1      j s    %1 j      i s    i j,s        %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*) [Função]

Impõe a condição de Lorentz através da substituição de 0 para todos os objectos indexados em *expr* que possui um índice de derivada idêntico ao índice contravariante.

igeodesic_coords (*expr*, *nome*) [Função]

Faz com que símbolos de Christoffel não diferenciados e a primeira derivada do tensor métrico tendam para zero em *expr*. O *nome* na função `igeodesic_coords` refere-se à métrica *nome* (se isso aparecer em *expr*) enquando os coeficientes de conecção devem ser chamados com os nomes `ichr1` e/ou `ichr2`. O seguinte exemplo demonstra a verificação da identidade cíclica satisfeita através do tensor da curvatura de Riemann usando a função `igeodesic_coords`.

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(icurvature([r,s,t],[u]))$
             u            u         %1         u            u         %1
(%t2) - ichr2      - ichr2     ichr2    + ichr2      + ichr2     ichr2
             r t,s        %1 s      r t        r s,t        %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           u
(%t4) - ichr2      + ichr2      + ichr2      - ichr2      - ichr2
             t s,r        t r,s       s t,r        s r,t       r t,s

                                                                 u
                                                          + ichr2
                                                                 r s,t
(%i5) canform(%);
(%o5)                                0
```

27.2.5 Referenciais móveis

Maxima actualmente tem a habilidade de executar cálculos usando referenciais móveis. Essas podem ser referenciais ortonormais (tetrads, vielbeins) ou um referencial arbitrária.

Para usar referenciais, primeiro escolha `iframe_flag` para `true`. Isso faz com que os símbolos de Christoffel, `ichr1` e `ichr2`, sejam substituídos pelos referenciais mais gerais de coeficientes de conecção `icc1` e `icc2` em cálculos. Especialmente, o comportamento de `covdiff` e `icurvature` são alterados.

O referencial é definido através de dois tensores: o campo de referencial inversa (`ifri`), a base tetrad dual), e a métrica do referencial `ifg`. A métrica do referencial é a matriz identidade para referenciais ortonormais, ou a métrica de Lorentz para referenciais ortonormais no espaço-tempo de Minkowski. O campo de referencial inverso define a base do referencial (vectores unitários). Propriedades de contração são definidas para o campo de referencial e para a métrica do referencial.

Quando `iframe_flag` for `true`, muitas expressões `itensor` usam a métrica do referencial `ifg` em lugar da métrica definida através de `imetric` para o decremento e para o incremento de índices.

IMPORTANTE: Escolhendo a variável `iframe_flag` para `true` NÃO remove a definição das propriedades de contração de uma métrica definida através de uma chamada a `defcon` ou `imetric`. Se um campo de referencial for usado, ele é melhor para definir a métrica através de atribuição desse nome para a variável `imetric` e NÃO invoque a função `imetric`.

Maxima usa esses dois tensores para definir os coeficientes de referencial (`ifc1` e `ifc2`) cuja forma parte dos coeficientes de conecção (`icc1` e `icc2`), como demonstra o seguinte exemplo:

```
(%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        i
(%t4)                  v   (ifc2     + ichr2    ) + v
                             %1 j        %1 j     ,j
(%i5) ishow(ev(%,ifc2))$
            %1    i %2
           v   ifg     (ifb        - ifb        + ifb       )
                           j %2 %1      %2 %1 j      %1 j %2     i
(%t5)      ------------------------------------------------- + v
                                  2                               ,j
(%i6) ishow(ifb([a,b,c]))$
                        %5    %4
(%t6)               ifr   ifr   (ifri        - ifri       )
                       a     b       c %4,%5       c %5,%4
```

Um método alternativo é usado para calcular o suporte do referencial (`ifb`) se o sinalizador `iframe_bracket_form` é escolhido para `false`:

```
(%i8) block([iframe_bracket_form:false],ishow(ifb([a,b,c])))$
                         %7    %6        %6      %7
(%t8)              (ifr   ifr     - ifr     ifr  ) ifri
                       a     b,%7      a,%7    b       c %6
```

`iframes ()` [Função]

Uma vez que nessa versão do Maxima, identidades de contração para `ifr` e `ifri` são sempre definidas, como é o suporte do referencial (`ifb`), essa função não faz nada.

`ifb` [Variável]

O suporte do referencial. A contribuição da métrica do referencial para os coeficientes de conecção é expressa usando o suporte do referencial:

$$\mathtt{ifc1}_{abc} = \frac{\mathtt{ifb}_{cab} + \mathtt{ifb}_{bca} + \mathtt{ifb}_{abc}}{2}$$

O suporte do referencial por si mesmo é definido em termos de campo de referencial e métrica do referencial. Dois métodos alternativos de cálculo são usados dependendo do valor de `frame_bracket_form`. Se `true` (o padrão) ou se o sinalizador `itorsion_flag` for `true`:

$$\mathtt{ifb}_{abc} = \mathtt{ifr}^{d}_{b}\,\mathtt{ifr}^{e}_{c}\,(\mathtt{ifri}_{a\,d,e} - \mathtt{ifri}_{a\,e,d} - \mathtt{ifri}_{a\,f}\,\mathtt{itr}^{f}_{d\,e})$$

Otherwise:

$$\mathtt{ifb}_{abc} = (\mathtt{ifr}^{e}_{b}\,\mathtt{ifr}^{d}_{c,e} - \mathtt{ifr}^{d}_{b,e}\,\mathtt{ifr}^{e}_{c}\,)\,\mathtt{ifri}_{a\,d}$$

`icc1` [Variável]

Coeficientes de conecção de primeiro tipo. Em `itensor`, definido como

$$\mathtt{icc1}_{abc} = \mathtt{ichr1}_{abc} - \mathtt{ikt1}_{abc} - \mathtt{inmc1}_{abc}$$

Nessa expressão, se `iframe_flag` for `true`, o símbolo de Christoffel `ichr1` é substituído com o coeficiente de conecção do referencial `ifc1`. Se `itorsion_flag` for `false`, `ikt1` será omitido. `ikt1` é também omitido se uma base de referencial for

usada, como a torsão está já calculada como parte do suporte do referencial. Ultimamente, como `inonmet_flag` é `false`, `inmc1` não estará presente.

`icc2` [Variável]

Coeficientes de conecção de segundo tipo. Em `itensor`, definido como

```
    c         c        c         c
icc2   = ichr2   - ikt2   - inmc2
    ab        ab       ab        ab
```

Nessa expressão, se `iframe_flag` for `true`, o símbolo de Christoffel `ichr2` é substituído com o coeficiente de conecção `ifc2`. Se `itorsion_flag` for `false`, `ikt2` será omitido. `ikt2` também será omitido se uma base de referencial for usada, uma vez que a torsão já está calculada como parte do suporte do referencial. Ultimamente, como `inonmet_flag` é `false`, `inmc2` não estará presente.

`ifc1` [Variável]

Coeficiente de referencial de primeiro tipo (também conhecido como coeficientes de rotação de Ricci). Esse tensor representa a contribuição da métrica do referencial para o coeficiente de conecção de primeiro tipo. Definido como:

```
           - ifb      + ifb      + ifb
                c a b      b c a      a b c
ifc1    = --------------------------------
    abc                 2
```

`ifc2` [Variável]

Coeficiente de referencial de primeiro tipo. Esse tensor representa a contribuição da métrica do referencial para o coeficiente de conecção de primeiro tipo. Definido como uma permutação de suporte de referencial (`ifb`) com os índices apropriados incrementados e decrementados como necessário:

```
    c       cd
ifc2   = ifg   ifc1
    ab             abd
```

`ifr` [Variável]

O campo do referencial. Contrai (`ifri`) para e com a forma do campo inverso do referencial para formar a métrica do referencial (`ifg`).

`ifri` [Variável]

O campo inverso do referencial. Especifica a base do referencial (vectores base duais). Juntamente com a métrica do referencial, forma a base de todos os cálculos baseados em referenciais.

`ifg` [Variável]

A métrica do referencial. O valor padrão é `kdelta`, mas pode ser mudada usando `components`.

`ifgi` [Variável]

O inverso da métrica do referencial. Contrai com a métrica do referencial (`ifg`) para `kdelta`.

`iframe_bracket_form` [Variável de Opção]

Valor por omissão: `true`

Especifica como o suporte do referencial (`ifb`) é calculado.

27.2.6 Torsão e não metricidade

Maxima pode trabalhar com torsão e não metricidade. Quando o sinalizador `itorsion_flag` for escolhido para `true`, a contribuição de torsão é adicionada aos coeficientes de conecção. Similarmente, quando o sinalizador `inonmet_flag` for `true`, componentes de não metricidades são incluídos.

`inm` [Variável]

O vector de não metricidade. Conforme a não metricidade está definida através da derivada covariante do tensor métrico. Normalmente zero, o tensor da métrica derivada covariante irá avaliar para o seguinte quando `inonmet_flag` for escolhido para `true`:

```
g     =- g  inm
 ij;k     ij  k
```

`inmc1` [Variável]

Permutação covariante de componentes do vector de não metricidade. Definida como

```
            g   inm  - inm  g   - g   inm
             ab    c      a  bc    ac    b
inmc1    = ------------------------------
     abc                2
```

(Substitue `ifg` em lugar de `g` se um referencial métrico for usada.)

`inmc2` [Variável]

Permutação covariante de componentes do vector de não metricidade. Usada nos coeficicientes de conecção se `inonmet_flag` for `true`. Definida como:

```
                      c         c         cd
          -inm  kdelta  - kdelta  inm  + g   inm  g
     c        a       b         a    b          d  ab
inmc2   = ----------------------------------------
     ab                       2
```

(Substitue `ifg` em lugar de `g` se um referencial métrico for usada.)

`ikt1` [Variável]
Permutação covariante do tensor de torsão (também conhecido como contorsão). Definido como:

$$ikt1_{abc} = \frac{-g_{ad}\,itr^{d}_{cb} - g_{bd}\,itr^{d}_{ca} - itr_{ab}\,g_{cd}}{2}$$

(Substitue `ifg` em lugar de `g` se um referencial métrico for usada.)

`ikt2` [Variável]
Permutação contravariante do tensor de torsão (também conhecida como contorsão). Definida como:

$$ikt2^{c}_{ab} = g^{cd}\,ikt1_{abd}$$

(Substitue `ifg` em lugar de `g` se um referencial métrico for usada.)

`itr` [Variável]
O tensor de torsão. Para uma métrica com torsão, diferenciação covariante repetida sobre uma funçào escalar não irá comutar,como demonstrado através do seguinte exemplo:

```
(%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 Álgebra externa (como em produto externo)

O pacote `itensor` pode executar operações sobre campos tensores covariantes totalmente antisimétricos. Um campo tensor totalmente antisimétrico de classe (0,L) corresponde a uma forma diferencial L. Sobre esses objectos, uma operação de multiplicação funciona como um produto externo, ou produto cunha, é definido.

Desafortunadamente, nem todos os autores concordam sobre a definição de produto cunha. Alguns autores preferem uma definição que corresponde à noção de antisimetrização: nessas palavras, o produto cunha de dois campos vectoriais, por exemplo, pode ser definido como

```
                a a  - a a
                 i j    j i
      a  /\ a  = -----------
       i     j        2
```

Mais geralmente, o produto de uma forma p e uma forma q pode ser definido como

```
                          1     k1..kp l1..lq
    A       /\ B       = ------ D             A       B
```

```
 i1..ip     j1..jq   (p+q)!  i1..ip j1..jq  k1..kp  l1..lq
```

onde `D` simboliza o delta de Kronecker.

Outros autores, todavia, preferem uma definição "geométrica" que corresponde à notação de elemento volume:

```
a  /\ a  = a a  - a a
 i     j    i j    j i
```

e, no caso geral

```
                        1    k1..kp l1..lq
A       /\ B       = ----- D              A       B
 i1..ip     j1..jq   p! q!  i1..ip j1..jq  k1..kp  l1..lq
```

Uma vez que `itensor` é um pacote de algebra de tensores, a primeira dessas duas definições aparenta ser a mais natural por si mesma. Muitas aplicações, todavia, usam a segunda definição. Para resolver esse dilema, um sinalizador tem sido implementado que controla o comportamento do produto cunha: se `igeowedge_flag` for `false` (o padrão), a primeira, definição `"tensorial"` é usada, de outra forma a segunda, definição `"geométrica"` irá ser aplicada.

`~` [Operador]

O operador do produto cunha é definido como sendo o acento til `~`. O til é um operador binário. Seus argumentos podem ser expressões envolvendo escalares, tensores covariantes de categoria 1, ou tensores covariantes de categoria `1` que tiverem sido declarados antisimétricos em todos os índices covariantes.

O comportamento do operador do produto cunha é controlado através do sinalizador `igeowedge_flag`, como no seguinte exemplo:

```
(%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
```

| [Operador]

A barra vertical | denota a operação binária "contração com um vector". Quando um tensor covariante totalmente antisimétrico é contraído com um vector contravariante, o resultado é o mesmo independente de qual índice foi usado para a contração. Dessa forma, é possível definir a operação de contração de uma forma livre de índices.

No pacote `itensor`, contração com um vector é sempre realizada com relação ao primeiro índice na ordem literal de ordenação. Isso garante uma melhor simplificação de expressões envolvendo o operador |. Por exemplo:

```
(%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 que isso é essencial que os tensores usado como o operador | seja declarado totalmente antisimétrico em seus índices covariantes. De outra forma, os resultados serão incorrectos.

`extdiff` (*expr*, *i*) [Função]

Calcula a derivada externa de *expr* com relação ao índice *i*. A derivada externa é formalmente definida como o produto cunha do operador de derivada parcial e uma forma diferencial. Como tal, essa operação é também controlada através da escolha de `igeowedge_flag`. Por exemplo:

```
(%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
                              j k,i    i k,j    i j,k
```

hodge (*expr*) [Função]

Calcula o Hodge dual de *expr*. Por exemplo:

```
(%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        g
                                                     %1 %106  %2 %107
                                                g        g      A        /6
                                                 %3 %108  %4 %8  %5 %6 %7
(%i9) lc2kdt(%)$

(%i10) %,kdelta$

(%i11) ishow(canform(contract(expand(%))))$
(%t11)                       - A
                                %106 %107 %108
```

igeowedge_flag [Variável de Opção]

Valor por omissão: `false`

Controla o comportamento de produto cunha e derivada externa. Quando for esconhida para `false` (o padrão), a noção de formas diferenciais irá corresponder àquela de

um campo tensor covariante totalmente antisimétrico. Quando escolhida para `true`, formas diferenciais irão concordar com a noção do elemento volume.

27.2.8 Exportando expressões TeX

O pacote `itensor` fornece suporte limitado à exportação de expressões de tensores para o TeX. Uma vez que expressões `itensor` aparecem como chamada a funções, o comando regular `tex` do Maxima não produzirá a saída esperada. Pode tentar no seu lugar o comando `tentex`, o qual tenta traduzir expressões de tensores dentro de objectos TeX indexados apropriadamente.

`tentex` (*expr*) [Função]

Para usar a função `tentex`, deve primeiro chamar `tentex`, como no seguinte exemplo:

```
(%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            i
(%t4)  ichr2    ichr2     - ichr2    ichr2     - ichr2      + ichr2
            j k      m1 l        j l      m1 k        j l,k        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 o uso da declaração `idummyx`, para evitar o aparecimento do sinal de porcentagem na expressão TeX, o qual pode induzir a erros de compilação.

Note Bem: Essa vesão da função `tentex` é um tanto quanto experimental.

27.2.9 Interagindo com o pacote `ctensor`

O pacote `itensor` possui a habilidade de gerar código Maxima que pode então ser executado no contexto do pacote `ctensor`. A função que executa essa tarefa é `ic_convert`.

`ic_convert` (*eqn*) [Função]

Converte a equação *eqn* na sintaxe `itensor` para uma declaração de atribuição `ctensor`. Adições implícitas sobre índices que ocorrem exactamente duas vezes são tornadas explícitas enquanto objectos indexados são transformados em arrays (os arrays subscritos estão na ordem de covariância seguidos de índices contravariantes dos objectos indexados). A derivada de um objecto indexado será substituída pela forma substantiva de `diff` tomada com relação a `ct_coords` subscrita pelo índice de derivação. Os símbolos de Christoffel `ichr1` e `ichr2` irão ser traduzidos para `lcs` e `mcs`, respectivamente e se `metricconvert` for `true` então todas as ocorrências da métrica com dois índices covariantes (ou contravariantes) irão ser renomeadas para `lg` (ou `ug`). Adicionalmente, ciclos `do` irão ser introduzidos adicionando sobre todos os índices livres de forma que a declaração de atribuição transformada pode ser avaliada

através de apenas fazendo `ev`. Os seguintes exemplos demonstam os recursos dessa função.

```
(%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 Palavras reservadas

As palavras seguintes do Maxima são usadas internamente pelo pacote `itensor` e não podem ser redefinidas:

```
Keyword    Comments
-----------------------------------------
indices2() versão interna de indices()
conti      Lista de índices contravariantes
covi       Lista de índices covariantes de um objecto indexado
deri       Lista de índices de derivada de um objecto indexado
name       Retorna o nome de um objecto indexado
concan
irpmon
lc0
_lc2kdt0
_lcprod
_extlc
```

28 ctensor

28.1 Introdução a ctensor

`ctensor` é um pacote de manipulação de componentes. Para usar o pacote `ctensor`, digite `load("ctensor")`. Para começar uma sessão iterativa com `ctensor`, digite `csetup()`. O primeiro que será pedido pelo pacote é a dimensão a ser manipulada. Se a dimensão for 2, 3 ou 4 então a lista de coordenadas padrão é `[x,y]`, `[x,y,z]` ou `[x,y,z,t]` respectivamente. Esses nomes podem ser mudados através da atribuição de uma nova lista de coordenadas para a variável `ct_coords` (descrita abaixo) e o utilizador é questionado sobre isso. Deve ter o cuidado de evitar conflitos de nomes de coordenadas com outras definições de objectos.

No próximo passo, o utilizador informa a métrica ou directamente ou de um ficheiro especificando sua posição ordinal. Como um exemplo de um ficheiro de métrica comum, veja `share/tensor/metrics.mac`. A métrica é armazenada na matriz LG. Finalmente, o inverso da métrica é calculado e armazenado na matriz UG. Se tem a opção de realizar todos os cálculos em séries de potência.

A seguir, mostramos um exemplo de protocolo para a métrica estática, esfericamente simétrica (coordenadas padrão) que será aplicada ao problema de derivação das equações de vácuo de Einstein (que levam à solução de Schwarzschild). Muitas das funções em `ctensor` irão ser mostradas como exemplos para a métrica padrão.

```
(%i1) load("ctensor");
(%o1)      /usr/local/lib/maxima/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 Definições para ctensor

28.2.1 Inicialização e configuração

`csetup` () [Função]

É uma função no pacote `ctensor` (component tensor) que inicializa o pacote e permite ao utilizador inserir uma métrica interativamente. Veja `ctensor` para mais detalhes.

`cmetric` (*dis*) [Função]

`cmetric` () [Função]

É uma função no pacote `ctensor` que calcula o inverso da métrica e prepara o pacote para cálculos adiante.

Se `cframe_flag` for `false`, a função calcula a métrica inversa `ug` a partir da matriz `lg` (definida pelo utilizador). O determinante da métrica é também calculado e armazenado na variável `gdet`. Mais adiante, o pacote determina se a métrica é diagonal e escolhe o valor de `diagmetric` conforme a determinação. Se o argumento opcional *dis* estiver presente e não for `false`, a saída é mostrada ao utilizador pela linha de comando para que ele possa ver o inverso da métrica.

Se `cframe_flag` for `true`, a função espera que o valor de `fri` (a matriz referencial inversa) e `lfg` (a métrica do referencial) sejam definidas. A partir dessas, a matriz do referencial `fr` e a métrica do referencial inverso `ufg` são calculadas.

`ct_coordsys` (*sistema_de_coordenadas*, *extra_arg*) [Função]

`ct_coordsys` (*sistema_de_coordenadas*) [Função]

Escolhe um sistema de coordenadas predefinido e uma métrica. O argumento *sistema_de_coordenadas* pode ser um dos seguintes símbolos:

SYMBOL	Dim	Coordenadas	Descrição/comentários
cartesian2d	2	[x,y]	Sist. de coord. cartesianas 2D
polar	2	[r,phi]	Sist. de coord. Polare
elliptic	2	[u,v]	
confocalelliptic	2	[u,v]	
bipolar	2	[u,v]	
parabolic	2	[u,v]	
cartesian3d	3	[x,y,z]	Sist. de coord. cartesianas 3D

```
polarcylindrical      3  [r,theta,z]
ellipticcylindrical   3  [u,v,z]           Elíptica 2D com Z cilíndrico
confocalellipsoidal   3  [u,v,w]
bipolarcylindrical    3  [u,v,z]           Bipolar 2D com Z cilíndrico
paraboliccylindrical  3  [u,v,z]           Parabólico 2D com Z cilíndrico
paraboloidal          3  [u,v,phi]
conical               3  [u,v,w]
toroidal              3  [u,v,phi]
spherical             3  [r,theta,phi]     Sist. de coord. Esféricas
oblatespheroidal      3  [u,v,phi]
oblatespheroidalsqrt  3  [u,v,phi]
prolatespheroidal     3  [u,v,phi]
prolatespheroidalsqrt 3  [u,v,phi]
ellipsoidal           3  [r,theta,phi]
cartesian4d           4  [x,y,z,t]         Sist. de coord. 4D
spherical4d           4  [r,theta,eta,phi]
exteriorschwarzschild 4  [t,r,theta,phi]   Métrica de Schwarzschild
interiorschwarzschild 4  [t,z,u,v]        Métrica de Schwarzschild Interior
kerr_newman           4  [t,r,theta,phi]   Métrica simétrica axialmente alterad
```

`sistema_de_coordenadas` pode também ser uma lista de funções de transformação, seguida por uma lista contendo as varáveis coordenadas. Por exemplo, pode especificar uma métrica esférica como segue:

```
(%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
```

Funções de transformação podem também serem usadas quando `cframe_flag` for `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;
      [ cos(phi) cos(theta)  - cos(phi) r sin(theta)  - sin(phi) r cos(theta) ]
      [                                                                       ]
(%o4) [ 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) ]
```

O argumento opcional *extra_arg* pode ser qualquer um dos seguintes:

`cylindrical` diz a `ct_coordsys` para anexar uma coordenada adicional cilíndrica.

`minkowski` diz a `ct_coordsys` para anexar uma coordenada com assinatura métrica negativa.

`all` diz a `ct_coordsys` para chamar `cmetric` e `christof(false)` após escolher a métrica.

Se a variável global `verbose` for escolhida para `true`, `ct_coordsys` mostra os valores de `dim`, `ct_coords`, e ou `lg` ou `lfg` e `fri`, dependendo do valor de `cframe_flag`.

`init_ctensor ()` [Função]

Inicializa o pacote `ctensor`.

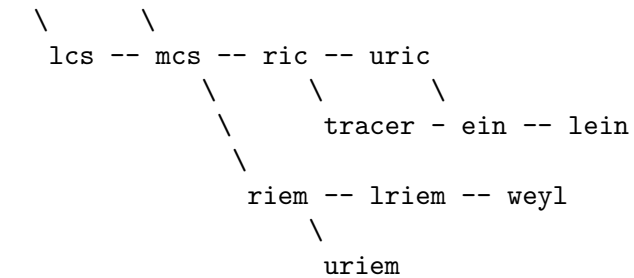
A função `init_ctensor` reinicializa o pacote `ctensor`. Essa função remove todos os arrays e matrizes usados por `ctensor`, coloca todos os sinalizadores de volta a seus valores padrão, retorna `dim` para 4, e retorna a métrica do referencial para a métrica do referencial de Lorentz.

28.2.2 Os tensores do espaço curvo

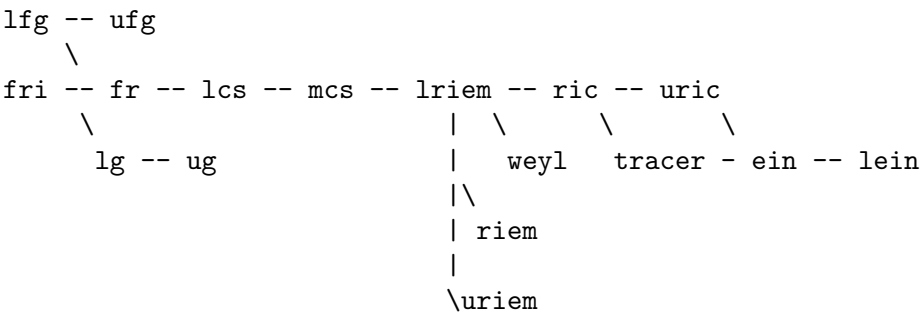
O principal propósito do pacote `ctensor` é calcular os tensores do espaç(tempo) curvo, mais notavelmente os tensores usados na relatividade geral.

Quando uma base métrica é usada, `ctensor` pode calcular os seguintes tensores:

```
lg  -- ug
```



`ctensor` pode também usar referenciais móveis. Quando `cframe_flag` for escolhida para `true`, os seguintes tensores podem ser calculados:



`christof` (*dis*) [Função]

Uma função no pacote `ctensor`. Essa função calcula os símbolos de Christoffel de ambos os tipos. O argumento *dis* determina quais resultados são para serem imediatamente mostrados. Os símbolos de Christoffel de primeiro e de segundo tipo são armazenados nos arrays `lcs[i,j,k]` e `mcs[i,j,k]` respectivamente e definidos para serem simétricos nos primeiros dois índices. Se o argumento para `christof` for `lcs` ou for `mcs` então o único valor não nulo de `lcs[i,j,k]` ou de `mcs[i,j,k]`, respectivamente, será mostrado. Se o argumento for `all` então o único valor não nulo de `lcs[i,j,k]` e o único valor não nulo de `mcs[i,j,k]` serão mostrados. Se o argumento for `false` então a exibição dos elementos não acontecerá. Os elementos do array `mcs[i,j,k]` são definidos de uma tal maneira que o índice final é contravariante.

`ricci` (*dis*) [Função]

Uma função no pacote `ctensor`. `ricci` calcula as componentes contravariantes (simétricas) `ric[i,j]` do tensor de Ricci. Se o argumento *dis* for `true`, então as componentes não nulas são mostradas.

`uricci` (*dis*) [Função]

Essa função primeiro calcula as componentes contravariantes `ric[i,j]` do tensor de Ricci. Então o tensor misto de Ricci é calculado usando o tensor métrico contravariante. Se o valor do argumento *dis* for `true`, então essas componentes mistas, `uric[i,j]` (o índice `"i"` é covariante e o índice `"j"` é contravariante), serão mostradas directamente. De outra forma, `ricci(false)` irá simplesmente calcular as entradas do array `uric[i,j]` sem mostrar os resultados.

`scurvature` () [Função]

Retorna a curvatura escalar (obtida através da contração do tensor de Ricci) do Riemaniano multiplicado com a métrica dada.

`einstein` (*dis*) [Função]

Uma função no pacote `ctensor`. `einstein` calcula o tensor misto de Einstein após os símbolos de Christoffel e o tensor de Ricci terem sido obtidos (com as funções `christof` e `ricci`). Se o argumento *dis* for `true`, então os valores não nulos do tensor misto de Einstein `ein[i,j]` serão mostrados quando `j` for o índice contravariante. A variável `rateinstein` fará com que a simplificação racional ocorra sobre esses componentes. Se `ratfac` for `true` então as componentes irão também ser factoradas.

`leinstein` (*dis*) [Função]

Tensor covariante de Einstein. `leinstein` armazena o valor do tensor covariante de Einstein no array `lein`. O tensor covariante de Einstein é calculado a partir tensor misto de Einstein `ein` através da multiplicação desse pelo tensor métrico. Se o argumento *dis* for `true`, então os valores não nulos do tensor covariante de Einstein são mostrados.

`riemann` (*dis*) [Função]

Uma função no pacote `ctensor`. `riemann` calcula o tensor de curvatura de Riemann a partir da métrica dada e correspondendo aos símbolos de Christoffel. As seguintes convenções de índice são usadas:

$$R[i,j,k,l] = R^{l}_{ijk} = \Gamma^{l}_{ij,k} - \Gamma^{l}_{ik,j} + \Gamma^{l}_{mk}\Gamma^{m}_{ij} - \Gamma^{l}_{mj}\Gamma^{m}_{ik}$$

Essa notação é consistente com a notação usada por no pacote `itensor` e sua função `icurvature`. Se o argumento opcional *dis* for `true`, as componentes não nulas `riem[i,j,k,l]` serão mostradas. Como com o tensor de Einstein, vários comutadores escolhidos pelo utilizador controlam a simplificação de componentes do tensor de Riemann. Se `ratriemann` for `true`, então simplificação racional será feita. Se `ratfac` for `true` então cada uma das componentes irá também ser factorada.

Se a variável `cframe_flag` for `false`, o tensor de Riemann é calculado directamente dos símbolos de Christoffel. Se `cframe_flag` for `true`, o tensor covariante de Riemann é calculado primeiro dos coeficientes de campo do referencial.

`lriemann` (*dis*) [Função]

Tensor covariante de Riemann (`lriem[]`).

Calcula o tensor covariante de Riemann como o array `lriem`. Se o argumento *dis* for `true`, únicos valores não nulos são mostrados.

Se a variável `cframe_flag` for `true`, o tensor covariante de Riemann é calculado directamente dos coeficientes de campo do referencial. De outra forma, o tensor (3,1) de Riemann é calculado primeiro.

Para informação sobre a ordenação de índice, veja `riemann`.

`uriemann` (*dis*) [Função]

Calcula as componentes contravariantes do tensor de curvatura de Riemann como elementos do array `uriem[i,j,k,l]`. Esses são mostrados se *dis* for `true`.

`rinvariant` () [Função]

Compõe o invariante de Kretchmann (`kinvariant`) obtido através da contração dos tensores

```
lriem[i,j,k,l]*uriem[i,j,k,l].
```

Esse objecto não é automaticamente simplificado devido ao facto de poder ser muito largo.

`weyl` (*dis*) [Função]

Calcula o tensor conformal de Weyl. Se o argumento *dis* for `true`, as componentes não nulas `weyl[i,j,k,l]` irão ser mostradas para o utilizador. De outra forma, essas componentes irão simplesmente serem calculadas e armazenadas. Se o comutador `ratweyl` é escolhido para `true`, então as componentes irão ser racionalmente simplificadas; se `ratfac` for `true` então os resultados irão ser factorados também.

28.2.3 Expansão das séries de Taylor

O pacote `ctensor` possui a habilidade para truncar resultados assumindo que eles são aproximações das séries de Taylor. Esse comportamenteo é controlado através da variável `ctayswitch`; quando escolhida para `true`, `ctensor` faz uso internamente da função `ctaylor` quando simplifica resultados.

A função `ctaylor` é invocada pelas seguintes funções de `ctensor`:

```
Function     Comments
--------------------------------
christof()   só para mcs
ricci()
uricci()
einstein()
riemann()
weyl()
checkdiv()
```

`ctaylor` () [Função]

A função `ctaylor` trunca seus argumentos através da conversão destes para uma série de Taylor usando `taylor`, e então chamando `ratdisrep`. Isso tem efeito combinado de abandonar termos de ordem mais alta na variável de expansão `ctayvar`. A ordem dos termos que podem ser abandonados é definida através de `ctaypov`; o ponto em torno do qual a expansão da série é realizada está especificado em `ctaypt`.

Como um exemplo, considere uma métrica simples que é uma perturbação da métrica de Minkowski. Sem restrições adicionais, mesmo uma métrica diagonal produz expressões para o tensor de Einstein que são de longe muito complexas:

```
(%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
```

Todavia, se nós recalcularmos esse exemplo como uma aproximação que é linear na variável `l`, pegamos expressões muito simples:

```
(%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
```

Essa compatibilidade pode ser útil, por exemplo, quando trabalhamos no limite do campo fraco longe de uma fonte gravitacional.

28.2.4 Campos de referencial

Quando a variável `cframe_flag` for escolhida para `true`, o pacote `ctensor` executa seus cálculos usando um referencial móvel.

`frame_bracket` (*fr*, *fri*, *diagframe*) [Função]

O delimitador do referencial (`fb[]`).

Calcula o delimitador do referencial conforme a seguinte definição:

```
   c          c         c        d     e
ifb   = ( ifri    - ifri    ) ifr   ifr
   ab         d,e       e,d      a     b
```

28.2.5 Classificação Algébrica

Um novo recurso (a partir de November de 2004) de `ctensor` é sua habilidade para calcular a classificação de Petrov de uma métrica espaço tempo tetradimensional. Para uma demonstração dessa compatibilidade, veja o ficheiro `share/tensor/petrov.dem`.

`nptetrad` () [Função]

Calcula um tetrad nulo de Newman-Penrose (`np`) e seus índices ascendentes em contrapartida (`npi`). Veja `petrov` para um exemplo.

O tetrad nulo é construído assumindo que um referencial métrico ortonormal tetradimensional com assinatura métrica (-,+,+,+) está sendo usada. As componentes do tetrad nulo são relacionadas para a matriz referencial inversa como segue:

```
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*) [Função]

Calcula os cinco coeficientes de Newman-Penrose `psi[0]`...`psi[4]`. Se `psi` for escolhida para `true`, os coeficientes são mostrados. Veja `petrov` para um exemplo.

Esses coeficientes são calculados a partir do tensor de Weyl em uma base de coordenada. Se uma base de referencial for usada, o tensor de Weyl é primeiro convertido para a base de coordenada, que pode ser um procedimento computacional expansível. Por essa razão, em alguns casos pode ser mais vantajoso usar uma base de coordenada em primeiro lugar antes que o tensor de Weyl seja calculado. Note todavia, que para a construção de um tetrad nulo de Newman-Penrose é necessário uma base de referencial. Portanto, uma sequência de cálculo expressiva pode começar com uma base de referencial, que é então usada para calcular `lg` (calculada automaticamente através de `cmetric`) e em seguida calcula `ug`. Nesse ponto, pode comutar de volta para uma base de coordenada escolhendo `cframe_flag` para `false` antes de começar a calcular os símbolos de Christoffel. Mudando para uma base de referencial num estágio posterior pode retornar resultados inconsistentes, já que pode terminar com uma grande mistura de tensores, alguns calculados numa base de referencial, e outros numa base de coordenada, sem nenhum modo para distinguir entre os dois tipos.

petrov () [Função]

Calcula a classificação de petrov da métrica caracterizada através de `psi[0]`...`psi[4]`.

Por exemplo, o seguinte demonstra como obter a classificação de Petrov da métrica de Kerr:

```
(%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
```

A função de classificação Petrov é baseada no algoritmo publicado em "Classifying geometries in general relativity: III Classification in practice" por Pollney, Skea, e d'Inverno, Class. Quant. Grav. 17 2885-2902 (2000). Exceto para alguns casos de teste simples, a implementação não está testada até 19 de Dezembro de 2004, e é provável que contenha erros.

28.2.6 Torsão e não metricidade

`ctensor` possui a habilidade de calcular e incluir coeficientes de torsão e não metricidade nos coeficientes de conecção.

Os coeficientes de torsão são calculados a partir de um tensor fornecido pelo utilizador `tr`, que pode ser um tensor de categoria (2,1). A partir disso, os coeficientes de torsão `kt` são calculados de acordo com a seguinte fórmula:

```
                 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 que somente o tensor de índice misto é calculao e armazenado no array `kt`.

Os coeficientes de não metricidade são calculados a partir do vector de não metricidade fornecido pelo utilizador `nm`. A partir disso, os coeficientes de não metricidade `nmc` são calculados como segue:

```
              k    k         km
         -nm  D  - D  nm  + g   nm  g
   k        i  j    i   j          m  ij
nmc  = -------------------------------
   ij                2
```

onde D simboliza o delta de Kronecker.

Quando `ctorsion_flag` for escolhida para `true`, os valores de `kt` são subtraídos dos coeficientes de conecção indexados mistos calculados através de `christof` e armazenados em `mcs`. Similarmente, se `cnonmet_flag` for escolhida para `true`, os valores de `nmc` são subtraídos dos coeficientes de conecção indexados mistos.

Se necessário, `christof` chama as funções `contortion` e `nonmetricity` com o objectivo de calcular `kt` e `nm`.

`contortion` (*tr*) [Função]
: Calcula os coeficientes de contorsão de categoria (2,1) a partir do tensor de torsão *tr*.

`nonmetricity` (*nm*) [Função]
: Calcula o coeficiente de não metricidade de categoria (2,1) a partir do vector de não metricidade *nm*.

28.2.7 Recursos diversos

`ctransform` (*M*) [Função]

Uma função no pacote `ctensor` que irá executar uma transformação de coordenadas sobre uma matriz simétrica quadrada arbitrária *M*. O utilizador deve informar as funçãoes que definem a transformação. (Formalmente chamada `transform`.)

`findde` (*A*, *n*) [Função]

Retorna uma lista de equações diferenciais únicas (expressões) correspondendo aos elementos do array quadrado *n* dimensional *A*. Actualmente, *n* pode ser 2 ou 3. `deindex` é uma lista global contendo os índices de *A* correspondendo a essas únicas equações diferenciais. Para o tensor de Einstein (`ein`), que é um array dimensional, se calculado para a métrica no exemplo abaixo, `findde` fornece as seguintes equações diferenciais independentes:

```
(%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 + 2 a d d
       x                    x x         x        x    x           x

                                                        2         2
                                                  - 2 a  d , a  x + a  - a]
                                                       x     x
(%i10) deindex;
```

```
(%o10)                    [[1, 1], [2, 2], [4, 4]]
```

cograd () [Função]

Calcula o gradiente covariante de uma função escalar permitindo ao utilizador escolher o nome do vector correspondente como o exemplo sob `contragrad` ilustra.

contragrad () [Função]

Calcula o gradiente contravariante de uma função escalar permitindo ao utilizador escolher o nome do vector correspondente como o exemplo abaixo como ilustra a métrica de Schwarzschild:

```
(%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 () [Função]

Calcula o tensor d'Alembertiano da função escalar assim que as dependências tiverem sido declaradas sobre a função. Po exemplo:

```
(%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` () [Função]

Calcula a divergência covariante do tensor de segunda categoria misto (cujo primeiro índice deve ser covariante) imprimindo as correspondentes n componentes do campo do vector (a divergência) onde n = `dim`. Se o argumento para a função for `g` então a divergência do tensor de Einstein será formada e pode ser zero. Adicionalmente, a divergência (vector) é dada no array chamado `div`.

`cgeodesic` (*dis*) [Função]

Uma função no pacote `ctensor`. `cgeodesic` calcula as equações geodésicas de movimento para uma dada métrica. Elas são armazenadas no array `geod[i]`. Se o argumento *dis* for `true` então essas equações são mostradas.

`bdvac` (*f*) [Função]

Gera as componentes covariantes das equações de campo de vácuo da teoria de gravitação de Brans-Dicke. O campo escalar é especificado através do argumento *f*, que pode ser um nome de função (com apóstrofo) com dependências funcionais, e.g., `'p(x)`.

As componentes de segunda categoria do tensor campo covariante são as componentes de segunda categoria representadas pelo array `bd`.

`invariant1` () [Função]

Gera o tensor misto de Euler-Lagrange (equações de campo) para a densidade invariante de R^2. As equações de campo são componentes de um array chamado `inv1`.

`invariant2` () [Função]

*** NOT YET IMPLEMENTED ***

Gera o tensor misto de Euler-Lagrange (equações de campo) para a densidade invariante de `ric[i,j]*uriem[i,j]`. As equações de campo são as componentes de um array chamado `inv2`.

`bimetric` () [Função]

*** NOT YET IMPLEMENTED ***

Gera as euauações de campo da teoria bimétrica de Rosen. As equações de campo são as componentes de um array chamado `rosen`.

28.2.8 Funções utilitárias

`diagmatrixp` (*M*) [Função]

Retorna `true` se *M* for uma matriz diagonal ou um array (2D).

`symmetricp` (*M*) [Função]

Retorna `true` se *M* for uma matriz simétrica ou um array (2D).

`ntermst (`*f*`)` [Função]

Fornece ao utilizador um rápido quadro do "tamanho" do tensor duplamente subscrito (array) *f*. Imprime uma lista de dois elementos onde o segundo elemento corresponde a N-TERMOS de componentes especificadas através dos primeiros elementos. Nesse caminho, é possível rapidamente encontrar as expressões não nulas e tentar simplificação.

`cdisplay (`*ten*`)` [Função]

Mostra todos os elementos do tensor *ten*, como representados por um array multidimensional. Tensores de categoria 0 e 1, assim como outros tipos de variáveis, são mostrados com `ldisplay`. Tensores de categoria 2 são mostrados como matrizes bidimensionais, enquanto tensores de alta categoria são mostrados como uma lista de matrizes bidimensionais. Por exemplo, o tensor de Riemann da métrica de Schwarzschild pode ser visto como:

```
(%i1) load("ctensor");
(%o1)       /share/tensor/ctensor.mac
(%i2) ratfac:true;
(%o2)                                true
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3)                                done
(%i4) riemann(false);
(%o4)                                done
(%i5) cdisplay(riem);
                [ 0              0                   0           0      ]
                [                                                       ]
                [                                 2                     ]
                [      3 m (r - 2 m)   m    2 m                         ]
                [ 0  - ------------- + -- - ----       0           0      ]
                [            4          3     4                         ]
                [           r          r     r                          ]
                [                                                       ]
     riem     = [                                m (r - 2 m)           ]
         1, 1   [ 0              0               -----------      0      ]
                [                                     4                 ]
                [                                    r                  ]
                [                                                       ]
                [                                              m (r - 2 m) ]
                [ 0              0                   0       -----------  ]
                [                                                4      ]
                [                                               r       ]

                                  [    2 m (r - 2 m)       ]
                                  [ 0  -------------  0  0 ]
                                  [          4            ]
                                  [         r             ]
                       riem     = [                       ]
                           1, 2   [ 0        0        0  0 ]
```

```
                                  [                      ]
                                  [ 0        0        0  0 ]
                                  [                      ]
                                  [ 0        0        0  0 ]

                                  [         m (r - 2 m)    ]
                                  [ 0  0  - -----------  0 ]
                                  [              4       ]
                                  [             r        ]
                       riem     = [                      ]
                           1, 3   [ 0  0        0        0 ]
                                  [                      ]
                                  [ 0  0        0        0 ]
                                  [                      ]
                                  [ 0  0        0        0 ]

                                  [            m (r - 2 m) ]
                                  [ 0  0  0  - ----------- ]
                                  [                 4      ]
                                  [                r       ]
                       riem     = [                       ]
                           1, 4   [ 0  0  0        0       ]
                                  [                       ]
                                  [ 0  0  0        0       ]
                                  [                       ]
                                  [ 0  0  0        0       ]

                                 [       0        0  0  0 ]
                                 [                        ]
                                 [       2 m              ]
                                 [ - ------------  0  0  0 ]
                       riem     = [    2                   ]
                           2, 1   [   r  (r - 2 m)         ]
                                  [                        ]
                                  [       0        0  0  0 ]
                                  [                        ]
                                  [       0        0  0  0 ]

                            [     2 m                                  ]
                            [ ------------  0        0              0  ]
                            [  2                                       ]
                            [ r  (r - 2 m)                             ]
                            [                                          ]
                            [      0        0        0              0  ]
                            [                                          ]
                  riem     = [                      m                  ]
                      2, 2   [      0        0  - ------------        0  ]
```

```
[                    2                              ]
[                   r  (r - 2 m)                    ]
[                                                   ]
[                                          m        ]
[      0         0         0          - ------------ ]
[                                          2        ]
[                                        r  (r - 2 m) ]
```

```
                [ 0  0       0         0 ]
                [                        ]
                [            m           ]
                [ 0  0  ------------  0 ]
riem        = [         2               ]
    2, 3        [        r  (r - 2 m)    ]
                [                        ]
                [ 0  0       0         0 ]
                [                        ]
                [ 0  0       0         0 ]
```

```
                [ 0  0  0       0        ]
                [                        ]
                [               m        ]
                [ 0  0  0  ------------  ]
riem        = [            2            ]
    2, 4        [           r  (r - 2 m) ]
                [                        ]
                [ 0  0  0       0        ]
                [                        ]
                [ 0  0  0       0        ]
```

```
                      [ 0  0  0  0 ]
                      [            ]
                      [ 0  0  0  0 ]
                      [            ]
riem        = [ m                  ]
    3, 1      [ -  0  0  0 ]
                      [ r          ]
                      [            ]
                      [ 0  0  0  0 ]
```

```
                      [ 0  0  0  0 ]
                      [            ]
                      [ 0  0  0  0 ]
                      [            ]
riem        = [    m               ]
    3, 2      [ 0  -  0  0 ]
                      [    r       ]
```

```
                    [           ]
                    [ 0  0  0  0 ]

                [   m                    ]
                [ - -   0   0       0    ]
                [   r                    ]
                [                        ]
                [       m                ]
                [  0   - -  0       0    ]
riem        =   [       r                ]
    3, 3        [                        ]
                [  0    0   0       0    ]
                [                        ]
                [             2 m - r    ]
                [  0    0   0  ------- + 1 ]
                [               r        ]

                    [ 0  0  0    0   ]
                    [                ]
                    [ 0  0  0    0   ]
                    [                ]
    riem        =   [            2 m ]
        3, 4        [ 0  0  0  - --- ]
                    [             r  ]
                    [                ]
                    [ 0  0  0    0   ]

                [       0        0  0  0 ]
                [                        ]
                [       0        0  0  0 ]
                [                        ]
 riem       =   [       0        0  0  0 ]
    4, 1        [                        ]
                [      2                 ]
                [ m sin (theta)          ]
                [ -------------  0  0  0 ]
                [       r                ]

                [ 0        0        0  0 ]
                [                        ]
                [ 0        0        0  0 ]
                [                        ]
 riem       =   [ 0        0        0  0 ]
    4, 2        [                        ]
                [         2              ]
                [    m sin (theta)       ]
                [ 0  -------------  0  0 ]
```

```
                                  [          r             ]

                                [ 0  0          0          0 ]
                                [                            ]
                                [ 0  0          0          0 ]
                                [                            ]
                     riem     = [ 0  0          0          0 ]
                         4, 3   [                            ]
                                [               2            ]
                                [         2 m sin (theta)    ]
                                [ 0  0  - ---------------  0 ]
                                [                 r          ]

                  [        2                                          ]
                  [   m sin (theta)                                   ]
                  [ - -------------         0                0        0 ]
                  [         r                                         ]
                  [                                                   ]
                  [                         2                         ]
                  [                   m sin (theta)                   ]
riem     =        [        0         - -------------         0        0 ]
    4, 4          [                          r                        ]
                  [                                                   ]
                  [                                        2          ]
                  [                                 2 m sin (theta)   ]
                  [        0                0        ---------------  0 ]
                  [                                        r          ]
                  [                                                   ]
                  [        0                0                0        0 ]

(%o5)                                done
```

`deleten` (*L*, *n*) [Função]

Retorna uma nova lista consistindo de *L* com o *n*'ésimo elemento apagado.

28.2.9 Variáveis usadas por `ctensor`

`dim` [Variável de opção]

Valor por omissão: 4

Uma opção no pacote `ctensor`. `dim` é a dimensão de multiplicação com o padrão 4. O comando `dim: n` irá escolher a dimensão para qualquer outro valor `n`.

`diagmetric` [Variável de opção]

Valor por omissão: `false`

Uma opção no pacote `ctensor`. Se `diagmetric` for `true` rotinas especiais calculam todos os objectos geométricos (que possuem o tensor métrico explicitamente) levando

em consideração a diagonalidade da métrica. Tempo de execuçào reduzido irá, com certeza, resultar dessa escolha. Nota: essa opção é escolhida automaticamente por `csetup` se uma métrica diagonal for especificada.

`ctrgsimp` [Variável de opção]
Faz com que simplificações trigonométricas sejam usadas quando tensores forem calculados. Actualmente, `ctrgsimp` afecta somente cálculos envolvendo um referencial móvel.

`cframe_flag` [Variável de opção]
Faz com que cálculos sejam executados relativamente a um referencial móvel em oposição a uma métrica holonómica. O referencial é definido através do array do referencial inverso `fri` e da métrica do referencial `lfg`. Para cálculos usando um referencial Cartesiano, `lfg` pode ser a matriz unitária de dimensão apropriada; para cálculos num referencial de Lorentz, `lfg` pode ter a assinatura apropriada.

`ctorsion_flag` [Variável de opção]
Faz com que o tensor de contorsão seja incluído no cálculo dos coeficientes de conecção. O tensor de contorsão por si mesmo é calculado através de `contortion` a partir do tensor `tr` fornecido pelo utilizador.

`cnonmet_flag` [Variável de opção]
Faz com que os coeficientes de não metricidade sejam incluídos no cálculo dos coeficientes de conecção. Os coeficientes de não metricidade são calculados a partir do vector de não metricidade `nm` fornecido pelo utilizador através da função `nonmetricity`.

`ctayswitch` [Variável de opção]
Se escolhida para `true`, faz com que alguns cálculos de `ctensor` sejam realizados usando expansões das séries de Taylor. actualmente, `christof`, `ricci`, `uricci`, `einstein`, e `weyl` levam em conta essa escolha.

`ctayvar` [Variável de opção]
Variável usada pela expansão de séries de Taylor se `ctayswitch` é escolhida para `true`.

`ctaypov` [Variável de opção]
Maximo expoente usado em expansões de séries de Taylor quando `ctayswitch` for escolhida para `true`.

`ctaypt` [Variável de opção]
Ponto em torno do qual expansões de séries de Taylor sao realizadas quando `ctayswitch` for escolhida para `true`.

`gdet` [Variável de sistema]
O determinante do tensor métrico `lg`. Calculado através de `cmetric` quando `cframe_flag` for escolhido para `false`.

`ratchristof` [Variável de opção]
Faz com que simplificações racionais sejam aplicadas através de `christof`.

`rateinstein` [Variável de opção]

Valor por omissão: `true`

Se `true` simplificação racional será executada sobre as componentes não nulas de tensores de Einstein; se `ratfac` for `true` então as componentes irão também ser factoradas.

`ratriemann` [Variável de opção]

Valor por omissão: `true`

Um dos comutadores que controlam simplificações dos tensores de Riemann; se `true`, então simplificações racionais irão ser concluídas; se `ratfac` for `true` então cada uma das componentes irá também ser factorada.

`ratweyl` [Variável de opção]

Valor por omissão: `true`

Se `true`, esse comutador faz com que a função de `weyl` aplique simplificações racionais aos valores do tensor de Weyl. Se `ratfac` for `true`, então as componentes irão também ser factoradas.

`lfg` [Variável]

O referencial métrico covariante. Por padrão, é inicializado para o referencial tetradimensional de Lorentz com assinatura (+,+,+,-). Usada quando `cframe_flag` for `true`.

`ufg` [Variável]

A métrica do referencial inverso. Calculada de `lfg` quando `cmetric` for chamada enquanto `cframe_flag` for escolhida para `true`.

`riem` [Variável]

O tensor de categoria (3,1) de Riemann. Calculado quando a função `riemann` é invocada. Para informação sobre ordenação de índices, veja a descrição de `riemann`.

Se `cframe_flag` for `true`, `riem` é calculado a partir do tensor covariante de Riemann `lriem`.

`lriem` [Variável]

O tensor covariante de Riemann. Calculado através de `lriemann`.

`uriem` [Variável]

O tensor contravariante de Riemann. Calculado através de `uriemann`.

`ric` [Variável]

O tensor misto de Ricci. Calculado através de `ricci`.

`uric` [Variável]

O tensor contravariante de Ricci. Calculado através de `uricci`.

`lg` [Variável]

O tensor métrico. Esse tensor deve ser especificado (como uma `dim` através da matriz `dim`) antes que outro cálculo possa ser executado.

`ug` [Variável]

O inverso do tensor métrico. Calculado através de `cmetric`.

`weyl` [Variável]
O tensor de Weyl. Calculado através de `weyl`.

`fb` [Variável]
Coeficientes delimitadores do referencial, como calculado através de `frame_bracket`.

`kinvariant` [Variável]
O invariante de Kretchmann. Calculado através de `rinvariant`.

`np` [Variável]
Um tetrad nulo de Newman-Penrose. Calculado através de `nptetrad`.

`npi` [Variável]
O índice ascendente do tetrad nulo de Newman-Penrose. Calculado através de `nptetrad`. Definido como `ug.np`. O produto `np.transpose(npi)` é constante:

```
(%i39) trigsimp(np.transpose(npi));
                                [  0   - 1  0  0 ]
                                [                ]
                                [ - 1   0   0  0 ]
(%o39)                          [                ]
                                [  0    0   0  1 ]
                                [                ]
                                [  0    0   1  0 ]
```

`tr` [Variável]
Tensor de categoria 3 fornecido pelo utilizador representando torsão. Usado por `contortion`.

`kt` [Variável]
O tensor de contorsão, calculado a partir de `tr` através de `contortion`.

`nm` [Variável]
Vetor de não metrcidade fornecido pelo utilizador. Usado por `nonmetricity`.

`nmc` [Variável]
Os coeficientes de não metricidade, calculados a partir de `nm` por `nonmetricity`.

`tensorkill` [Variável de sistema]
Variável indicando se o pacote tensor foi inicializado. Escolhida e usada por `csetup`, retornada ao seu valor original através de `init_ctensor`.

`ct_coords` [Variável de opção]
Valor por omissão: `[]`

Uma opção no pacote `ctensor`. `ct_coords` contém uma lista de coordenadas. Enquanto normalmente definida quando a função `csetup` for chamada, se pode redefinir as coordenadas com a atribuição `ct_coords: [j1, j2, ..., jn]` onde os j's são os novos nomes de coordenadas. Veja também `csetup`.

28.2.10 Nomes reservados

Os seguintes nomes são usados internamente pelo pacote `ctensor` e não devem ser redefinidos:

```
Name         Description
--------------------------------------
_lg()        Avalia para lfg se for usado o referencial métrico,
                  para lg de outra forma
_ug()        Avalia para ufg se for usado o referencial métrico,
                  para ug de outra forma
cleanup()    Remove ítens da lista deindex
contract4()  Usado por psi()
filemet()    Usado por csetup() quando lendo a métrica de um ficheiro
findde1()    Usado por findde()
findde2()    Usado por findde()
findde3()    Usado por findde()
kdelt()      Delta de Kronecker (não generalizado)
newmet()     Usado por csetup() para escolher uma métrica
                  interativamente
setflags()   Usado por init_ctensor()
readvalue()
resimp()
sermet()     Usado por csetup() para informar uma métricacom série
                  de Taylor
txyzsum()
tmetric()    Referencial métrico, usado por cmetric() quando
                  cframe_flag:true
triemann()   Tensor de Riemann na base do referencial, usado quando
                  cframe_flag:true
tricci()     Tensor de Ricci na base do referencial, usado quando
                  cframe_flag:true
trrc()       Coeficientes de rotação de Ricci, usado por
                  christof()
yesp()
```

28.2.11 Modificações

Em Novembro de 2004, o pacote `ctensor` foi extensivamente reescrito. Muitas funções e variáveis foram renomeadas com o objectivo de tornar o pacote com a versão comercial do Macsyma.

```
Novo Nome    Nome Antigo     Descrição
-------------------------------------------------------------------
ctaylor()    DLGTAYLOR()     Expansão da série de Taylor de uma
-----------------------------expressão
lgeod[]      EM              Equações geodésicas
ein[]        G[]             Tensor misto de Einstein
ric[]        LR[]            Tensor misto de Ricci
ricci()      LRICCICOM()     Calcula o tensor misto de Ricci
```

ctaypov	MINP	Maximo expoente em expansões de séries de
------------------------------		Taylor
cgeodesic()	MOTION	Calcula as equações geodésicas
ct_coords	OMEGA	Coordenadas métricas
ctayvar	PARAM	Variável de expansão de séries de
------------------------------		Taylor
lriem[]	R[]	Tensor covariante de Riemann
uriemann()	RAISERIEMANN()	Calcula o tensor contravariante de
------------------------------		Riemann
ratriemann	RATRIEMAN	Simplificação racional do tensor de
------------------------------		Riemann
uric[]	RICCI[]	Tensor de Ricci contravariante
uricci()	RICCICOM()	Calcula o tensor de Ricci contravariante
cmetric()	SETMETRIC()	Escolhe a métrica
ctaypt	TAYPT	Ponto para expansões de séries de Taylor
ctayswitch	TAYSWITCH	Escolhe o comutador de séries de Taylor
csetup()	TSETUP()	Inicia sessão interativa de configuração
ctransform()	TTRANSFORM()	Transformação de coordenadas interativa
uriem[]	UR[]	Tensor contravariante de Riemann
weyl[]	W[]	Tensor (3,1) de Weyl

29 Pacote atensor

29.1 Introdução ao Pacote atensor

`atensor` é um pacote de manipulção de tensores algébricos. Para usar `atensor`, digite `load("atensor")`, seguido por uma chamada à função `init_atensor`.

A essência de `atensor` é um conjunto de regras de simplificação para o operador de produto (ponto) não comutativo (`"."`). `atensor` reconhece muitos tipos de álgebra; as regras de simplificação correspondentes são activadas quando a função `init_atensor` é chamada.

A compatibilidade de `atensor` pode ser demonstrada pela definição da álgebra de quaterniões como uma álgebra de Clifford Cl(0,2) com dois vectores fundamentais. As três unidades quaterniónicas imaginárias fundamentais são então os dois vectores base e seu produto, i.e.:

```
i = v     j = v     k = v  .  v
     1         2         1    2
```

Embora o pacote `atensor` tenha uma definição interna para a álgebra dos quaterniões, isso não foi usado nesse exemplo, no qual nós nos esforçamos para construir a tabela de multiplicação dos quaterniões como uma matriz:

```
(%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` reconhece como bases vectoriais símbolos indexados, onde o símbolo é aquele armazenado em `asymbol` e o iíndice está entre 1 e `adim`. Para símbolos indexado, e somente para símbolos indexados, as formas bilineares `sf`, `af`, e `av` são avaliadas. A avaliação substitui os valores de `aform[i,j]` em lugar de `fun(v[i],v[j])` onde `v` representa o valor de `asymbol` e `fun` é ainda `af` ou `sf`; ou, isso substitui `v[aform[i,j]]` em lugar de `av(v[i],v[j])`.

Desnecessário dizer, as funções `sf`, `af` e `av` podem ser redefinidas.

Quando o pacote `atensor` é chamado, os seguintes sinalizadores são configurados:

```
dotscrules:true;
dotdistrib:true;
dotexptsimp:false;
```

Se quiser experimentar com uma álgebra não associativa, pode também considerar a configuração de `dotassoc` para `false`. Nesse caso, todavia, `atensimp` não stará sempre habilitado a obter as simplificações desejadas.

29.2 Definições para o Pacote atensor

`init_atensor` (*alg_type*, *opt_dims*) [Função]
`init_atensor` (*alg_type*) [Função]

Inicializa o pacote `atensor` com o tipo especificado de álgebra. *alg_type* pode ser um dos seguintes:

`universal`: A álgebra universal tendo regras não comutativas.

`grassmann`: A álgebra de Grassman é definida pela relação de comutação `u.v+v.u=0`.

`clifford`: A álgebra de Clifford é definida pela relação de comutação `u.v+v.u=-2*sf(u,v)` onde `sf` é a função valor-escalar simétrico. Para essa álgebra, *opt_dims* pode ser acima de três inteiros não negativos, representando o número de dimensões positivas, dimensões degeneradas, e dimensões negativas da álgebra, respectivamente. Se quaisquer valores *opt_dims* são fornecidos, `atensor` irá configurar os valores de `adim` e `aform` apropriadamente. Caso contrário, `adim` irá por padrão para 0 e `aform` não será definida.

`symmetric`: A álgebra simétrica é definida pela relação de comutação `u.v-v.u=0`.

`symplectic`: A álgebra simplética é definida pela relação de comutação `u.v-v.u=2*af(u,v)` onde `af` é uma função valor-escalar antisimétrica. Para a

álgebra simplética, *opt_dims* pode mais de dois inteiros não negativos, representando a dimensão não degenerada e e a dimensão degenerada, respectivamente. Se quaisquer valores *opt_dims* são fornecidos, `atensor` irá configurar os valores de `adim` e `aform` apropriadamente. Caso contrário, `adim` irá por padrão para 0 e `aform` não será definida.

`lie_envelop`: O invólucro da álgebra de Lie é definido pela relação de comutação `u.v-v.u=2*av(u,v)` onde `av` é uma função antisimétrica.

A função `init_atensor` também reconhece muitos tipos pré-definidos de álgebra:

`complex` implementa a álgebra de números complexos como a álgebra de Clifford Cl(0,1). A chamada `init_atensor(complex)` é equivalente a `init_atensor(clifford,0,0,1)`.

`quaternion` implementa a álgebra de quaterniões. A chamada `init_atensor(quaternion)` é equivalente a `init_atensor(clifford,0,0,2)`.

`pauli` implementa a álgebra de spinores de Pauli como a álgebra de Clifford Cl(3,0). Uma chamada a `init_atensor(pauli)` é equivalente a `init_atensor(clifford,3)`.

`dirac` implementa a álgebra de spinores de Dirac como a álgebra de Clifford Cl(3,1). Uma chamada a `init_atensor(dirac)` é equivalente a `init_atensor(clifford,3,0,1)`.

`atensimp` (*expr*) [Função]

Simplifica a expressão algébrica de tensores *expr* conforme as regras configuradas por uma chamada a `init_atensor`. Simplificações incluem aplicação recursiva de relações comutativas e resoluções de chamadas a `sf`, `af`, e `av` onde for aplicável. Uma salvaguarda é usada para garantir que a função sempre termine, mesmo para expressões complexas.

`alg_type` [Função]

O tipo de álgebra. Valores válidos sáo `universal`, `grassmann`, `clifford`, `symmetric`, `symplectic` e `lie_envelop`.

`adim` [Variável]

A dimensionalidade da álgebra. `atensor` usa o valor de `adim` para determinar se um objecto indexado é uma base vectorial válida. Veja `abasep`.

`aform` [Variável]

Valor por omissão para as formas bilineares `sf`, `af`, e `av`. O padrão é a matriz identidade `ident(3)`.

`asymbol` [Variável]

O símbolo para bases vectoriais.

`sf` (*u*, *v*) [Função]

É uma função escalar simétrica que é usada em relações comutativas. A implementação padrão verifica se ambos os argumentos são bases vectoriais usando `abasep` e se esse for o caso, substitui o valor correspondente da matriz `aform`.

`af` (*u*, *v*) [Função]

É uma função escalar antisimétrica que é usada em relações comutativas. A implementação padrão verifica se ambos os argumentos são bases vectoriais usando `abasep` e se esse for o caso, substitui o valor correspondente da matriz `aform`.

`av` (*u*, *v*) [Função]

É uma função antisimétrica que é usada em relações comutativas. A implementação padrão verifica se ambos os argumentos são bases vectoriais usando `abasep` e se esse for o caso, substitui o valor correspondente da matriz `aform`.

Por exemplo:

```
(%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*) [Função]

Verifica se esse argumento é uma base vectorial `atensor` .

E será, se ele for um símbolo indexado, com o símbolo sendo o mesmo que o valor de `asymbol`, e o índice tiver o mesmo valor numérico entre 1 e `adim`.

30 Séries

30.1 Introdução a Séries

Maxima contém funções `taylor` e `powerseries` (séries de potência) para encontrar as séries de funções diferenciáveis. Maxima também tem ferramentas tais como `nusum` capazes de encontrar a forma fechada de algumas séries. Operações tais como adição e multiplicação travalham da forma usual sobre séries. Essa secção apresenta as variáveis globais que controlam a expansão.

30.2 Definições para Séries

`cauchysum` [Variável de opção]

Valor por omissão: `false`

Quando multiplicando adições jutas com `inf` como seus limites superiores, se `sumexpand` for `true` e `cauchysum` for `true` então o produto de Cauchy será usado em lugar do produto usual. No produto de Cauchy o índice do somatório interno é uma função do índice do externo em lugar de variar independentemente.

Exemplo:

```
(%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*) [Função]

Para cada função *f_i* de uma variável *x_i*, `deftaylor` define *expr_i* como a séries de Taylor sobre zero. *expr_i* é tipicamente um polinómio em *x_i* ou um somatório; expressões mais gerais são aceitas por `deftaylor` sem reclamações.

`powerseries (f_i(x_i), x_i, 0)` retorna as séries definidas por `deftaylor`.

`deftaylor` retorna uma lista das funções *f_1*, ..., *f_n*. `deftaylor` avalia seus argumentos.

Exemplo:

```
(%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` [Variável de opção]

Valor por omissão: `true`

Quando `maxtayorder` for `true`, durante a manipulação algébrica de séries (truncadas) de Taylor, `taylor` tenta reter tantos termos quantos forem conhecidos serem correctos.

`niceindices (`*expr*`)` [Função]

Renomeia os índices de adições e produtos em *expr*. `niceindices` tenta renomear cada índice para o valor de `niceindicespref[1]`, a menos que o nome apareça nas parcelas do somatório ou produtório, nesses casos `niceindices` tenta os elementos seguintes de `niceindicespref` por sua vez, até que uma varável não usada unused variable seja encontrada. Se a lista inteira for exaurida, índices adicionais são constrídos através da anexaao de inteiros ao valor de `niceindicespref[1]`, e.g., `i0`, `i1`, `i2`, ...

`niceindices` retorna uma expressão. `niceindices` avalia seu argumento.

Exemplo:

```
(%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` [Variável de opção]

Valor por omissão: `[i, j, k, l, m, n]`

`niceindicespref` é a lista da qual `niceindices` obtém os nomes dos índices de adições e produtos products.

Os elementos de `niceindicespref` são tipicamente nomes de variáveis, embora que não seja imposto por `niceindices`.

Exemplo:

```
(%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*) [Função]

Realiza o somatório hipergeométrico indefinido de *expr* com relação a *x* usando um procedimento de decisão devido a R.W. Gosper. *expr* e o resultado deve ser expressável como produtos de expoentes inteiros, factoriais, binomios, e funções recionais.

Os termos "definido" and "e somatório indefinido" são usados analogamente a "definida" and "integração indefinida". Adicionar indefinidamente significa dar um resultado simólico para a adição sobre intervalos de comprimentos de variáveis, não apenas e.g. 0 a infinito. Dessa forma, uma vez que não existe fórmula para a adição parcial geral de séries binomiais, `nusum` não pode fazer isso.

`nusum` e `unsum` conhecem um porco sobre adições e subtrações de produtos finitos. Veja também `unsum`.

Exemplos:

```
(%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*) [Função]

Retorna uma lista de todas as funções racionais que possuem a dada expansão da séries de Taylor onde a adição dos graus do numerador e do denominador é menor que ou igual ao nível de truncação das séries de potência, i.e. são "melhores" aproximações, e que adicionalmente satisfazem o grau especificado associado.

taylor_series é uma séries de Taylor de uma variável. *numer_deg_bound* e *denom_deg_bound* são inteiros positivos especificando o grau associado sobre o numerador e o denominador.

taylor_series podem também ser séries de Laurent, e o grau associado pode ser `inf` que acarreta todas funções racionais cujo grau total for menor que ou igual ao comprimento das séries de potências a serem retornadas. O grau total é definido como *numer_deg_bound* + *denom_deg_bound*. O comprimento de séries de potência é definido como "nível de trncação" + 1 - min(0, "ordem das séries").

```
(%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)                           []
```

Não existe função racional de grau 4 numerador/denominador, com essa expansão de série de potência. Você obrigatoriamente em geral tem grau do numerador e grau do denominador adicionando para cima ao menor grau das séries de potência, com o objectivo de ter disponível coeficientes desconhecidos para resolver.

```
(%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` [Variável de opção]

Valor por omissão: `false`

Quando `powerdisp` for `true`, uma adição é mostrada com seus termos em ordem do crescimento do expoente. Dessa forma um polinómio é mostrado como séries de potências truncadas, com o termo constante primeiro e o maior expoente por último.

Por padão, termos de uma adição são mostrados em ordem do expoente decrescente.

powerseries (*expr*, *x*, a) [Função]

Retorna a forma geral expansão de séries de potência para *expr* na variável *x* sobre o ponto *a* (o qual pode ser `inf` para infinito).

Se `powerseries` incapaz de expandir *expr*, `taylor` pode dar os primeiros muitos termos de séries.

Quando `verbose` for `true`, `powerseries` mostra mensagens de progresso.

```
(%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 [Variável de opção]

Valor por omissão: `false`

Quando `psexpand` for `true`, uma expressão função racional extendida é mostrada completamente expandida. O comutador `ratexpand` tem o mesmo efeito.

Quando `psexpand` for `false`, uma expressão de várias variáveis é mostrada apenas como no pacote de função racional.

Quando `psexpand` for `multi`, então termos com o mesmo grau total nas variáveis são agrupados juntos.

revert (*expr*, *x*) [Função]

revert2 (*expr*, *x*, *n*) [Função]

Essas funções retornam a reversão de *expr*, uma série de Taylor sobre zero na variável *x*. `revert` retorna um polinómio de grau igual ao maior expoente em *expr*. `revert2`

retorna um polinómio de grau *n*, o qual pode ser maior que, igual a, ou menor que o grau de *expr*.

`load ("revert")` chama essas funções.

Exemplos:

```
(%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*`)` [Função]
`taylor (`*expr*`, [`*x_1*`, `*x_2*`, ...], a, `*n*`)` [Função]
`taylor (`*expr*`, [`*x*`, a, `*n*`, 'asymp])` [Função]
`taylor (`*expr*`, [`*x_1*`, `*x_2*`, ...], [a_1, a_2, ...], [`*n_1*`, `*n_2*`, ...])` [Função]
`taylor (`*expr*`, [`*x_1*`, a_1, `*n_1*`], [`*x_2*`, a_2, `*n_2*`], ...)` [Função]

`taylor (`*expr*`, `*x*`, a, `*n*`)` expande a expressão *expr* em uma série truncada de Taylor ou de Laurent na variável *x* em torno do ponto *a*, contendo termos até `(`*x*` - a)^`*n*.

Se *expr* é da forma `f(`*x*`)/g(`*x*`)` e `g(`*x*`)` não possui de grau acima do grau *n* então `taylor` tenta expandir `g(`*x*`)` acima do gau `2` *n*. Se existe ainda termos não zero, `taylor` dobra o grau de expansão de `g(`*x*`)` contanto que o grau da expansão o grau da expansão seja menor que ou igual a *n* `2^taylordepth`.

`taylor (`*expr*`, [`*x_1*`, `*x_2*`, ...], a, `*n*`)` retorna uma série de potência truncada de grau *n* em todas as variáveis *x_1*, *x_2*, ... sobre o ponto `(a, a, ...)`.

`taylor (expr, [x_1, a_1, n_1], [x_2, a_2, n_2], ...)` retorna uma série de potência truncada nas variáveis *x_1*, *x_2*, ... sobre o ponto `(a_1, a_2, ...)`, truncada em *n_1*, *n_2*, ...

`taylor (expr, [x_1, x_2, ...], [a_1, a_2, ...], [n_1, n_2, ...])` retorna uma série de potência truncada nas variáveis *x_1*, *x_2*, ... sobre o ponto `(a_1, a_2, ...)`, truncada em *n_1*, *n_2*, ...

`taylor (expr, [x, a, n, 'asymp])` retorna uma expansão de *expr* em expoentes negativos de `x - a`. O termo de maior ordem é `(x - a)^-n`.

Quando `maxtayorder` for `true`, então durante maniplulação algébrica da séries de Taylor (truncada), `taylor` tenta reter tantos termos quantos forem conhecidos serem correctos.

Quando `psexpand` for `true`, uma expressão de função racional extendida é mostrada completamente expandida. O comutador `ratexpand` tem o mesmo efeito. Quando `psexpand` for `false`, uma expressão de várias variáveis é mostrada apenas como no pacote de função racional. Quando `psexpand` for `multi`, então os termos com o mesmo grau total nas variáveis são agrupados juntos.

Veja também o comutador `taylor_logexpand` para controlar a expansão.

Exemplos:

```
(%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` [Variável de opção]

Valor por omissão: 3

Se existem ainda termos não zero, `taylor` dobra o grau da expansão de *g*(*x*) contanto que o grau da expansão seja menor que ou igual a *n* `2^taylordepth`.

`taylorinfo` (*expr*) [Função]

Retorna information about the séries de Taylor *expr*. O valor de retorno é uma lista de listas. Cada lista compreende o nome de uma variável, o ponto de expansão, e o grau da expansão.

`taylorinfo` retorna `false` se *expr* não for uma séries de Taylor.

Exemplo:

```
(%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*) [Função]
Retorna `true` se *expr* for uma séries de Taylor, e `false` de outra forma.

`taylor_logexpand` [Variável de opção]
Valor por omissão: `true`

`taylor_logexpand` controla expansão de logaritmos em séries de `taylor`.

Quando `taylor_logexpand` for `true`, todos logaritmos são expandidos completamente dessa forma problemas de reconhecimento de zero envolvendo envolvendo identidades logarítmicas não atrapalham o processo de expansão. Todavia, esse esquema não é sempre matematicamente correcto uma vez que isso ignora informações de ramo.

Quando `taylor_logexpand` for escolhida para `false`, então a expansão logarítmica que ocorre é somente aquela que for necessária para obter uma séries de potência formal.

`taylor_order_coefficients` [Variável de opção]
Valor por omissão: `true`

`taylor_order_coefficients` controla a ordenação dos coeficientes em uma série de Taylor.

Quando `taylor_order_coefficients` for `true`, coeficientes da séries de Taylor são ordenados canonicamente.

`taylor_simplifier` (*expr*) [Função]
Simplifica coeficientes da séries de potência *expr*. `taylor` chama essa função.

`taylor_truncate_polynomials` [Variável de opção]
Valor por omissão: `true`

Quando `taylor_truncate_polynomials` for `true`, polinómios são truncados baseados sobre a entrada de níveis de truncação.

De outra forma, entrada de polinómios para `taylor` são consideradas terem precisão infinita.

`taytorat` (*expr*) [Função]
Converte *expr* da forma `taylor` para a forma de expressão racional canónica (CRE). O efeito é o mesmo que `rat (ratdisrep (`*expr*`))`, mas mais rápido.

`trunc` (*expr*) [Função]

Coloca notas na representação interna da expressão geral *expr* de modo que isso é mostrado como se suas adições forem séries de Taylor truncadas. *expr* is not otherwise modified.

Exemplo:

```
(%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*) [Função]

Retorna a primeira diferençã de trás para frente `f(n) - f(n - 1)`. Dessa forma `unsum` logicamente é a inversa de `sum`.

Veja também `nusum`.

Exemplos:

```
(%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` [Variável de opção]

Valor por omissão: `false`

Quando `verbose` for `true`, `powerseries` mostra mensagens de progresso.

31 Teoria dos Números

31.1 Definições para Teoria dos Números

`bern` (*n*) [Função]

Retorna o *n*'ésimo número de Bernoulli para o inteiro *n*. Números de Bernoulli iguais a zero são suprimidos se `zerobern` for `false`.

Veja também `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*) [Função]

Retorna o *n*'ésimo polinómio de Bernoulli na variável *x*.

`bfzeta` (*s*, *n*) [Função]

Retorna a função zeta de Riemann para o argumento *s*. O valor de retorno é um grande inteiro em ponto flutuante (bfloat); *n* é o número de dígitos no valor de retorno.

`load ("bffac")` chama essa função.

`bfhzeta` (*s*, *h*, *n*) [Função]

Retorna a função zeta de Hurwitz para os argumentos *s* e *h*. O valor de retorno é um grande inteiro em ponto flutuante (bfloat); *n* é o números de dígitos no valor de retorno.

A função zeta de Hurwitz é definida como

```
sum ((k+h)^-s, k, 0, inf)
```

`load ("bffac")` chama essa função.

`binomial` (*x*, *y*) [Função]

O coeficiente binomial `x!/(y! (x - y)!)`. Se *x* e *y* forem inteiros, então o valor numérico do coeficiente binomial é calculado. Se *y*, ou *x* - *y*, for um inteiro, o the coeficiente binomial é expresso como um polinómio.

Exemplos:

```
(%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*) [Função]

Retorna o *n*'ésimo número de Bernoulli para o inteiro *n*. `burn` pode ser mais eficitente que `bern` para valores grandes e isolados de *n* (talvez *n* maior que 105 ou algo parecido), como `bern` calcula todos os números de Bernoulli até o índice *n* antes de retornar.

`burn` explora a observação que números de Bernoulli (racionais) podem ser aproximados através de zetas (transcendentes) com eficiência tolerável.

`load ("bffac")` chama essa função.

`cf` (*expr*) [Função]

Converte *expr* em uma fração contínua. *expr* é uma expressão compreendendo frações contínuas e raízes quadradas de inteiros. Operandos na expressão podem ser combinados com operadores aritméticos. Com excessão de frações contínuas e raízes quadradas, factores na expressão devem ser números inteiros ou racionais. Maxima não conhece operações sobre frações contínuas fora de `cf`.

`cf` avalia seus argumentos após associar `listarith` a `false`. `cf` retorna uma fração contínua, representada como uma lista.

Uma fração contínua `a + 1/(b + 1/(c + ...))` é representada através da lista `[a, b, c, ...]`. Os elementos da lista `a`, `b`, `c`, ... devem avaliar para inteiros. *expr* pode também conter `sqrt (n)` onde `n` é um inteiro. Nesse caso `cf` fornecerá tantos termos de fração contínua quantos forem o valor da variável `cflength` vezes o período.

Uma fração contínua pode ser avaliada para um número através de avaliação da representação aritmética retornada por `cfdisrep`. Veja também `cfexpand` para outro caminho para avaliar uma fração contínua.

Veja também `cfdisrep`, `cfexpand`, e `cflength`.

Exemplos:

- *expr* é uma expressão compreendendo frações contínuas e raízes quadradas de inteiros.

```
(%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` controla quantos períodos de fração contínua são computados para números algébricos, números irracionais.

```
(%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]
```

- Um fração contínua pode ser avaliado através da avaliação da representação aritmética retornada por `cfdisrep`.

```
(%i1) cflength: 3$
(%i2) cfdisrep (cf (sqrt (3)))$
(%i3) ev (%, numer);
(%o3)                    1.731707317073171
```

- Maxima não conhece operações sobre frações contínuas fora de `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*) [Função]

Constrói e retorna uma expressão aritmética comum da forma `a + 1/(b + 1/(c + ...))` a partir da representação lista de uma fração contínua `[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*) [Função]

Retorna uma matriz de numeradores e denominadores dos último (columa 1) e penúltimo (columa 2) convergentes da fração contínua *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` [Variável de opção]

Valor por omissão: 1

`cflength` controla o número de termos da fração contínua que a função `cf` fornecerá, como o valor de `cflength` vezes o período. Dessa forma o padrão é fornecer um período.

```
(%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*`)` [Função]

`divsum (`*n*`)` [Função]

`divsum (`*n*`, `*k*`)` retorna a adição dos divisores de *n* elevados à *k*'ésima potência.

`divsum (`*n*`)` retorna a adição dos divisores de *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*`)` [Função]

Retorna o *n*'ésimo número de Euler para o inteiro *n* não negativo.

Para a constante de Euler-Mascheroni, veja `%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` [Constante]

A constante de Euler-Mascheroni, 0.5772156649015329 ....

`factorial (`*x*`)` [Função]

Representa a função factorial. Maxima trata `factorial (`*x*`)` da mesma forma que *x*`!`. Veja `!`.

`fib (`*n*`)` [Função]

Retorna o *n*'ésimo número de Fibonacci. `fib(0)` igual a 0 e `fib(1)` igual a 1, e `fib (-`*n*`)` igual a `(-1)^(`*n*` + 1) * fib(`*n*`)`.

Após chamar `fib`, `prevfib` é iguala `fib (`*x*` - 1)`, o número de Fibonacci anterior ao último calculado.

```
(%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*) [Função]

Expressa números de Fibonacci que aparecem em *expr* em termos da constante `%phi`, que é `(1 + sqrt(5))/2`, aproximadamente 1.61803399.

Exemplos:

```
(%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*) [Função]

Para um inteiro positivo *n* retorna a factoração de *n*. Se `n=p1^e1..pk^nk` for a decomposição de *n* em factores primos, `ifactors` retorna `[[p1, e1], ... , [pk, ek]]`.

Os métodos de factoração usados são divisões triviais por primos até 9973, o método rho de Pollard e o método da curva elíptica.

```
(%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*) [Função]

Retorna a parte inteira da *n*'ésima raíz do valor absoluto de *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*) [Função]

Calcula o inverso de *n* módulo *m*. `inv_mod (n,m)` retorna `false`, se *n* modulo *m* for zero.

```
(%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*) [Função]

Retorna símbolo de Jacobi de *p* e *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*) [Função]

Retorna o menor múltiplo comum entre seus argumentos. Os argumentos podem ser expressões gerais também inteiras.

`load ("functs")` chama essa função.

minfactorial (*expr*) [Função]

Examina *expr* procurando por ocorrências de dois factoriais que diferem por um inteiro. `minfactorial` então converte um em um polinómio vezes o outro.

```
(%i1) n!/(n+2)!;
                                 n!
(%o1)                         --------
                              (n + 2)!
(%i2) minfactorial (%);
                                  1
(%o2)                     ---------------
                          (n + 1) (n + 2)
```

next_prime (*n*) [Função]

Retorna o menor primo maior que *n*.

```
(%i1) next_prime(27);
(%o1)                         29
```

partfrac (*expr*, *var*) [Função]

Expande a expressão *expr* em frações parciais com relação à variável principal *var*. `partfrac` faz uma decomposição completa de fração parcial. O algoritmo utilizado é baseado no facto que os denominadores de uma expansão de fração parcial (os factores do denominador original) são relativamente primos. Os numeradores podem ser escritos como combinação linear dos denominadores, e a expansão acontece.

```
(%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*) [Função]

Usa um algoritmo modular para calcular `a^n mod m` onde *a* e *n* são inteiros e *m* é um inteiro positivo. Se *n* for negativo, `inv_mod` é usada para encontrar o inverso modular.

```
(%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*) [Função]

Teste de primalidade. Se `primep (n)` retornar `false`, *n* é um número compostro e se esse teste retornar `true`, *n* é um número primo com grande probabilidade.

Para *n* menor que 3317044064679887385961981 uma versão determinista do teste de Miller-Rabin é usada. Se `primep (n)` retornar `true`, então *n* é um número primo.

Para *n* maior que 34155071728321 `primep` usa `primep_number_of_tests` que é os testes de pseudo-primalidade de Miller-Rabin e um teste de pseudo-primalidade de Lucas. A probabilidade que *n* irá passar por um teste de Miller-Rabin é menor que 1/4. Usando o valor padrão 25 para `primep_number_of_tests`, a probabilidade de *n* passar no teste sendo composto é muito menor que 10^-15.

`primep_number_of_tests` [Variável de opção]

Valor por omissão: 25

Número de testes de Miller-Rabin usados em `primep`.

`prev_prime` (*n*) [Função]

Retorna o maior primo menor que *n*.

```
(%i1) prev_prime(27);
(%o1)                         23
```

`qunit` (*n*) [Função]

Retorna a principal unidade do campo dos números quadráticos reais `sqrt (n)` onde *n* é um inteiro, i.e., o elemento cuja norma é unidade. Isso é importante para resolver a equação de Pell `a^2 - n b^2 = 1`.

```
(%i1) qunit (17);
(%o1)                      sqrt(17) + 4
(%i2) expand (% * (sqrt(17) - 4));
(%o2)                            1
```

totient (*n*) [Função]

Retorna o número de inteiros menores que ou iguais a *n* que são relativamente primos com *n*.

zerobern [Variável de opção]

Valor por omissão: `true`

Quando `zerobern` for `false`, `bern` exclui os números de Bernoulli que forem iguais a zero. Veja `bern`.

zeta (*n*) [Função]

Retorna a função zeta de Riemann se *x* for um inteiro negativo, 0, 1, ou número par positivo, e retorna uma forma substantiva `zeta (n)` para todos os outros argumentos, incluindo não inteiros racionais, ponto flutuante, e argumentos complexos.

Veja também `bfzeta` e `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 [Variável de opção]

Valor por omissão: `true`

Quando `zeta%pi` for `true`, `zeta` retorna uma expressão proporcional a `%pi^n` para inteiro par `n`. De outra forma, `zeta` retorna uma forma substantiva `zeta (n)` para inteiro par `n`.

```
(%i1) zeta%pi: true$
(%i2) zeta (4);
                                   4
                                %pi
(%o2)                           ----
                                 90
(%i3) zeta%pi: false$
(%i4) zeta (4);
(%o4)                        zeta(4)
```

32 Simetrias

32.1 Definições para Simetrias

32.1.1 Mudando a base do sistema de numeração

`comp2pui` (*n*, *L*) [Função]

implementa a passagem das funções simétricas completamente simétricas fornecidas na lista *L* para as funções simétricas elementares de 0 a *n*. Se a lista *L* contiver menos que *n+1* elementos, será completada com valores formais do tipo *h1*, *h2*, etc. Se o primeiro elemento da lista *L* existir, ele é interpretado como sendo o tamanho do alfabeto, de outra forma o tamanho é escolhido para *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*) [Função]

vai de funções simétricas elementares para as funções completas. Similar a `comp2ele` e `comp2pui`.

Outras funções para mudanças de base: `comp2ele`.

`ele2comp` (*m*, *L*) [Função]

Vai de funções simétricas elementares para funções completas. Similar a `comp2ele` e a `comp2pui`.

Outras funções para mudanças de base: `comp2ele`.

`elem` (*ele*, *sym*, *lvar*) [Função]

ddecompões o polinómio simétrico *sym*, nas variáveis contidas na lista *lvar*, em termos de funções elementares simétricas fornecidas na lista *ele*. Se o primeiro elemento de *ele* for fornecido, esse primeiro elemento será o tamanho do alfabeto, de outra forma o tamanho será o grau do polinómio *sym*. Se valores forem omitidos na lista *ele*, valores formais do tipo *e1*, *e2*, etc. serão adicionados. O polinómio *sym* pode ser fornecido de três diferentes formas: contraída (`elem` pode então ser 1, seu valor padrão), particionada (`elem` pode ser 3), ou extendida (i.e. o polinómio completo, e `elem` pode então ser 2). A função `pui` é usada então da mesma forma.

sobre um alfabeto de tamanho 3 com *e1*, a primeira funçào elementar simétrica, com valor 7, o polinómio simétrico em 3 variáveis cuja forma contraída (que aqui depende de duas de suas variáveis) é *x^4-2*x*y* decomposto como segue em funções elementares simétricas:

```
(%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
```

Outras funções para mudanças de base: `comp2ele`.

`mon2schur` (*L*) [Function]

a lsita L representa a função de Schur S_L: temos $L = [i_1, i_2, \ldots, i_q]$, with $i_1 \leq i_2 \leq \ldots \leq i_q$. A função de Schur $S_{i_1,i_2,\ldots,i_q}$ é a menor da matriz infinita h_{i-j}, $i \geq 1, j \geq 1$, consistindo das q primeiras linhas e as colunas $i_1 + 1, i_2 + 2, \ldots, i_q + q$.

Essa função de Schur pode ser escrita em termos de monômios usando `treinat` e `kostka`. A forma retornada é um polinómio simétrico na representação contraída nas variáveis $x_1, x_2, \ldots$.

```
(%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
```

o qual significa que para 3 variáveis fornece:

```
2 x1 x2 x3 + x1^2 x2 + x2^2 x1 + x1^2 x3 + x3^2 x1
 + x2^2 x3 + x3^2 x2
```

Outras funções para mudanças de base: `comp2ele`.

`multi_elem` (*l_elem*, *multi_pc*, *l_var*) [Função]

decompões um polinómio multi-simétrico na forma multi-contraída *multi_pc* nos grupos de variáveis contidas na lista de listas *l_var* en termos de funções elementares simétricas contidas em *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
```

Outras funções para mudanças de base: `comp2ele`.

`multi_pui` [Função]

é para a função `pui` o que a função `multi_elem` é para a função `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*) [Função]

decompõe o polinómio simétrico *sym*, nas variáveis na lista *lvar*, em termos de funções exponenciais na lista L. Se o primeiro elemento de L for fornecido, esse primeiro elemento será o tamanho do alfabeto, de outra forma o tamanho será o grau do polinómio *sym*. Se valores forem omitidos na lista L, valores formais do tipo *p1*, *p2* , etc. serão adicionados. O polinómio *sym* pode ser fornecido de três diferentes formas:

contraída (`elem` pode então ser 1, seu valor padrão), particionada (`elem` pode ser 3), ou extendida (i.e. o polinómio completo, e `elem` pode então ser 2). A função `pui` é usada da mesma forma.

```
(%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
```

Outras funções para mudanças de base: `comp2ele`.

`pui2comp` (*n*, *lpui*) [Função]

converte a dista das primeiras *n* funções completas (com o comprimento em primeiro lugar) em termos de funções exponenciais fornecidas na lista *lpui*. se a lista *lpui* for vazia, o cardinal é *n*, de outra forma o cardinal será seu primeiro elemento (como em `comp2ele` e em `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
```

Outras funções para mudanças de base: `comp2ele`.

`pui2ele` (*n*, *lpui*) [Função]

efectiva a passagem de funções exponenciais para as funções elementares simétricas. Se o sinalizador `pui2ele` for `girard`, `pui2ele` irá retornar a lista de funções elementares simétricas de 1 a *n*, e se o sinalizador for `close`, `pui2ele` retornará a *n*-ésima função simétrica elementar.

Outras funções para mudanças de base: `comp2ele`.

puireduc (*n*, *lpui*) [Função]

lpui é uma lista cujo primeiro elemento é um inteiro *m*. `puireduc` fornece as primeiras *n* funções exponenciais em termos das primeiras *m* funções.

```
(%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*) [Função]

P é um polinómio nas variáveis da lista *l_var*. Cada uma dessas variáveis represetna uma função simétrica completa. Na lista *l_var* o *i*-ésima função simétrica completa é representada através da concatenação da letra `h` com o inteiro *i*: `hi`. Essa função expressa *P* em termos de funções de Schur.

```
(%i1) schur2comp (h1*h2 - h3, [h1, h2, h3]);
(%o1)                          s
                                1, 2
(%i2) schur2comp (a*h3, [h3]);
(%o2)                          s  a
                                3
```

32.1.2 Modificando represetnações

cont2part (*pc*, *lvar*) [Função]

Retorna o polinómio particionado associado à forma contraída *pc* cujas variáveis estão em *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*) [Função]

retorna uma forma contraída (i.e. um monômio de grupo ssimétrico) do polinómio *psym* nas variáveis contidas na lista *lvar*. A função `explose` executa a operação inversa. A função `tcontract` testa a simétria do polinómio.

```
(%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*) [Função]

retorna o polinómio simétrico associado com a forma contraída *pc*. A lista *lvar* conté as variáveis.

```
(%i1) explose (a*x + 1, [x, y, z]);
(%o1)                   a z + a y + a x + 1
```

`part2cont` (*ppart*, *lvar*) [Função]

vai da forma particionada para a forma contraída de um polinómio simétrico. A forma contraída é convertida com as variáveis em *lvar*.

```
(%i1) part2cont ([[2*a^3*b, 4, 1]], [x, y]);
                                    3    4
(%o1)                         2 a  b x  y
```

`partpol` (*psym*, *lvar*) [Função]

psym é um polinómio simétrico nas variáveis da lista *lvar*. Essa função retorna sua represetnação particionada.

```
(%i1) partpol (-a*(x + y) + 3*x*y, [x, y]);
(%o1)                [[3, 1, 1], [- a, 1, 0]]
```

`tcontract` (*pol*, *lvar*) [Função]

testa se o polinómio *pol* é simétrico nas variáveis da lista *lvar*. Se for, `tcontract` retorna uma representação contraída como o faz a função `contract`.

`tpartpol` (*pol*, *lvar*) [Função]

testa se o polinómio *pol* é simétrico nas variáveis da lista *lvar*. Se for, `tpartpol` retorna sua represetnação particionada como o faz a função `partpol`.

`direct` ([*p_1*, ..., *p_n*], *y*, *f*, [*lvar_1*, ..., *lvar_n*]) [Função]

calcula a imagem directa (see M. Giusti, D. Lazard et A. Valibouze, ISSAC 1988, Rome) associada à função *f*, na lista de variáveis *lvar_1*, ..., *lvar_n*, e nos polinómios *p_1*, ..., *p_n* na variável *y*. A quantidade de argumetnos que a funçào *f* pode receber é importante para o cálculo. Dessa forma, se a expressão para *f* não depende de alguma variável, é inútil incluir essa variável, e não incluir essa variável irá também reduzir consideravelmente o montante cálculos efetuados.

```
(%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
```

Encontrando um polinómio cujas raízes são somatórios $a + u$ onde a é uma raíz de $z^2 - e_1 z + e_2$ e u é uma raíz de $z^2 - + f_1 z + f_2$.

```
(%i1) ratsimp (direct ([z^2 - e1* z + e2, z^2 - f1* z + f2],
                        z, a + u, [[u], [a]]));
       4                   3            2
(%o1) y  + (- 2 f1 - 2 e1) y  + (2 f2 + f1  + 3 e1 f1 + 2 e2

     2   2                              2              2
 + e1 ) y  + ((- 2 f1 - 2 e1) f2 - e1 f1  + (- 2 e2 - e1 ) f1

                  2                     2           2
 - 2 e1 e2) y + f2  + (e1 f1 - 2 e2 + e1 ) f2 + e2 f1  + e1 e2 f1

    2
 + e2
```

`direct` aceita dois sinalizadores: `elementaires` (elementares) e `puissances` (exponenciais - valor padrão) que permitem a decomposição de polinómios simétricos que aparecerem nesses cálculos em funções simétricas elementares ou em funções exponenciais respectivamente.

Funções de `sym` utilizadas nesta função :

`multi_orbit` (portanto `orbit`), `pui_direct`, `multi_elem` (portanto `elem`), `multi_pui` (portanto `pui`), `pui2ele`, `ele2pui` (se o sinalizador `direct` for escolhido para `puissances`).

`multi_orbit` (*P*, [*lvar_1*, *lvar_2*,..., *lvar_p*]) [Função]

P é um polinómio no conjunto de variáveis contidas nas lista *lvar_1*, *lvar_2*, ..., *lvar_p*. Essa função retorna a órbita do polinómio *P* sob a ação do produto dos grupos simétricos dos conjuntos de variáveis represetnadas nas *p* listas.

```
(%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]
```

Veja também: `orbit` para a ação de um grupo simétrico simples.

`multsym` (*ppart_1*, *ppart_2*, *n*) [Função]

retorna oproduto de dois polinómios simétricos em *n* varieis trabalhando somente módulo a ação do grupo simétrico de ordem *n*. O polinómios estão em sua forma particionada.

Dados 2 polinómio simétricos em *x*, *y*: `3*(x + y) + 2*x*y` e `5*(x^2 + y^2)` cujas formas particionadas são `[[3, 1], [2, 1, 1]]` e `[[5, 2]]`, seu produto irá ser

```
(%i1) multsym ([[3, 1], [2, 1, 1]], [[5, 2]], 2);
(%o1)         [[10, 3, 1], [15, 3, 0], [15, 2, 1]]
```

isso é `10*(x^3*y + y^3*x) + 15*(x^2*y + y^2*x) + 15*(x^3 + y^3)`.

Funções para mudar as representacões de um polinómio simétrico:

`contract`, `cont2part`, `explose`, `part2cont`, `partpol`, `tcontract`, `tpartpol`.

`orbit` (*P*, *lvar*) [Função]

calcula a órbita do polinómio *P* nas variáveis na lista *lvar* sob a ação do grupo simétrico do conjunto das variáveis na lista *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]
```

Veja também `multi_orbit` para a ação de um produto de grupos simétricos sobre um polinómio.

`pui_direct` (*orbite*, [*lvar_1*, ..., *lvar_n*], [*d_1*, *d_2*, ..., *d_n*]) [Função]

Tomemos *f* para ser um polinómio em *n* blocos de variáveis *lvar_1*, ..., *lvar_n*. Façamos *c_i* ser o n;umero de variáveis em *lvar_i*, e *SC* ser o produto de *n* grupos simétricos de grau *c_1*, ..., *c_n*. Essas ações dos grupos naturalmente sobre *f*. A lista *orbite*

é a órbita, denotada `SC(f)`, da função f sob a ação de SC. (Essa lista pode ser obtida através da função `multi_orbit`.) Os di são inteiros de forma que $c_1 \leq d_1, c_2 \leq d_2, \ldots, c_n \leq d_n$.

Tomemos SD para ser o produto dos grupos simétricos $S_{d_1} \times +S_{d_2} \times \cdots \times S_{d_n}$. A função `pui_direct` retorna as primeiras n funções exponenciais de `SD(f)` deduzidas das funções exponenciais de `SC(f)`, onde n é o tamanho de `SD(f)`.

O resultado está na multi-forma contraída com relação a SD, i.e. somente um elemento é mantido por órbita, sob a ação de 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             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 ]
```

32.1.3 Partições

`kostka` (*part_1*, *part_2*) [Função]

escrita por P. Esperet, calcula o número de Kostka da partição *part_1* e *part_2*.

```
(%i1) kostka ([3, 3, 3], [2, 2, 2, 1, 1, 1]);
(%o1)                           6
```

`lgtreillis` (*n*, *m*) [Função]

retorna a lista de partições de peso n e comprimento m.

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

Veja também: `ltreillis`, `treillis` e `treinat`.

`ltreillis` (*n*, *m*) [Função]

retorna a lista de partições de peso n e comprimento menor que ou igual a m.

```
(%i1) ltreillis (4, 2);
(%o1)                [[4, 0], [3, 1], [2, 2]]
```

Veja também: `lgtreillis`, `treillis` e `treinat`.

`treillis (n)` [Função]

retorna todas as partições de peso n.

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

Veja também: `lgtreillis`, `ltreillis` e `treinat`.

`treinat (part)` [Função]

retorna a lista de partições inferiores à partiçào *part* com relação à ordem natural.

```
(%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]]
```

Outras funções de mudança de representação :

Veja também: `lgtreillis`, `ltreillis` e `treillis`.

32.1.4 Polinómios e suas raízes

`ele2polynome (L, z)` [Função]

retorna o polinómio em z de forma que as funções elementares simétricas de suas raízes estejam na lista `L = [n, e_1, ..., e_n]`, onde n é o grau dos polinómios e *e_i* é a i-ésima função simétrica elementar.

```
(%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
```

o inverso: `polynome2ele (P, z)`.

Veja também: `polynome2ele`, `pui2polynome`.

`polynome2ele (P, x)` [Função]

fornece a lista `l = [n, e_1, ..., e_n]` onde n é o grau do polinómio P na variável x e *e_i* é a i-ésima função simétrica elementar das raízes de 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
```

A inversa: `ele2polynome (l, x)`

`prodrac (L, k)` [Função]

L é uma lista contendo as funções simétricas elementares sobre um conjunto *A*. `prodrac` retorna o polinómio cujas raízes são os produtos *k* por *k* dos elementos de *A*.

Veja também `somrac`.

`pui2polynome (x, lpui)` [Função]

calcula o polinómio em *x* cujas funções exponenciais das raízes são dadas na lista *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
```

Veja também: `polynome2ele`, `ele2polynome`.

`somrac (L, k)` [Função]

A lista *L* contains função simétrica elementars de um polynomial *P* . The function computes the polinómio whose roots are the *k* by *k* distinct sums of the roots of *P*.

Also see `prodrac`.

32.1.5 Resolvents

`resolvante (P, x, f, [x_1,..., x_d])` [Função]

calculates the resolvent of the polinómio *P* in *x* of degree *n* >= *d* by the function *f* expressed nas variáveis *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*.

Para melhorar a eficiência do cálculo se pode escolher sinalizadores em `resolvante` de fora a usar os algoritmos apropriados:

Se a função *f* for unitária :

- um polinómio em uma variável simples,
- linear ,
- alternado,
- um somatório,
- simétrico,

- um produto,
- a função da resolvente de Cayley (utilisável de grau 5 em diante)

```
(x1*x2 + x2*x3 + x3*x4 + x4*x5 + x5*x1 -
     (x1*x3 + x3*x5 + x5*x2 + x2*x4 + x4*x1))^2
```

  geral,

o sinalizador da `resolvante` poderá ser respectivamente :

- 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, -

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, []);

" resolvente de Cayley "
```

```
        6       5         4          3           2
(%o21) x  - 40 x  + 4080 x  - 92928 x  + 3772160 x  + 37880832 x

                                                          + 93392896
```

Para a resolvente de Cayley, os 2 últimos argumentos são neutros e o polinómio fornecido na entrada deve ser necessáriamente de grau 5.

Veja também :

`resolvante_bipartite`, `resolvante_produit_sym`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`.

`resolvante_alternee1` (*P*, *x*) [Função]

calcula a transformação de `P(x)` de grau *n* pela função $\prod_{1\leq i<j\leq n-1}(x_i-x_j)$. $\prod_{1\leq i<j\leq n-1}(x_i - x_j)$.

Veja também :

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante` , `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`, `resolvante_bipartite`.

`resolvante_bipartite` (*P*, *x*) [Função]

calcula a trasformação de `P(x)` de mesmo grau *n* através da função $x_1x_2\cdots x_{n/2} + x_{n/2+1}\cdots x_n$.

Veja também :

`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
```

Veja também :

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante`, `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`, `resolvante_alternee1`.

`resolvante_diedrale` (*P*, *x*) [Função]

+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
```

Veja também :

resolvante_produit_sym, resolvante_unitaire, resolvante_alternee1, resolvante_klein, resolvante_klein3, resolvante_vierer, resolvante.

resolvante_klein (*P*, *x*) [Função]

+calculates the transformation of *P*(*x*) by the function +*x_1 x_2 x_4 + x_4*.

Veja também :

resolvante_produit_sym, resolvante_unitaire, resolvante_alternee1, resolvante, resolvante_klein3, resolvante_vierer, resolvante_diedrale.

resolvante_klein3 (*P*, *x*) [Função]

calcula a transformação de *P*(*x*) através da função *x_1 x_2 x_4 + x_4*.

Veja também :

resolvante_produit_sym, resolvante_unitaire, resolvante_alternee1, resolvante_klein, resolvante, resolvante_vierer, resolvante_diedrale.

resolvante_produit_sym (*P*, *x*) [Função]

calcula a lista de todas as resolventes de produto do polinómio *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]);

" resolvente produto "
       10      8       7    6        5       4       3     2
(%o3) y   + 3 y  + 14 y  - y  - 14 y  - 31 y  - 21 y  - 2 y  + 1
```

Veja também :

resolvante, resolvante_unitaire, resolvante_alternee1, resolvante_klein, resolvante_klein3, resolvante_vierer, resolvante_diedrale.

resolvante_unitaire (*P*, *Q*, *x*) [Função]

+computes the resolvent of the polinómio *P*(*x*) by the +polynomial *Q*(*x*).

Veja também :

resolvante_produit_sym, resolvante, resolvante_alternee1, resolvante_klein, resolvante_klein3, resolvante_vierer, resolvante_diedrale.

resolvante_vierer (*P*, *x*) [Função]

calcula a transformação de *P*(*x*) pela função *x_1 x_2 - x_3 x_4*.

Veja também :

resolvante_produit_sym, resolvante_unitaire, resolvante_alternee1, resolvante_klein, resolvante_klein3, resolvante, resolvante_diedrale.

`multinomial` (*r*, *part*) [Função]

onde *r* é o peso da partição *part*. Essa função retorna o coefinciente multinomial associado: se as partes de *part* forem *i_1*, *i_2*, ..., *i_k*, o resultado é `r!/(i_1! i_2! ... i_k!)`.

`permut` (*L*) [Função]

retorna a lista de permutações da lista *L*.

`tcontract` (*pol*, *lvar*) [Função]

testa se o polinómio *pol* é simétrico nas variáveis contidas na lista *lvar*. se for é rtornado uma forma contraída da forma retornada pela função `contract`.

Outras funções de mudança de representação :

`contract`, `cont2part`, `explose`, `part2cont`, `partpol`, `tpartpol`.

`tpartpol` (*pol*, *lvar*) [Função]

testa se o polinómio *pol* é simétrico nas variáveis contidas na lista *lvar*. Se for simétrico `tpartpol` produz a forma particionada como a função `partpol`.

Outras funções de mudança de representação :

`contract`, `cont2part`, `explose`, `part2cont`, `partpol`, `tcontract`.

`treillis` (*n*) [Função]

retorna todas as partições de peso *n*.

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

Veja também : `lgtreillis`, `ltreillis` e `treinat`.

`treinat` (*part*) [Função]

retorna a lista das partições inferiores à partição *part* pela ordem natural.

```
(%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]]
```

Veja também : `lgtreillis`, `ltreillis` e `treillis`.

33 Grupos

33.1 Definições para Grupos

`todd_coxeter` (*relação*, *subgroupo*) [Função]
`todd_coxeter` (*relação*) [Função]

Acha a ordem de G/H onde G é o módulo do Grupo Livre *relação*, e H é o subgroupo de G gerado por *subgroupo*. *subgroupo* é um argumento opcional, cujo valor padrão é []. Em fazendo isso a função produz uma tabela de multiplicação à direita de G sobre G/H, onde os co-conjuntos são enumerados [H,Hg2,Hg3,...]. Isso pode ser visto internamente no `todd_coxeter_state`.

Exemplo:

```
(%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 Ambiente em Tempo de Execução

34.1 Introdução a Ambiente em Tempo de Execução

`maxima-init.mac` é um ficheiro que é chamado automaticamente quando o Maxima inicia. Pode usar `maxima-init.mac` para personalizar o seu ambiente no Maxima. `maxima-init.mac`, se existir, é tipicamente colocado no directório chamado por `maxima_userdir`, embora possa estar em qualquer outro directório procurado pela função `file_search`.

Aqui está um exemplo do ficheiro `maxima-init.mac`:

```
setup_autoload ("specfun.mac", ultraspherical, assoc_legendre_p);
showtime:all;
```

Nesse Exemplo, `setup_autoload` diz ao Maxima para chamar o ficheiro especificado (`specfun.mac`) se qualquer das funções (`ultraspherical`, `assoc_legendre_p`) forem chamadas sem estarem definidas. Dessa forma, não precisa de se lembrar de chamar o ficheiro antes das funções.

A declaração `showtime: all` diz ao Maxima para activar a opção `showtime`. O ficheiro `maxima-init.mac` pode conter quaisquer outras atribuições ou outras declarações do Maxima.

34.2 Interrupções

O utilizador pode parar uma computação que consome muito tempo com o caractere ^C (control-C). A ação padrão é parar a computação e mostrar outra linha de comando do utilizador. Nesse caso, não é possível continuar a computação interrompida.

Se a variável `*debugger-hook*` é escolhida para `nil`, através do comando

```
:lisp (setq *debugger-hook* nil)
```

então na ocasião do recebimento do ^C, Maxima iniciará o depurador Lisp, e o utilizador pode usar o depurador para inspecionar o ambiente Lisp. A computação interrompida pode ser retomada através do comando `continue` no depurador Lisp. O método de retorno para ao Maxima partindo do depurador Lisp (outro como executando a computação para complementação) é diferente para cada versão do Lisp.

Em sistemas Unix, o caratere ^Z (control-Z) faz com que Maxima pare tudo e aguarde em segundo plano, e o controle é retornado para a linha de comando do shell. O comando `fg` faz com que o Maxima retorne ao primeiro plano e continue a partir do ponto no qual foi interrompido.

34.3 Definições para Ambiente em Tempo de Execução

`feature` [Declaração]

Maxima compreende dois tipos distintos de recurso, recursos do sistema e recursos aplicados a expressões matemáticas. Veja Também `status` para informações sobre recursos do sistema. Veja Também `features` e `featurep` para informações sobre recursos matemáticos.

`feature` por si mesmo não é o nome de uma função ou variável.

`featurep` (*a*, *f*) [Função]

Tenta determinar se o objecto *a* tem o recurso *f* na base dos factos dentro base de dados corrente. Se possue, é retornado `true`, de outra forma é retornado `false`.

Note que `featurep` retorna `false` quando nem *f* nem a negação de *f* puderem ser estabelecidas.

`featurep` avalia seus argumentos.

Veja também `declare` e `features`.

```
(%i1) declare (j, even)$
(%i2) featurep (j, integer);
(%o2)                          true
```

`maxima_tempdir` [Variável de sistema]

`maxima_tempdir` nomeia o directório no qual Maxima cria alguns ficheiros temporários. Em particular, ficheiros temporários para impressão são criados no `maxima_tempdir`.

O valor inicial de `maxima_tempdir` é o directório do utilizador, se o maxima puder localizá-lo; de outra forma Maxima supõe um directório adequado.

A `maxima_tempdir` pode ser atribuído uma sequência de caracteres que corresponde a um directório.

`maxima_userdir` [Variável de sistema]

`maxima_userdir` nomeia um directório no qual Maxima espera encontrar seus próprios ficheiros e os do ficheiros do Lisp. (Maxima procura em alguns outros directórios também; `file_search_maxima` e `file_search_lisp` possuem a lista completa.)

O valor inicial de `maxima_userdir` é um subdirectório do directório do utilizador, se Maxima puder localizá-lo; de outra forma Maxima supõe um directório adequado.

A `maxima_userdir` pode ser atribuído uma sequência de caracteres que corresponde a um directório. Todavia, fazendo uma atribuição a `maxima_userdir` não muda automaticamente o valor de `file_search_maxima` e de `file_search_lisp`; Essas variáveis devem ser modificadas separadamente.

`room` () [Função]
`room` (*true*) [Função]
`room` (*false*) [Função]

Mostra uma descrição do estado de armazenamento e gerenciamento de pilha no Maxima. `room` chama a função Lisp de mesmo nome.

- `room ()` mostra uma descrição moderada.
- `room (true)` mostra uma descrição detalhada.
- `room (false)` mostra uma descrição resumida.

`status` (*feature*) [Função]
`status` (*feature*, *recurso_ativo*) [Função]
`status` (*status*) [Função]

Retorna informações sobre a presença ou ausência de certos recursos dependentes do sistema operacional.

- `status (feature)` retorna uma lista dos recursos do sistema. Inclui a versão do Lisp, tipo de sistema operacional, etc. A lista pode variar de um tipo de Lisp para outro.
- `status (feature,` *recurso_ativo*`)` retorna `true` se *recurso_ativo* está na lista de ítens retornada através de `status (feature)` e `false` de outra forma. `status` não avalia o argumento *recurso_ativo*. O operador apóstrofo-apóstrofo, `''`, evita a avaliação. Um recurso cujo nome contém um caractere especial, tal como um hífem, deve ser fornecido como um argumento em forma de sequência de caracteres. Por Exemplo, `status (feature, "ansi-cl")`.
- `status (status)` retorna uma lista de dois elementos `[feature, status]`. `feature` e `status` são dois argumentos aceitos pela função `status`; Não está claro se essa lista tem significância adicional.

A variável `features` contém uma lista de recursos que se aplicam a expressões matemáticas. Veja `features` e `featurep` para maiores informações.

`time` (*%o1, %o2, %o3, ...*) [Função]

Retorna uma lista de tempos, em segundos, usados para calcular as linhas de saída `%o1`, `%o2`, `%o3`, ... O tempo retornado é uma estimativa do Maxima do tempo interno de computação, não do tempo decorrido. `time` pode somente ser aplicado a variáveis(rótulos) de saída de linha; para quaisquer outras variáveis, `time` retorna `unknown` (tempo desconhecido).

Escolha `showtime: true` para fazer com que Maxima moste o tempo de computação e o tempo decorrido a cada linha de saída.

`timedate ()` [Função]

Retorna uma sequência de caracteres representando a data e hora atuais. A sequência de caracteres tem o formato `HH:MM:SS Dia, mm/dd/aaaa (GMT-n)`, Onde os campos são horas, minutos, segundos, dia da semana, mês, dia do mês, ano, e horas que diferem da hora GMT.

O valor de retorno é uma sequência de caracteres Lisp.

Exemplo:

```
(%i1) d: timedate ();
(%o1) 08:05:09 Wed, 11/02/2005 (GMT-7)
(%i2) print ("timedate mostra o tempo actual", d)$
timedate reports current time 08:05:09 Wed, 11/02/2005 (GMT-7)
```

35 Opções Diversas

35.1 Introdução a Opções Diversas

Nessa secção várias opções são tratadas pelo facto de possuirem um efeito global sobre a operação do Maxima. Também várias listas tais como a lista de todas as funções definidas pelo utilizador, são discutidas.

35.2 Compartilhado

O directório "share" do Maxima contém programas e outros ficheiros de interesse para os utilizadors do Maxima, mas que não são parte da implementação do núcleo do Maxima. Esses programas são tipicamente chamados via `load` ou `setup_autoload`.

`:lisp *maxima-sharedir*` mostra a localização do directório compartilhado dentro do sistema de ficheiros do utilizador.

`printfile ("share.usg")` imprime uma lista de pacotes desactualizados dos pacotes compartilhados. Usuários podem encontrar isso de forma mais detalhada navegando no directório compartilhado usando um navegador de sistema de ficheiro.

35.3 Definições para Opções Diversas

`aliases` [Variável de sistema]

Valor por omissão: `[]`

`aliases` é a lista de átomos que possuem um alias definido pelo utilizador (escolhido através das funções `alias`, `ordergreat`, `orderless` ou através da declaração do átomo como sendo um `noun` (substantivo) com `declare`).

`alphabetic` [Declaração]

`alphabetic` é uma declaração reconhecida por `declare`. A expressão `declare(`*s*`, alphabetic)` diz ao Maxima para reconhecer como alfabético todos os caracteres em *s*, que deve ser uma sequência de caracteres.

Veja também *Identificadores*.

Exemplo:

```
(%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*`)` [Função]

Procura por nomes Maxima que possuem *string* aparecendo em qualquer lugar dentro de seu nome. Dessa forma, `apropos (exp)` retorna uma lista de todos os sinalizadores e funções que possuem `exp` como parte de seus nomes, tais como `expand`, `exp`, e

`exponentialize`. Dessa forma, se lembrar apenas uma parte do nome de alguma coisa, pode usar este comando para achar o restante do nome. Similarmente, pode dizer `apropos (tr_)` para achar uma lista de muitos dos comutadores relatando para o tradutor, muitos dos quais começam com `tr_`.

`args` (*expr*) [Função]

Retorna a lista de argumentos de `expr`, que pode ser de qualquer tipo de expressão outra como um átomo. Somente os argumentos do operador de nível mais alto são extraídos; subexpressões de `expr` aparecem como elementos ou subexpressões de elementos da lista de argumentos.

A ordem dos ítens na lista pode depender do sinalizador global `inflag`.

`args (`*expr*`)` é equivalente a `substpart ("[", `*expr*`, 0)`. Veja também `substpart`.

Veja também `op`.

`genindex` [Variável de opção]

Valor por omissão: `i`

`genindex` é o prefixo usado para gerar a próxima variável do somatório quando necessário.

`gensumnum` [Variável de opção]

Valor por omissão: 0

`gensumnum` é o sufixo numérico usado para gerar variável seguinte do somatório. Se isso for escolhido para `false` então o índice consistirá somente de `genindex` com um sufixo numérico.

`inf` [Constante]

Infinito positivo real.

`infinity` [Constante]

Infinito complexo, uma magnitude infinita de ângulo de fase arbitrária. Veja também `inf` e `minf`.

`infolists` [Variável de sistema]

Valor por omissão: `[]`

`infolists` é uma lista dos nomes de todas as listas de informação no Maxima. São elas:

`labels` Todos associam `%i`, `%o`, e rótulos `%t`.

`values` Todos associam átomos que são variáveis de utilizador, não opções do Maxima ou comutadores, criados através de `:` ou `::` ou associando funcionalmente.

`functions`
Todas as funções definidas pelo utilizador, criadas através de `:=` ou `define`.

`arrays` Todos os arrays declarados e não declarados, criados através de `:`, `::`, ou `:=`.

`macros` Todas as macros definidas pelo utilizador.

`myoptions`
Todas as opções alguma vez alteradas pelo utilizador (mesmo que tenham ou não elas tenham mais tarde retornadas para seus valores padrão).

`rules` Todos os modelos definidos pelo utilizador que coincidirem e regras de simplificação, criadas através de `tellsimp`, `tellsimpafter`, `defmatch`, ou `defrule`.

`aliases` Todos os átomos que possuem um alias definido pelo utilizador, criado através das funções `alias`, `ordergreat`, `orderless` ou declarando os átomos como um `noun` com `declare`.

`dependencies`
Todos os átomos que possuem dependências funcionais, criadas através das funções `depends` ou `gradef`.

`gradefs` Todas as funções que possuem derivadas definidas pelo utilizador, cridas através da função `gradef`.

`props` Todos os átomos que possuem quaisquer propriedades outras que não essas mencionadas acima, tais como propriedades estabelecidas por `atvalue` , `matchdeclare`, etc., também propriedades estabelecidas na função `declare`.

`let_rule_packages`
Todos os pacote de régras em uso definidos pelo utilizador mais o pacote especial `default_let_rule_package`. (`default_let_rule_package` é o nome do pacote de régras usado quando um não está explicitamente escolhido pelo utilizador.)

`integerp` (*expr*) [Função]
Retorna `true` se *expr* é um inteiro numérico literal, de outra forma retorna `false`.

`integerp` retorna falso se seu argumento for um símbolo, mesmo se o argumento for declarado inteiro.

Exemplos:

```
(%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` [Variável de opção]

Valor por omissão: `false`

`m1pbranch` é principal descendente de `-1` a um expoente. Quantidades tais como `(-1)^(1/3)` (isto é, um expoente racional "ímpar") e `(-1)^(1/4)` (isto é, um expoente racional "par") são manuseados como segue:

```
                 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*) [Função]

Retorna `true` se *expr* for um inteiro literal, número racional, número em ponto flutuante, ou um grande número em ponto flutuante, de outra forma retorna `false`.

`numberp` retorna falso se seu argumento for um símbolo, mesmo se o argumento for um número simbólico tal como `%pi` ou `%i`, ou declarado ser par, ímpar, inteiro, racional, irracional, real, imaginário, ou complexo.

Exemplos:

```
(%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`) [Função]

Retorna uma lista de nomes de todas as propriedades associadas com o átomo *a*.

`props` [Símbolo especial]

`props` são átomos que possuem qualquer propriedade outra como essas explicitamente mencionadas em `infolists`, tais como atvalues, matchdeclares, etc., também propriedades especificadas na função `declare`.

`propvars` (*prop*) [Função]

Retorna uma lista desses átomos sobre a lista `props` que possui a propriedade indicada através de *prop*. Dessa forma `propvars (atvalue)` retorna uma lista de átomos que possuem atvalues.

`put` (*átomo*, *valor*, *indicador*) [Função]

Atribui *valor* para a propriedade (especificada através de *indicador*) do *átomo*. *indicador* pode ser o nome de qualquer propriedade, não apenas uma propriedade definida pelo sistema.

`put` avalia seus argumentos. `put` retorna *valor*.

Exemplos:

```
(%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` (*átomo*, *valor*, *indicador*) [Função]

Atribui *valor* para a propriedade (especificada através de *indicador*) do *átomo*. Isso é o mesmo que `put`, excepto que os argumentos nã são avaliados.

Exemplo:

```
(%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` (*átomo*, *indicador*) [Função]

Remove a propriedade indicada através de *indicador* do *átomo*.

`remove` (`a_1`, `p_1`, ..., `a_n`, `p_n`) [Função]
`remove` ([`a_1`, ..., `a_m`], [`p_1`, ..., `p_n`], ...) [Função]
`remove` ("`a`", *operator*) [Função]
`remove` (`a`, *transfun*) [Função]
`remove` (*all*, `p`) [Função]

Remove propriedades associadas a átomos.

`remove (a_1, p_1, ..., a_n, p_n)` remove a propriedade `p_k` do átomo `a_k`.

`remove ([a_1, ..., a_m], [p_1, ..., p_n], ...)` remove as propriedades `p_1`, `...`, `p_n` dos átomos *a_1*, ..., *a_m*. Pode existir mais que um par de listas.

`remove (all, p)` remove a propriedade *p* de todos os átomos que a possuem.

A propriedade removida pode ser definida pelo sistema tal como `function`, `macro` ou `mode_declare`, ou propriedades definidas pelo utilizador.

uma propriedade pode ser `transfun` para remover a versão traduzida Lisp de uma função. Após executar isso, a versão Maxima da função é executada em lugar da versão traduzida.

`remove ("a", operator)` ou, equivalentemente, `remove ("a", op)` remove de *a* as propriedades `operator` declaradas através de `prefix`, `infix`, `nary`, `postfix`, `matchfix`, ou `nofix`. Note que o nome do operador deve ser escrito como uma sequência de caracteres com apóstofo.

`remove` sempre retorna `done` se um átomo possui ou não uma propriedade especificada. Esse comportamento é diferente das funções remove mais específicas `remvalue`, `remarray`, `remfunction`, e `remrule`.

`remvalue` (*nome_1*, ..., *nome_n*) [Função]
`remvalue` (*all*) [Função]

Remove os valores de Variáveis de utilizador *nome_1*, ..., *nome_n* (que podem ser subscritas) do sistema.

`remvalue (all)` remove os valores de todas as variáveis em `values`, a lista de todas as variáveis nomeadas através do utilizador (em oposição a essas que são automaticamente atribuídas através do Maxima).

Veja também `values`.

`rncombine` (*expr*) [Função]

Transforma *expr* combinando todos os termos de *expr* que possuem denominadores idênticos ou denominadores que diferem de cada um dos outros apenas por factores numéricos somente. Isso é ligeiramente diferente do comportamento de de `combine`, que colecta termos que possuem denominadores idênticos.

Escolhendo `pfeformat: true` e usando `combine` retorna resultados similares a esses que podem ser obtidos com `rncombine`, mas `rncombine` pega o passo adicional de multiplicar cruzado factores numérios do denominador. Esses resultados em forma ideal, e a possibilidade de reconhecer alguns cancelamentos.

Para usar essa função escreva primeiramente `load("rncomb")`.

`scalarp` (*expr*) [Função]

Retorna `true` se *expr* for um número, constante, ou variável declarada `scalar` com `declare`, ou composta inteiramente de números, constantes, e tais Variáveis, bmas não contendo matrizes ou listas.

`setup_autoload` (*nomeficheiro*, *função_1*, ..., *função_n*) [Função]

Especifica que se qualquer entre *função_1*, ..., *função_n* for referenciado e não ainda definido, *nomedeqrquivo* é chamado via `load`. *nomeficheiro* usualmente contém definições para as funções especificadas, embora isso não seja obrigatório.

`setup_autoload` não trabalha para funções array.

`setup_autoload` não avalia seus argumentos.

Exemplo:

```
(%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 função ultraspherical
Warning - you are redefining the Macsyma função 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 Regras e Modelos

36.1 Introdução a Regras e Modelos

Essa secção descreve coincidências de modelos definidos pelo utilizador e regras de simplificação. Existem dois grupos de funções que implementam até certo ponto diferentes esquemas de coincidência de modelo. Em um grupo estão `tellsimp`, `tellsimpafter`, `defmatch`, `defrule`, `apply1`, `applyb1`, e `apply2`. Em outro grupo estão `let` e `letsimp`. Ambos os esquemas definem modelos em termos de variáveis de modelo declaradas por `matchdeclare`.

Regras de coincidência de modelos definidas por `tellsimp` e `tellsimpafter` são aplicadas automaticamente através do simplificador do Maxima. Regras definidas através de `defmatch`, `defrule`, e `let` são aplicadas através de uma chamada explícita de função.

Existe mecanismos adicionais para regras aplicadas a polinómios através de `tellrat`, e para álgebra comutativa e não comutativa no pacote `affine`.

36.2 Definições para Regras e Modelos

`apply1` (*expr*, *rule_1*, ..., *rule_n*) [Função]

Repetidamente aplica *rule_1* a *expr* até que isso falhe, então repetidamente aplica a mesma regra a todas as subexpressões de *expr*, da esquerda para a direita, até que *rule_1* tenha falhado sobre todas as subexpressões. Chama o resultado da transformação de *expr* dessa maneira de *expr_2*. Então *rule_2* é aplicada no mesmo estilo iniciando no topo de *expr_2*. Quando *rule_n* falhar na subexpressão final, o resultado é retornado.

`maxapplydepth` é a intensidade de nível mais distante de subexpressões processadas por `apply1` e `apply2`.

Veja também `applyb1`, `apply2`, e `let`.

`apply2` (*expr*, *rule_1*, ..., *rule_n*) [Função]

Se *rule_1* falhar sobre uma dada subexpressão, então *rule_2* é repetidamente aplicada, etc. Somente se todas as regras falharem sobre uma dada subexpressão é que o conjunto completo de regras é repetidamente aplicada à próxima subexpressão. Se uma das regras obtém sucesso, então a mesma subexpressão é reprocessada, iniciando com a primeira regra.

`maxapplydepth` é a intensidade do nível mais distante de subexpressões processadas através de `apply1` e `apply2`.

Veja também `apply1` e `let`.

`applyb1` (*expr*, *rule_1*, ..., *rule_n*) [Função]

Repetidamente aplica *rule_1* para a subexpressão mais distante de *expr* até falhar, então repetidamente aplica a mesma regra um nível mais acima (i.e., subexpressãos mais larga), até que *rule_1* tenha falhado sobre a expressão de nível mais alto. Então *rule_2* é aplicada com o mesmo estilo para o resultado de *rule_1*. após *rule_n* ter sido aplicada à expressão de nível mais elevado, o resultado é retornado.

`applyb1` é similar a `apply1` mas trabalha da base para cima em lugar de do topo para baixo.

`maxapplyheight` é o ápice que `applyb1` encontra antes de interromper.

Veja também `apply1`, `apply2`, e `let`.

`current_let_rule_package` [Variável de opção]

Valor por omissão: `default_let_rule_package`

`current_let_rule_package` é o nome do pacote de regras que está sendo usado por funções no pacote `let` (`letsimp`, etc.) se nenhum outro pacote de regras for especificado. A essa variável pode ser atribuído o nome de qualquer pacote de regras definido via comando `let`.

Se uma chamada tal como `letsimp (expr, nome_pct_regras)` for feita, o pacote de regras `nome_pct_regras` é usado para aquela chamada de função somente, e o valor de `current_let_rule_package` não é alterado.

`default_let_rule_package` [Variável de opção]

Valor por omissão: `default_let_rule_package`

`default_let_rule_package` é o nome do pacote de regras usado quando um não for explicitamente escolhido pelo utilizador com `let` ou através de alteração do valor de `current_let_rule_package`.

`defmatch` (*prognome*, *modelo*, *x_1*, ..., *x_n*) [Função]
`defmatch` (*prognome*, *modelo*) [Função]

Define uma função *prognome*(*expr*, *x_1*, ..., *x_n*) que testa *expr* para ver se essa expressão coincide com *modelo*.

modelo é uma expressão contendo os argumentos modelo *x_1*, ..., *x_n* (se existir algum) e alguns modelos de variáveis (se existir algum). os argumentos modelo são fornecidos explicitamente como argumentos para `defmatch` enquanto os modelos de variáveis são declarados através da função `matchdeclare`. Qualquer variável não declarada como modelo em `matchdeclare` ou como um argumento modelo em `defmatch` coincide somente com si mesma.

O primeiro argumento para a função criada *prognome* é uma expressão a serem comparadas contra o modelo e os outros argumentos são os atuais argumetnos que correspondem às variáveis respectivas *x_1*, ..., *x_n* no modelo.

Se a tentativa de coincidência obtiver sucesso, *prognome* retorna uma lista de equações cujos lados esquerdos são os argumetnos de modelo e variáveis de modelo, e cujo lado direito forem as subexpressões cujos argumentos de modelo e as variáveis coincidem. Os modelos de variáveis, mas não tos argumentos de modelo, são atribuídos às subexpressões que coincidirem. Se a coincidência falhar, *prognome* retorna `false`.

Um modelo literal (isto é, um modelo que não contiver nem argumentos de modelo nem variáveis de modelo) retorna `true` se a coincidência ocorrer.

Veja também `matchdeclare`, `defrule`, `tellsimp`, e `tellsimpafter`.

Exemplos:

Define uma função `linearp(expr, x)` que testa `expr` para ver se essa expressão da forma `a*x + b` tal que `a` e `b` não contenham `x` e `a` seja não nulo. Essa função de coincidência coincide com expressões que sejam lineares em qualquer variável, por que o argumento de modelo `x` é fornecido para `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 uma função `linearp(expr)` que testa `expr` para ver se essa expressão é da forma `a*x + b` tal que `a` e `b` não contenham `x` e `a` seja não nulo. Essa função de coincidência somente coincide com expressões lineares em `x`, não em qualquer outra variável, porque nenhum argumento de modelo é fornecido a `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 uma função `checklimits(expr)` que testa `expr` para ver se essa expressão é uma integral definida.

```
(%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 (*nomeregra*, *modelo*, *substituição*) [Função]

Define e nomeia uma regra de substituição para o modelo dado. Se a regra nomeada *nomeregra* for aplicada a uma expressão (através de `apply1`, `applyb1`, ou `apply2`), toda subexpressão coincidindo com o modelo será substituída por `substituição`. Todas as variáveis em `substituição` que tiverem sido atribuidos valores pela coincidência com o modelo são atribuidas esses valores na `substituição` que é então simplificado.

As regras por si mesmas podem ser tratadas como funções que transforma uma expressão através de uma operação de coincidência de modelo e substituição. Se a coincidência falhar, a função da regra retorna `false`.

disprule (*nomeregra_1*, ..., *nomeregra_2*) [Função]

disprule (*all*) [Função]

Mostra regras com os nomes *nomeregra_1*, ..., *nomeregra_n*, como retornado por `defrule`, `tellsimp`, ou `tellsimpafter`, ou um modelo definido por meio de `defmatch`.

Cada regra é mostrada com um rótulo de expressão intermédia (`%t`).

`disprule (all)` mostra todas as regras.

`disprule` não avalia seus argumentos.

`disprule` retorna a lista de rótulos de expressões intermedáirias correspondendo às regras mostradas.

Veja também `letrules`, que mostra regras definidas através de `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*, *prednome*, *arg_1*, ..., *arg_n*) [Função]
`let` ([*prod*, *repl*, *prednome*, *arg_1*, ..., *arg_n*], *nome_pacote*) [Função]

Define uma regra de substituição para `letsimp` tal que *prod* é substituído por *repl*. *prod* é um produto de expoentes positivos ou negativos dos seguintes termos:

- Atomos que `letsimp` irá procurar literalmente a menos que previamente chamando `letsimp` a função `matchdeclare` é usada para associar um predicado com o átomo. Nesse caso `letsimp` irá coincidir com o átomo para qualquer termo de um produto satisfazendo o predicado.
- Núcleos tais como `sin(x)`, `n!`, `f(x,y)`, etc. Como com átomos acima `letsimp` irá olhar um literal coincidente a menos que `matchdeclare` seja usada para associar um predicado com o argumento do núcleo.

Um termo para um expoente positivo irá somente coincidir com um termo tendo ao menos aquele expoente. Um termo para um expoente negativo por outro lado irá somente coincidir com um termo com um expoente ao menos já negativo. o caso de expentes negativos em *prod* o comutador `letrat` deve ser escolhido para `true`. Veja também `letrat`.

Se um predicado for incluído na função `let` seguido por uma lista de argumentos, uma tentativa de coincidência (i.e. uma que pode ser aceita se o predicado fosse omitido) é aceita somente se `prednome (arg_1', ..., arg_n')` avaliar para `true` onde *arg_i'* é o valor coincidente com *arg_i*. O *arg_i* pode ser o nome de qualquer átomo ou o argumento de qualquer núcleo aparecendo em *prod*. *repl* pode ser qualquer expressão racional. Se quaisquer dos átomos ou argumentos de *prod* aparecer em *repl* a substituição é feita.

O sinalizador global `letrat` controla a simplificação dos quocientes através de `letsimp`. Quando `letrat` for `false`, `letsimp` simplifica o numerador e o denominador de *expr* separadamente, e não simplifica o quociente. Substituições tais como `n!/n` vão para `(n-1)!` então falham quando `letrat` for `false`. Quando `letrat` for `true`, então o numerador, o denominador, e o quociente são simplificados nessa ordem.

Essas funções de substituição permitem-lhe trabalhar com muitos pacotes de regras. Cada pacote de regras pode conter qualquer número de regras `let` e é referenciado através de um nome definido pelo utilizador. `let ([prod, repl, prednome, arg_1, ..., arg_n], nome_pacote)` adiciona a regra *prednome* ao pacote de regras *nome_pacote*. `letsimp (expr, nome_pacote)` aplica as regras em *nome_pacote*. `letsimp (expr, nome_pacote1, nome_pacote2, ...)` é equivalente a `letsimp (expr, nome_pacote1)` seguido por `letsimp (%, nome_pacote2)`, ...

`current_let_rule_package` é o nome do pacote de regras que está actualmente sendo usando. Essa variável pode receber o nome de qualquer pacote de regras definidos via o comando `let`. Quando qualquer das funções compreendidas no pacote `let` são chamadas sem o nome do pacote, o pacote nomeado por `current_let_rule_package` é usado. Se uma chamada tal como `letsimp (expr, nome_pct_regras)` é feita, o pacote de regras *nome_pct_regras* é usado somente para aquele comando `letsimp`, e `current_let_rule_package` não é alterada. Se não especificado de outra forma,

`current_let_rule_package` avalia de forma padronizada para `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` [Variável de opção]

Valor por omissão: `false`

Quando `letrat` for `false`, `letsimp` simplifica o numerador e o denominador de uma razão separadamente, e não simplifica o quociente.

Quando `letrat` for `true`, o numerador, o denominador, e seu quocienten são simplificados nessa ordem.

```
(%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 ()` [Função]

`letrules (`*nome_pacote*`)` [Função]

Mostra as regras em um pacote de regras. `letrules ()` mostra as regras no pacote de regras corrente. `letrules (`*nome_pacote*`)` mostra as regras em `nome_pacote`.

O pacote de regras corrente é nomeado por `current_let_rule_package`. Se não especificado de outra forma, `current_let_rule_package` avalia de forma padrão para `default_let_rule_package`.

Veja também `disprule`, que mostra regras definidas por `tellsimp` e `tellsimpafter`.

`letsimp (`*`expr`*`)` [Função]
`letsimp (`*`expr`*`, `*`nome_pacote`*`)` [Função]
`letsimp (`*`expr`*`, `*`nome_pacote_1`*`, ..., `*`nome_pacote_n`*`)` [Função]

Repetidamente aplica a substituição definida por `let` até que nenhuma mudança adicional seja feita para *expr*.

`letsimp (`*`expr`*`)` usa as regras de `current_let_rule_package`.

`letsimp (`*`expr`*`, `*`nome_pacote`*`)` usa as regras de *nome_pacote* sem alterar `current_let_rule_package`.

`letsimp (`*`expr`*`, `*`nome_pacote_1`*`, ..., `*`nome_pacote_n`*`)` é equivalente a `letsimp (`*`expr`*`, `*`nome_pacote_1`*`)`, seguido por `letsimp (%, `*`nome_pacote_2`*`)`, e assim sucessivamente.

`let_rule_packages` [Variável de opção]

Valor por omissão: `[default_let_rule_package]`

`let_rule_packages` é uma lista de todos os pacotes de regras `let` definidos pelo utilizador mais o pacote padrão `default_let_rule_package`.

`matchdeclare (`*`a_1`*`, `*`pred_1`*`, ..., `*`a_n`*`, `*`pred_n`*`)` [Função]

Associa um predicado *pred_k* com uma variável ou lista de variáveis *a_k* de forma que *a_k* coincida com expressões para as quais o predicado retorne qualquer coisa que não `false`.

Umpredicado é o nome de uma função, ou de uma expressão lambda, ou uma chamada de função ou chamada de função lambda iomitindo o úlltimo argumento, ou `true` ou `all`. Qualquer expressão coincide com `true` ou `all`. Se o predicado for especificado como uma chamada de função ou chamada de função lambda, a expressão a ser testada é anexada ao final da lista de argumentos; os argumentos são avaliados ao mesmo tempo que a coincidência é avaliada. De outra forma, o predicado é especificado como um nome de função ou expressão lambda, e a expressão a ser testada é o argumento sozinho. Uma função predicado não precisa ser definida quando `matchdeclare` for chamada; o predicado não é avaliado até que uma coincidência seja tentada.

Um predicado pode retornar uma expressão Booleana além de `true` ou `false`. Expressões Booleanas são avaliadas por `is` dentro da função da regra construída, de forma que não é necessário chamar `is` dentro do predicado.

Se uma expressão satisfaz uma coincidência de predicado, a variável de coincidência é atribuída à expressão, excepto para variáveis de coincidência que são operandos de adição `+` ou multiplicação `*`. Somente adição e multiplicação são manuseadas de forma especial; outros operadores enários (ambos os definidos internamente e os definidos pelo utilizador) são tratados como funções comuns.

No caso de adição e multiplicação, a variável de coincidência pode ser atribuida a uma expressão simples que satisfaz o predicado de coincidência, ou uma adição ou um produto (respectivamente) de tais expressões. Tal coincidência de termo multiplo é gulosa: predicados são avaliados na ordem em que suas variáveis associadas aparecem no modelo de coincidência, e o termo que satisfizer mais que um predicado é tomado pelo primeiro predicado que satisfizer. Cada predicado é testado contra todos

os operandos de adição ou produto antes que o próximo predicado seja avaliado. Adicionalmente, se 0 ou 1 (respectivamente) satisfazem um predicado de coincidência, e não existe outros termos que satisfaçam o predicado, 0 ou 1 é atribuído para a variável de coincidência associada com o predicado.

O algoritmo para processar modelos contendo adição e multiplicação faz alguns resultados de coincidência (por exemplo, um modelo no qual uma variável "coincida com qualquer coisa" aparecer) dependerem da ordem dos termos no modelo de coincidência e na expressão a ser testada a coincidência. Todavia, se todos os predicados de coincidência são mutuamente exclusivos, o resultado de coincidência é insensível a ordenação, como um predicado de coincidência não pode aceitar termos de coincidência de outro.

Chamado `matchdeclare` com uma variável *a* como um argumento muda a propriedade `matchdeclare` para *a*, se a variável *a* tiver sido declarada anteriormente; somente o `matchdeclare` mais recente está em efeito quando uma regra é definida, mudanças posteriores para a propriedade `matchdeclare` (via `matchdeclare` ou `remove`) não afectam regras existentes.

`propvars (matchdeclare)` retorna a lista de todas as variáveis para as quais exista uma propriedade `matchdeclare`. `printprops (a, matchdeclare)` retorna o predicado para a variável `a`. `printprops (all, matchdeclare)` retorna a lista de predicados para todas as variáveis `matchdeclare`. `remove (a, matchdeclare)` remove a propriedade `matchdeclare` da variável *a*.

As funções `defmatch`, `defrule`, `tellsimp`, `tellsimpafter`, e `let` constroem regras que testam expressões contra modelos.

`matchdeclare` coloca apóstrofo em seus argumentos. `matchdeclare` sempre retorna `done`.

Exemplos:

Um predicado é o nome de uma função, ou uma expressão lambda, ou uma chamada de função ou chamada a função lambda omitindo o último argumento, 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
```

Se uma expressão satisfaz um predicado de coincidência, a variável de coincidência é atribuída à expressão.

```
(%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]
```

No caso de adição e multiplicação, à variável de coincidência pode ser atribuída uma expressão simples que satisfaz o predicado de coincidência, ou um somatório ou produtório (respectivamente) de tais expressões.

```
(%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)]
```

Quando coincidindo argumentos de `+` e `*`, se todos os predicados de coincidência forem mutuamente exclusivos, o resultado da coincidência é insensíve à ordenação, como um predicado de coincidência não pode aceitar termos que coincidiram com outro.

```
(%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
```

As funções `propvars` e `printprops` retornam informações sobre variáveis de coincidência.

```
(%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` (*delimitador_e*, *delimitador_d*) [Função]
`matchfix` (*delimitador_e*, *delimitador_d*, *arg_pos*, *pos*) [Função]

Declara um operador `matchfix` com delimitadores esquerdo e direito *delimitador_e* e *delimitador_d*. Os delimitadores são especificados como sequências de caracteres.

Um operador "matchfix" é uma função que aceita qualquer número de argumentos, tal que os argumentos ocorram entre os delimitadores correspondentes esquerdo e direito. Os delimitadores podem ser quaisquer sequências de caracteres, contanto que o analisador de expressões do Maxima possa distinguir os delimitadores dos operandos e de outras expressões e operadores. Na prática essas regras excluem delimitadores não analisáveis tais como `%`, `,`, `$` e `;`, e pode ser necessário isolar os delimitadores com espaços em branco. O delimitador da direita pode ser o mesmo ou diferente do delimitador da esquerda.

Um delimitador esquerdo pode ser associado com somente um delimitador direito; dois diferentes operadores `matchfix` não podem ter o mesmo delimitador esquerdo.

Um operador existente pode ser redeclarado com um operador `matchfix` sem alterar suas outras propriedades. Particularmente, operadores internos tais como adição `+` podem ser declarados `matchfix`, mas funções operadores não podem ser definidas para operadores internos.

`matchfix (`*delimitador_e*`,` *delimitador_d*`,` *arg_pos*`,` *pos*`)` declara o argumento *arg_pos* como sendo um entre: expressão lógica, expressão comum do Maxima mas que não seja do tipo anterior, e qualquer outro tipo de expressão que não esteja incluída nos dois primeiros tipos. Essa declaração resulta em *pos* sendo um entre: expressão lógica, expressão comum do Maxima mas que não seja do tipo anterior, e qualquer outro tipo de expressão que não esteja incluída nos dois primeiros tipos e os delimitadores *delimitador_e* e *delimitador_d*.

A função para realizar uma operação `matchfix` é uma função comum definida pelo utilizador. A função operador é definida da forma usual com o operador de definição de função `:=` ou `define`. Os argumentos podem ser escritos entre os delimitadores, ou com o delimitador esquerdo com uma sequência de caracteres com apóstrofo e os argumentos seguindo entre parêntesis. `dispfun (`*delimitador_e*`)` mostra a definição da função operador.

O único operador interno `matchfix` é o construtor de listas `[ ]`. Parêntesis `( )` e aspas duplas `" "` atuam como operadores `matchfix`, mas não são tratados como tal pelo analisador do Maxima.

`matchfix` avalia seus argumentos. `matchfix` retorna seu primeiro argumento, *delimitador_e*.

Exemplos:

- Delimitadores podem ser quase quaisquer sequência de caracteres.

```
(%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~
```

- Operadores `matchfix` são funções comuns definidas pelo utilizador.

```
(%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*, *nome*) [Função]
remlet () [Função]
remlet (*all*) [Função]
remlet (*all*, *nome*) [Função]

Apaga a regra de substituiçao, prod –> repl, mais recentemente definida através dea função `let`. Se `nome` for fornecido a regra é apagada do pacote de regras chamado `nome`.

`remlet()` e `remlet(all)` apagam todas as regras de substituição do pacote de regras corrente. Se o nome de um pacote de regras for fornecido, e.g. `remlet (all, nome)`, o pacote de regras *nome* é também apagado.

Se uma substituição é para ser mudada usando o mesmo produto, `remlet` não precisa ser chamada, apenas redefina a substituição usando o mesmo produto (literalmente) com a função `let` e a nova substituição e/ou nome de predicado. Pode agora `remlet (prod)` ser chamada e a regra de substituição original é ressuscitada.

Veja também `remrule`, que remove uma regra definida através de `tellsimp` ou de `tellsimpafter`.

remrule (*op*, *nomeregra*) [Função]
remrule (*op*, *all*) [Função]

Remove regras definidas por `tellsimp`, ou `tellsimpafter`.

`remrule (op, nomeregra)` remove a regra com o nome `nomeregra` do operador *op*. Quando *op* for um operador interno ou um operador definido pelo utilizador (como definido por `infix`, `prefix`, etc.), *op* e *rulename* devem ser colocados entre aspas duplas.

`remrule (op, all)` remove todas as regras para o operador *op*.

Veja também `remlet`, que remove uma regra definida através de `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*) [Função]

é similar a `tellsimpafter` mas coloca nova informação antes da antiga de forma que essa nova regra seja aplicada antes das regras de simplificação internas.

`tellsimp` é usada quando for importante modificar a expressão antes que o simplificador trabalhe sobre ela, por exemplo se o simplificador "sabe" alguma coisa sobre a expressão, mas o que ele retorna não é para sua apreciação. Se o simplificador "sabe" alguma coisa sobre o principal operador da expressão, mas está simplesmente a escondê-lo, provavelmente querrá usar `tellsimpafter`.

O modelo pode não ser uma adição, um produto, variável simples, ou número.

`rules` é a lista de regras definidas por `defrule`, `defmatch`, `tellsimp`, e `tellsimpafter`.

Exemplos:

```
(%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` (*modelo*, *substituição*) [Função]

Define a uma regra de simplificação que o simplificador do Maxima aplica após as regras de simplificação internas. *modelo* é uma expressão, compreendendo variáveis de modelo (declaradas através de `matchdeclare`) e outros átomos e operações, considerados literais para o propósito de coincidência de modelos. *substituição* é substituída para uma expressão actual que coincide com *modelo*; variáveis de modelo em *substituição* são atribuidas a valores coincidentes na expressão actual.

modelo pode ser qualquer expressão não atômica na qual o principal operador não é uma variável de modelo; a regra de simplificação está associada com o operador principal. Os nomes de funções (com uma excessão, descrita abaixo), listas, e arrays podem aparecer em *modelo* como o principal operador somente como literais (não variáveis de modelo); essas regras fornecem expressões tais como `aa(x)` e `bb[y]` como modelos, se `aa` e `bb` forem variáveis de modelo. Nomes de funções, listas, e arrays que são variáveis de modelo podem aparecer como operadores outros que não o operador principal em *modelo*.

Existe uma excessão para o que foi dito acima com relação a regras e nomes de funções. O nome de uma função subscrita em uma expressão tal como `aa[x](y)` pode ser uma variável de modelo, porque o operador principal não é `aa` mas ao contrário o átomo Lisp `mqapply`. Isso é uma consequência da representação de expressões envolvendo funções subscritas.

Regras de simplificação são aplicadas após avaliação (se não suprimida através de colocação de apóstrofo ou do sinalizador `noeval`). Regras estabelecidas por `tellsimpafter` são aplicadas na ordem em que forem definidas, e após quaisquer regras internas. Regras são aplicadas de baixo para cima, isto é, aplicadas primeiro a subexpressões antes de ser aplicada à expressão completa. Isso pode ser necessário para repetidamente simplificar um resultado (por exemplo, via o operador apóstrofo-apóstrofo `''` ou o sinalizador `infeval`) para garantir que todas as regras são aplicadas.

Variáveis de modelo são tratadas como variáveis locais em regras de simplificação. Assim que uma regra é definida, o valor de uma variável de modelo não afecta a regra, e não é afectado pela regra. Uma atribuição para uma variável de modelo que resulta em uma coincidência de regra com sucesso não afecta a atribuição corrente (ou necessita disso) da variável de modelo. Todavia, como com todos os átomos no Maxima, as propriedades de variáveis de modelo (como declarado por `put` e funções relacionadas) são globais.

A regra construída por `tellsimpafter` é nomeada após o operador principal de `modelo`. Regras para operadores internos, e operadores definidos pelo utilizador definidos por meio de `infix`, `prefix`, `postfix`, `matchfix`, e `nofix`, possuem nomes que são sequências de caracteres do Maxima. Regras para outras funções possuem nomes que são identificadores comuns do Maxima.

O tratamento de substantivos e formas verbais é desprezívelmente confuso. Se uma regra é definida para uma forma substantiva (ou verbal) e uma regra para o verbo correspondente (ou substantivo) já existe, então a nova regra definida aplica-se a ambas as formas (substantiva e verbal). Se uma regra para a correspondente forma verbal (ou substantiva) não existe, a nova regra definida aplicar-se-á somente para a forma substantiva (ou verbal).

A regra construída através de `tellsimpafter` é uma função Lisp comum. Se o nome da regra for `$foorule1`, a construção `:lisp (trace $foorule1)` rastreia a função, e `:lisp (symbol-function '$foorule1` mostra sua definição.

`tellsimpafter` não avalia seus argumentos. `tellsimpafter` retorna a lista de regras para o operador principal de *modelo*, incluindo a mais recente regra estabelecia.

Veja também `matchdeclare`, `defmatch`, `defrule`, `tellsimp`, `let`, `kill`, `remrule`, e `clear_rules`.

Exemplos:

modelo pode ser qualquer expressão não atômica na qual o principal operador não é uma variável de modelo.

```
(%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))
```

Regras são aplicadas na ordem em que forem definidas. Se duas regras podem coincidir com uma expressão, a regra que foi primeiro definida é a que será aplicada.

```
(%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)
```

variáveis de modelo são tratadas como variáveis locais em regras de simplificação. (Compare a `defmatch`, que trata variáveis de modelo como variáveis globais.)

```
(%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
```

Como com todos os átomos, propriedades de variáveis de modelo são globais embora valores sejam locais. Nesse exemplo, uma propriedade de atribuição é declarada via `define_variable`. Essa é a propriedade do átomo `bb` através de todo o 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);
```

Regras são nomeadas após operadores principais. Nomes de regras para operadores internos e operadores definidos pelo utilizador são sequências de caracteres, enquanto nomes para outras funções são identificadores comuns.

```
(%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) [trigrule0, trigrule1, trigrule2, trigrule3, trigrule4,
htrigrule1, htrigrule2, htrigrule3, htrigrule4, foorule1,
foorule2, foorule3, +rule1, *rule1, ^rule1]
(%i8) foorule_name: first (%o1);
(%o8)                        foorule1
(%i9) plusrule_name: first (%o4);
(%o9)                         +rule1
(%i10) [?mstringp (foorule_name), symbolp (foorule_name)];
(%o10)                     [false, true]
(%i11) [?mstringp (plusrule_name), symbolp (plusrule_name)];
(%o11)                     [true, true]
(%i12) remrule (foo, foorule1);
(%o12)                           foo
(%i13) remrule ("^", "^rule1");
(%o13)                           ^
```

Um exemplo trabalhado: multiplicação anticomutativa.

```
(%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` () [Função]

Executa `kill (rules)` e então re-escolhe o próximo número de regra para 1 para adição `+`, multiplicação `*`, e exponenciação `^`.

37 Listas

37.1 Introdução a Listas

Listas são o bloco básico de construção para Maxima e Lisp.**Todos os outros tipos de dado como arrays, tabelas desordenadas, números são representados como listas Lisp. Essas listas Lisp possuem a forma

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

para indicar a expressão `a+2`.**No nível um do Maxima poderemos ver a notação infixa `a+2`.**Maxima também tem listas que foram impressas como

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

para uma lista com 4 elementos.**Internamente isso corresponde a uma lista Lisp da forma

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

O sinalizador que denota o tipo campo de uma expressão Maxima é uma lista em si mesmo, após ter sido adicionado o simplificador a lista poderá transforma-se

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

37.2 Definições para Listas

`append` (*list_1*, ..., *list_n*) [Função]

Retorna uma lista simples dos elementos de *list_1* seguidos pelos elementos de *list_2*, ... `append` também trabalha sobre expressões gerais, e.g. `append (f(a,b), f(c,d,e));` retorna `f(a,b,c,d,e)`.

Faça `example(append);` para um exemplo.

`assoc` (*key*, *list*, *default*) [Função]

`assoc` (*key*, *list*) [Função]

Essa função procura pela chave *key* do lado esquerdo da entrada *list* que é da forma `[x,y,z,...]` onde cada elemento de *list* é uma expressão de um operando binário e 2 elementos. Por exemplo `x=1`, `2^3`, `[a,b]` etc. A chave *key* é verificada contra o primeiro operando. `assoc` retorna o segundo operando se `key` for achada. Se a chave `key` não for achada isso retorna o valor padrão *default*. *default* é opcional e o padrão é `false`.

`atom` (*expr*) [Função]

Retorna `true` se *expr* for atomica (i.e. um número, nome ou sequência de caracteres) de outra forma retorna `false`. Desse modo `atom(5)` é `true` enquanto `atom(a[1])` e `atom(sin(x))` São `false` (assumindo `a[1]` e `x` não estão associados).

`cons` (*expr*, *list*) [Função]

Retorna uma nova lista construída do elemento *expr* como seu primeiro elemento, seguido por elementos de *list*. `cons` também trabalha sobre outras expressões, e.g. `cons(x, f(a,b,c));` -> `f(x,a,b,c)`.

`copylist` (*list*) [Função]

Retorna uma cópia da lista *list*.

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

Cria uma lista por avaliação de *form* com *x_1* associando a cada elemento *list_1*, e para cada tal associação anexa *x_2* para cada elemento de *list_2*, ... O número de elementos no resultado será o produto do número de elementos de cada lista. Cada variável *x_i* pode actualmente ser um síbolo –o qual não pode ser avaliado. A lista de argumentos será avaliada uma única vez no início do bloco de repetição.

```
(%i82) create_list1(x^i,i,[1,3,7]);
(%o82) [x,x^3,x^7]
```

Com um bloco de repetição duplo:

```
(%i79) create_list([i,j],i,[a,b],j,[e,f,h]);
(%o79) [[a,e],[a,f],[a,h],[b,e],[b,f],[b,h]]
```

Em lugar de *list_i* dois argumentos podem ser fornecidos cada um dos quais será avaliado como um número. Esses podem vir a ser inclusive o limite inferior e superior do bloco de repetição.

```
(%i81) create_list([i,j],i,[1,2,3],j,1,i);
(%o81) [[1,1],[2,1],[2,2],[3,1],[3,2],[3,3]]
```

Note que os limites ou lista para a variável *j* podem depender do valor corrente de *i*.

`delete` (*expr_1*, *expr_2*) [Função]

`delete` (*expr_1*, *expr_2*, *n*) [Função]

Remove todas as ocorrências de *expr_1* em *expr_2*. *expr_1* pode ser uma parcela de *expr_2* (se isso for uma adição) ou um factor de *expr_2* (se isso for um produto).

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

`delete`(*expr_1*, *expr_2*, *n*) remove as primeiras *n* ocorrências de *expr_1* em *expr_2*. Se houver menos que *n* ocorrências de *expr_1* em *expr_2* então todas as corrências seram excluídas.

```
(%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*) [Função]

Retorna o oitavo item de uma expressão ou lista *expr*. Veja `first` para maiores detalhes.

`endcons` (*expr*, *list*) [Função]

Retorna uma nova lista consistindo de elementos de `list` seguidos por *expr*. `endcons` também trabalha sobre expressões gerais, e.g. `endcons(x, f(a,b,c));` -> `f(a,b,c,x)`.

`fifth` (*expr*) [Função]

Retorna o quinto item da expressão ou lista *expr*. Veja `first` para maiores detalhes.

`first` (*expr*) [Função]

Retorna a primeira parte de *expr* que pode resultar no primeiro elemento de uma lista, a primeira linha de uma matriz, a primeira parcela de uma adição, etc. Note que `first` e suas funções relacionadas, `rest` e `last`, trabalham sobre a forma de *expr* que é mostrada não da forma que é digitada na entrada. Se a variável `inflag` é escolhida para `true` todavia, essa funções olharão na forma interna de *expr*. Note que o simplificador re-ordena expressões. Desse modo `first(x+y)` será `x` se `inflag` for `true` e `y` se `inflag` for `false` (`first(y+x)` fornece os mesmos resultados). As funções `second` .. `tenth` retornam da segunda até a décima parte do seu argumento.

`fourth` (*expr*) [Função]

Retorna o quarto item da expressão ou lista *expr*. Veja `first` para maiores detalhes.

`get` (*a*, *i*) [Função]

Recupera a propriedade de utilizador indicada por *i* associada com o átomo *a* ou retorna `false` se "a" não tem a propriedade *i*.

`get` avalia seus argumentos.

```
(%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*) [Função]

Cria uma nova lista contendo os elementos das lista *l* e *m*, intercaladas. O resultado tem os elementos `[`*l*`[1], `*m*`[1], `*l*`[2], `*m*`[2], ...]`. As listas *l* e *m* podem conter qualquer tipo de elementos.

Se as listas forem de diferentes comprimentos, `join` ignora elementos da lista mais longa.

Maxima reclama se *L_1* ou *L_2* não for uma lista.

Exemplos:

```
(%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*) [Função]
Retorna a última parte (parcela, linha, elemento, etc.) de *expr*.

`length` (*expr*) [Função]
Retorna (por padrão) o número de partes na forma externa (mostrada) de *expr*. Para listas isso é o número de elementos, para matrizes isso é o número de linhas, e para adições isso é o número de parcelas (veja `dispform`).

O comando `length` é afectado pelo comutador `inflag`. Então, e.g. `length(a/(b*c));` retorna 2 se `inflag` for `false` (Assumindo `exptdispflag` sendo `true`), mas 3 se `inflag` for `true` (A representação interna é essencialmente `a*b^-1*c^-1`).

`listarith` [Variável de opção]
Valor por omissão: `true` - se `false` faz com que quaisquer operações aritméticas com listas sejam suprimidas; quando `true`, operações lista-matriz são contagiosas fazendo com que listas sejam convertidas para matrizes retornando um resultado que é sempre uma matriz. Todavia, operações lista-lista podem retornar listas.

`listp` (*expr*) [Função]
Retorna `true` se *expr* for uma lista de outra forma retorna `false`.

`makelist` (*expr*, *i*, *i_0*, *i_1*) [Função]
`makelist` (*expr*, *x*, *list*) [Função]
Constrói e retorna uma lista, cada elemento dessa lista é gerado usando *expr*.

`makelist (`*expr*`, `*i*`, `*i_0*`, `*i_1*`)` retorna uma lista, o `j`'ésimo elemento dessa lista é igual a `ev (`*expr*`, `*i*`=j)` para `j` variando de *i_0* até *i_1*.

`makelist (`*expr*`, `*x*`, `*list*`)` retorna uma lista, o `j`'ésimo elemento é igual a `ev (`*expr*`, `*x*`=`*list*`[j])` para `j` variando de 1 até `length (`*list*`)`.

Exemplos:

```
(%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*) [Função]
Retorna `true` se `is(`*expr_1* `= a)` para algum elemento *a* em `args(`*expr_2*`)`, de outra forma retorna `false`.

`expr_2` é tipicamente uma lista, nesse caso `args(`*expr_2*`) =` *expr_2* e `is(`*expr_1* `= a)` para algum elemento *a* em `expr_2` é o teste.

`member` não inspeciona partes dos argumentos de `expr_2`, então `member` pode retornar `false` mesmo se `expr_1` for uma parte de algum argumento de `expr_2`.

Veja também `elementp`.

Exemplos:

```
(%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*) [Função]

Retorna o nono item da expressão ou lista *expr*. Veja `first` para maiores detalhes.

`rest` (*expr*, *n*) [Função]

`rest` (*expr*) [Função]

Retorna *expr* com seus primeiros *n* elementos removidos se *n* for positivo e seus últimos `-` *n* elementos removidos se *n* for negativo. Se *n* for 1 isso pode ser omitido. *expr* pode ser uma lista, matriz, ou outra expressão.

`reverse` (*list*) [Função]

Ordem reversa para os membros de *list* (não os membros em si mesmos). `reverse` também trabalha sobre expressões gerais, e.g. `reverse(a=b);` fornece `b=a`.

`second` (*expr*) [Função]

Retorna o segundo item da expressão ou lista *expr*. Veja `first` para maiores detalhes.

`seventh` (*expr*) [Função]

Retorna o sétimo item da expressão ou lista *expr*. Veja `first` para maiores detalhes.

`sixth` (*expr*) [Função]

Retorna o sexto item da expressão ou lista *expr*. Veja `first` para maiores detalhes.

`sublist_indices` (*L*, *P*) [Função]

Retorna os índices dos elementos `x` da lista *L* para os quais o predicado `maybe(`*P*`(x))` retornar `true`; isso inclui `unknown` bem como `false`. *P* pode ser um nome de função ou uma expressão lambda. *L* deve ser uma lista literal.

Exemplos:

```
(%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*) [Função]

Retorna o décimo item da expressão ou lista *expr*. Veja `first` para maiores detalhes.

`third` (*expr*) [Função]

Retorna o terceiro item da expressão ou lista *expr*. Veja `first` para maiores detalhes.

38 Conjuntos

38.1 Introdução a Conjuntos

Maxima fornece funções de conjunto, tais como intersecção e união, para conjuntos finitos que são definidos por enumeração explícita. Maxima trata listas e conjuntos como objectos distintos. Este recurso torna possível trabalhar com conjuntos que possuem elementos que são ou listas ou conjuntos.

Adicionalmente, para funções de conjuntos finitos, Maxima fornece algumas funções relacionadas com análise combinatória: números de Stirling de primeiro e de segundo tipo, números de Bell, coeficientes multinomiais e partições de inteiros não negativos, entre outras. Maxima também define a função delta de Kronecker.

38.1.1 Utilização

Para construir um conjunto com elementos `a_1, ..., a_n`, escreva `set(a_1, ..., a_n)` ou `{a_1, ..., a_n}`; para construir o conjunto vazio, escreva `set()` ou `{}`. Para inserção de dados, `set(...)` e `{ ... }` são equivalentes. Os conjuntos são sempre mostrados entre chaves (`{ ... }`).

Se um elemento é listado mais de uma vez, o simplificador do Maxima elimina o elemento redundante.

```
(%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]}
```

Dois elementos x e y são redundantes (nomeadamente, considerados o mesmo para propósito de construção de conjuntos) se e somente se `is(`x `=` y`)` retornar `true`. Note que `is(equal(`x`,` y`))` pode retornar `true` enquanto `is(`x `=` y`)` retorna `false`; nesse caso os elementos x e y são considerados distintos.

```
(%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
```

Para construir um conjunto dos elementos de uma lista, use `setify`.

```
(%i1) setify ([b, a]);
(%o1)                           {a, b}
```

Os elementos de conjunto `x` e `y` serão considerados iguais se `is(x = y)` for avaliando para `true`. Dessa forma, `rat(x)` e `x` são iguais como elementos de conjunto; consequentemente,

```
(%i1) {x, rat(x)};
(%o1)                             {x}
```

Adicionalmente, uma vez que `is((x - 1)*(x + 1) = x^2 - 1)` avalia para `false`, `(x - 1)*(x + 1)` e `x^2 - 1` são considerados elementos de conjunto diferentes; dessa forma

```
(%i1) {(x - 1)*(x + 1), x^2 - 1};
                                          2
(%o1)                {(x - 1) (x + 1), x  - 1}
```

Para reduzir esse conjunto a um conjunto simples, apliquemos `rat` a cada elemento do conjunto

```
(%i1) {(x - 1)*(x + 1), x^2 - 1};
                                          2
(%o1)                {(x - 1) (x + 1), x  - 1}
(%i2) map (rat, %);
                                   2
(%o2)/R/                          {x  - 1}
```

Para remover redundâncias em outros conjuntos, poderá ter que usar outras funções de simplificação. Aqui está um exemplo que usa `trigsimp`:

```
(%i1) {1, cos(x)^2 + sin(x)^2};
                              2         2
(%o1)                 {1, sin (x) + cos (x)}
(%i2) map (trigsimp, %);
(%o2)                            {1}
```

Um conjunto está simplificado quando os seus elementos não são redundantes e o conjunto está ordenado. A versão actual das funções de conjunto usam a função `orderlessp` do Maxima para ordenar conjuntos; contudo, *versões futuras das funções de conjunto poderão vir a usar uma função de ordenação diferente.*

Algumas operações sobre conjuntos, tais como substituições, forçam automaticamente a uma re-simplificação; por exemplo,

```
(%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 trata listas e conjuntos como objectos distintos; funções tais como `union` e `intersection` produzem um erro se qualquer argumento não for um conjunto. se precisar aplicar uma função de conjunto a uma lista, use a função `setify` para converter essa lista num conjunto. Dessa forma

```
(%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}
```

Para extrair todos os elementos de um conjunto `s` que satisfazem um predicado `f`, use `subset(s, f)`. (Um *predicado* é um uma função que avalia para os valores booleanos `true`/`false`.) Por exemplo, para encontrar as equações num dado conjunto que não depende de uma variável `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}
```

A secção *Definições para Conjuntos* possui uma lista completa das funções de conjunto no Maxima.

38.1.2 Iterações entre Elementos de Conjuntos

Existem duas formas de fazer iterações sobre elementos de conjuntos. Uma forma é usar `map`; por exemplo:

```
(%i1) map (f, {a, b, c});
(%o1)                    {f(a), f(b), f(c)}
```

A outra forma consiste em usar `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
```

As funções `first` e `rest` do Maxima trabalham actualmente sobre conjuntos. Aplicada a um conjunto, `first` retorna o primeiro elemento mostrado de um conjunto; qual o élemento que será mostrado dependerá da implementação. Se `s` for um conjunto, então `rest(s)` é equivalente a `disjoin(first(s), s)`. Actualmente, existem outras funções do Maxima que trabalham correctamente sobre conjuntos. Em versões futuras das funções de conjunto, `first` e `rest` podem vir a funcionar diferentemente ou deixar de funcionar.

38.1.3 Erros

As funções de conjunto usam a função `orderlessp` do Maxima para organizar os elementos dum conjunto e a função (a nível do Lisp) `like` para testar a igualdade entre elementos de conjuntos. Ambas essas funções possuem falhas conhecidas que podem se manifestar quando tentar usar conjuntos com elementos que são listas ou matrizes que contenham expressões na forma racional canónica (CRE). Um exemplo é

```
(%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.
```

Essa expressão faz com que o Maxima produza um erro (a mensagem de erro dependerá da versão do Lisp que o Maxima estiver a utilizar). Outro exemplo é

```
(%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.
```

Essas falhas são causadas por falhas em `orderlessp` e `like`, e não por falhas nas funções de conjunto. Para ilustrar, experimente as expressões

```
(%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
```

Até que essas falhas forem corrigidas, não construa conjuntos com elementos que sejam listas ou matrizes contendo expressões na forma racional canónica (CRE); um conjunto com um elemento na forma CRE, contudo, pode não ser um problema:

```
(%i1) {x, rat (x)};
(%o1)                          {x}
```

A `orderlessp` do Maxima possui outra falha que pode causar problemas com funções de conjunto; nomeadamente, o predicado de ordenação `orderlessp` não é transitivo. O mais simples exemplo conhecido que mostra isso é

```
(%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
```

Essa falha pode causar problemas com todas as funções de conjunto bem como com funções do Maxima em geral. É provável, mas não certo, que essa falha possa ser evitada se todos os elementos do conjunto estiverem ou na forma CRE ou tiverem sido simplificados usando `ratsimp`.

Os mecanismos `orderless` e `ordergreat` do Maxima são incompatíveis com as funções de conjunto. Se precisar usar `orderless` ou `ordergrreat`, chame todas essas funções antes de construir quaisquer conjuntos, e não use `unorder`.

Se encontrar alguma coisa que pense ser uma falha em alguma função de conjunto, por favor relate isso para a base de dados de falhas do Maxima. Veja `bug_report`.

38.1.4 Autores

Stavros Macrakis de Cambridge, Massachusetts e Barton Willis da Universidade de Nebraska e Kearney (UNK) escreveram as fnções de conjunto do Maxima e sua documentação.

38.2 Definições para Conjuntos

`adjoin` (*x*, *a*) [Função]

Calcula a união do conjunto *a* com `{x}`.

`adjoin` falha se *a* não for um conjunto literal.

`adjoin(x, a)` e `union(set(x), a)` são equivalentes; contudo, `adjoin` pode ser um pouco mais rápida que `union`.

Veja também `disjoin`.

Exemplos:

```
(%i1) adjoin (c, {a, b});
(%o1)                       {a, b, c}
(%i2) adjoin (a, {a, b});
(%o2)                        {a, b}
```

`belln (n)` [Função]

Representa o n-ésimo número de Bell. `belln(n)` é o número de partições de um conjunto de n elementos.

Para inteiros não negativos n, `belln(n)` simplifica para o n-ésimo número de Bell. `belln` não simplifica para qualquer outro tipo de argumento.

`belln`, aplicada a equações, listas, matrizes e conjuntos, é calculada em forma distributiva.

Exemplos:

`belln` aplicado a inteiros não negativos.

```
(%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` aplicado a argumentos que não são inteiros não negativos.

```
(%i1) [belln (x), belln (sqrt(3)), belln (-9)];
(%o1)        [belln(x), belln(sqrt(3)), belln(- 9)]
```

`cardinality (a)` [Função]

Calcula o número de elementos distintos do conjunto a.

`cardinality` ignora elementos redundantes mesmo quando a simplificação não estiver habilitada.

Exemplos:

```
(%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)` [Função]

Retorna um conjunto de listas da forma `[x_1, ..., x_n]`, onde x_1, ..., x_n são elementos dos conjuntos b_1, ... , b_n, respectivamente.

`cartesian_product` falha se qualquer argumento não for um conjunto literal.

Exemplos:

```
(%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*) [Função]

Retorna o conjunto *a* sem o elemento *x*. Se *x* não for um elemento de *a*, retorna *a* sem modificações.

`disjoin` reclama se *a* não for um conjunto literal.

`disjoin(x, a)`, `delete(x, a)`, e `setdifference(a, set(x))` são todos equivalentes. Desses, `disjoin` é geralmente mais rápido que os outros.

Exemplos:

```
(%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*) [Função]

Retorna `true` se e somente se os conjuntos *a* e *b* forem disjuntos.

`disjointp` falha se ou *a* ou *b* não forem conjuntos literais.

Exemplos:

```
(%i1) disjointp ({a, b, c}, {1, 2, 3});
(%o1)                           true
(%i2) disjointp ({a, b, 3}, {1, 2, 3});
(%o2)                           false
```

`divisors` (*n*) [Função]

Representa o conjunto dos divisores de *n*.

`divisors(n)` produz um conjunto de divisores inteiros quando *n* for um inteiro não nulo. O conjunto dos divisores inclui os elementos 1 e *n*. Os divisores de um inteiro negativo são os divisores do seu valor absoluto.

`divisors`, aplicada a equações, listas, matrizes e conjuntos, é calculada em forma distributiva.

Exemplos:

Podemos verificar que 28 é um número perfeito: a adição dos seus divisores (excepto o próprio 28) é 28.

```
(%i1) s: divisors(28);
(%o1)                 {1, 2, 4, 7, 14, 28}
(%i2) lreduce ("+", args(s)) - 28;
(%o2)                           28
```

`divisors` é uma função de simplificação. Substituindo 8 por `a` em `divisors(a)` calcula os divisores sem ser preciso pedir que `divisors(8)` seja reavaliada.

```
(%i1) divisors (a);
(%o1)                       divisors(a)
(%i2) subst (8, a, %);
(%o2)                      {1, 2, 4, 8}
```

`divisors`, aplicada a equações, listas, matrizes e conjuntos, é calculada em forma distributiva.

```
(%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`) [Função]

Retorna `true` se e somente se *x* for um elemento do conjunto *a*.

`elementp` falha se *a* não for um conjunto literal.

Exemplos:

```
(%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`) [Função]

Retorna `true` se e somente se *a* for o conjunto vazio ou uma lista vazia.

Exemplos:

```
(%i1) map (emptyp, [{}, []]);
(%o1)                      [true, true]
(%i2) map (emptyp, [a + b, {{}}, %pi]);
(%o2)                 [false, false, false]
```

`equiv_classes` (*s*, *F*) [Função]

Retorna um conjunto das classes de equivalências do conjunto *s* com relação à relação de equivalência *F*.

F é uma função de duas variáveis definida sobre o produto cartesiano *s* por *s*. O valor de retorno de *F* é ou `true` ou `false`, ou uma expressão *expr* tal que `is(`*expr*`)` é ou `true` ou `false`.

Quando *F* não for uma relação de equivalência, `equiv_classes` aceita-a sem reclamação, mas o resultado é geralmente incorrecto nesse caso.

Exemplos:

A relação de equivalência é uma expressão lambda a qual retorna `true` ou `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}}
```

A relação de equivalência é o nome de uma função relacional que avalia para `true` ou `false`.

```
(%i1) equiv_classes ({1, 1.0, 2, 2.0, 3, 3.0}, equal);
```

```
(%o1)            {{1, 1.0}, {2, 2.0}, {3, 3.0}}
```

As classes de equivalência são números que diferem por um múltiplo de 3.

```
(%i1) equiv_classes ({1, 2, 3, 4, 5, 6, 7}, lambda ([x, y], remainder (x - y, 3)
(%o1)              {{1, 4, 7}, {2, 5}, {3, 6}}
```

`every` (*f*, *s*) [Função]
`every` (*f*, *L_1*, ..., *L_n*) [Função]

Retorna `true` se o predicado *f* for `true` para todos os argumentos fornecidos.

Dado um conjunto como segundo argumento, `every(f, s)` retorna `true` se `is(f(a_i))` retornar `true` para todos os *a_i* em *s*. `every` pode ou não avaliar *f* para todos os *a_i* em *s*. Uma vez que os conjuntos são desordenados, `every` pode avaliar `f(a_i)` em qualquer ordem.

Dada uma ou mais listas como argumentos, `every(f, L_1, ..., L_n)` retorna `true` se `is(f(x_1, ..., x_n))` retornar `true` para todos os *x_1*, ..., *x_n* em *L_1*, ..., *L_n*, respectivamente. `every` pode ou não avaliar *f* para toda combinação *x_1*, ..., *x_n*. `every` avalia listas na ordem de incremento do índice.

Dado um conjunto vazio `{}` ou uma lista vazia `[]` como argumentos, `every` retorna `false`.

Quando o sinalizador global `maperror` for `true`, todas as listas *L_1*, ..., *L_n* deverão ter o mesmo comprimento. Quando `maperror` for falso, as listas dadas como argumentos serão efectivamente truncadas para o comprimento da menor lista.

Os resultados do predicado *f* que avaliarem (via `is`) para algo diferente de `true` ou `false` são governados através da variável global `prederror`. Quando `prederror` for `true`, tais valores são tratados como `false`, e o valor de retorno de `every` é `false`. Quando `prederror` for `false`, tais valores são tratados como `unknown`, e o valor de retorno de `every` é `unknown`.

Exemplos:

`every` aplicada a um conjunto simples. O predicado é uma função de um argumento.

```
(%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` aplicada a duas listas. O predicado é uma função de dois argumentos.

```
(%i1) every ("=", [a, b, c], [a, b, c]);
(%o1)                         true
(%i2) every ("#", [a, b, c], [a, b, c]);
(%o2)                         false
```

Predicado *f* que produz resultados diferentes de `true` ou `false`, governados por meio da variável global `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*) [Função]
`extremal_subset` (*s*, *f*, *min*) [Função]

Encontra o subconjunto de *s* para o qual a função *f* toma valores máximos ou mínimos.

`extremal_subset(s, f, max)` encontra o subconjunto do conjunto ou lista *s* para os quais a função real *f* assume um valor máximo.

`extremal_subset(s, f, min)` encontra o subconjunto do conjunto ou lista *s* para a qual a função real *f* assume um valor mínimo.

Exemplos:

```
(%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*) [Função]

Colecta argumentos de subexpressões que possuem o mesmo operador que *expr* e constrói uma expressão a partir desses argumentos colectados.

Subexpressões nas quais o operador é diferente do operador principal de `expr` são copiadas sem modificação, mesmo se tiverem subexpressões com o mesmo operador que `expr`.

É possível que `flatten` construia expressões nas quais o número de argumentos difira dos argumentos declarados para um operador; isso pode provocar uma mensagem de erro do simplificador ou do avaliador. `flatten` não tenta detectar tais situações.

Expressões com representações especiais, por exemplo, expressões racionais canónicas (CRE), não podem usar a função `flatten`; nesses casos, `flatten` retorna os seus argumentos sem modificação.

Exemplos:

Aplicado a uma lista, `flatten` reúne todos os elementos da lista que sejam listas.

```
(%i1) flatten ([a, b, [c, [d, e], f], [[g, h]], i, j]);
(%o1)            [a, b, c, d, e, f, g, h, i, j]
```

Aplicado a um conjunto, `flatten` reúne todos os elementos do conjunto que sejam conjuntos.

```
(%i1) flatten ({a, {b}, {{c}}});
(%o1)                         {a, b, c}
(%i2) flatten ({a, {[a], {a}}});
(%o2)                         {a, [a]}
```

o efeito de `flatten` é similar a declarar o operador principal para ser enário. No entanto, `flatten` não faz efeito sobre subexpressões que possuem um operador diferente do operador principal, enquanto uma declaração enária faz efeito.

```
(%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` trata funções subscritas da mesma forma que qualquer outro operador.

```
(%i1) flatten (f[5] (f[5] (x, y), z));
(%o1)                       f (x, y, z)
                             5
```

É possível que `flatten` construa expressões nas quais o número de argumentos difira dos argumentos declarados para um operador;

```
(%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)` [Função]

Substitui todo operador de conjunto em *a* por um operador de lista, e retorna o resultado. `fullt_listify` substitui operadores de conjunto em subexpressões aninhadas, mesmo se o operador principal não for (`set`).

`listify` substitui unicamente o operador principal.

Exemplos:

```
(%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)` [Função]

Quando *a* for uma lista, substitui o operador de lista por um operador de conjunto, e aplica `fullsetify` a cada elemento que for um conjunto. Quando *a* não for uma lista, o resultado é *a* na sua forma original e sem modificações.

`setify` substitui unicamente o operador principal.

Exemplos:

Na linha (%o2), o argumento de `f` não é convertido para um conjunto porque o operador principal de `f([b])` não é uma lista.

```
(%i1) fullsetify ([a, [a]]);
(%o1)                         {a, {a}}
(%i2) fullsetify ([a, f([b])]);
(%o2)                        {a, f([b])}
```

`identity (x)` [Função]

Retorna *x* para qualquer argumento *x*.

Exemplos:

`identity` pode ser usado como um predicado quando os argumentos forem valores Booleanos.

```
(%i1) every (identity, [true, true]);
(%o1)                         true
```

`integer_partitions` (*n*) [Função]
`integer_partitions` (*n*, *len*) [Função]

Calcula partições inteiras de *n*, isto é, listas de inteiros cuja soma dos elementos de cada lista é *n*.

`integer_partitions(n)` encontra o conjunto de todas as partições do inteiro *n*. Cada partição é uma lista ordenada do maior para o menor.

`integer_partitions(n, len)` encontra todas as partições com comprimento *len* ou menor; nesse caso, serão adicionados zeros ao final de cada partição de comprimento menor que *len*, para fazer com que todas as partições tenham exactamente *len* termos. Cada partição é uma lista ordenada do maior para o menor.

Uma lista $[a_1, ..., a_m]$ é uma partição de um inteiro não negativo n quando: (1) cada a_i é um inteiro não nulo, e (2) $a_1 + ... + a_m = n$. Dessa forma, 0 não tem partições.

Exemplos:

```
(%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]}
```

Para encontrar todas as partições que satisfazem uma condição, use a função `subset`; aqui está um exemplo que encontra todas as partições de 10 cujos elementos da lista são números primos.

```
(%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*) [Função]

`intersect` é o mesmo que `intersection`, como veremos.

`intersection` (*a_1*, ..., *a_n*) [Função]

Retorna um conjunto contendo os elementos que são comuns aos conjuntos *a_1* até *a_n*.

`intersection` falha se qualquer dos argumentos não for um conjunto literal.

Exemplos:

```
(%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*) [Função]

Representa a função delta de Kronecker.

`kron_delta` simplifica para 1 quando *x* e *y* forem idênticos ou equivalentes, e simplifica para 0 quando *x* e *y* não forem equivalentes. De outra forma, se não for certo que *x* e *y* são equivalentes, `kron_delta` simplificará para uma expressão substantiva. `kron_delta` implementa uma política de segurança para expressões em ponto flutuante: se a diferença `x - y` for um número em ponto flutuante, `kron_delta` simplifica para uma expressão substantiva quando *x* for aparentemente equivalente a *y*.

Especificamente, `kron_delta(x, y)` simplifica para 1 quando `is(x = y)` for `true`. `kron_delta` também simplifica para 1 quando `sign(abs(x - y))` for `zero` e `x - y` não for um número em ponto flutuante (e também não for um número de precisão simples em ponto flutuante nem um número de precisão dupla em poto flutuante, isto é, não for um bigfloat). `kron_delta` simplifica para 0 quando `sign(abs(x - y))` for `pos`.

Caso contrário, `sign(abs(x - y))` é diferente de `pos` ou `zero`, ou é `zero` e `x - y` é um número em ponto flutuante. Nesses casos, `kron_delta` retorna um expressão substantiva.

`kron_delta` é declarada como sendo simétrica. Isto é, `kron_delta(x, y)` é igual a `kron_delta(y, x)`.

Exemplos:

Os argumentos de `kron_delta` são idênticos. `kron_delta` simplifica para 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
```

Os argumentos de `kron_delta` são equivalentes, e a diferença entre eles não é um número em ponto flutuante. `kron_delta` simplifica para 1.

```
(%i1) assume (equal (x, y));
(%o1)                     [equal(x, y)]
(%i2) kron_delta (x, y);
(%o2)                          1
```

Os argumentos de `kron_delta` não são equivalentes. `kron_delta` simplifica para 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
```

Os argumentos de `kron_delta` podem ou não ser equivalentes. `kron_delta` simplifica para uma expressão substantiva.

```
(%i1) kron_delta (a, b);
(%o1)                   kron_delta(a, b)
(%i2) assume(x >= y)$
(%i3) kron_delta (x, y);
(%o3)                   kron_delta(x, y)
```

Os argumentos de `kron_delta` são equivalentes, mas a diferença entre eles é um número em ponto flutuante. `kron_delta` simplifica para uma expressão substantiva.

```
(%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` é simétrica.

```
(%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)` [Função]

Retorna uma lista contendo os elementos de *a* quando *a* for um conjunto. De outra forma, `listify` retorna *a*.

`full_listify` substitui todos os operadores de conjunto em *a* por operadores de lista.

Exemplos:

```
(%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)` [Função]
`lreduce (F, s, s_0)` [Função]

Extende a função de dois argumentos F para uma função de `n` argumentos, usando composição, onde *s* é uma lista.

`lreduce(F, s)` retorna `F(... F(F(s_1, s_2), s_3), ... s_n)`. Quando o argumento opcional *s_0* estiver presente, o resultado é equivalente a `lreduce(F, cons(s_0, s))`.

A função F é aplicada primeiro aos elementos mais à *esquerda* de lista; daí o nome "lreduce".

Veja também `rreduce`, `xreduce`, e `tree_reduce`.

Exemplos:

`lreduce` sem o argumento opcional.

```
(%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` com o argumento opcional.

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

`lreduce` aplicada a operadores binários internos do Maxima `/` é o operador de divisão.

```
(%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*) [Função]

Retorna um conjunto com elementos gerados a partir da expressão *expr*, onde *x* é uma lista de variáveis em *expr*, e *sé* um conjunto ou lista de listas. Para gerar cada elemento do conjunto, *expr* é avaliada com as variáveis *x* substituídas, em paralelo, por elementos de *s*.

Cada elemento de *s* deve ter o mesmo comprimento que *x*. A lista de variáveis *x* deve ser uma lista de símbolos, sem índices. Mesmo se existir somente um símbolo, *x* deve ser uma lista de um elemento, e cada elemento de *s* deve ser uma lista de um elemento.

Veja também `makelist`.

Exemplos:

```
(%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*) [Função]

Representa a função de Moebius.

Quando *n* for o produto de *k* primos distintos, `moebius(n)` simplifica para $(-1)^k$; quando $n = 1$, simplifica para 1; e simplifica para 0 para todos os outros inteiros positivos.

`moebius`, aplicada a equações, listas, matrizes e conjuntos, é calculada em forma distributiva.

Exemplos:

```
(%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*) [Função]
`multinomial_coeff` () [Função]

Calcula o coeficiente multinomial.

Quando cada *a_k* for um inteiro não negativo, o coeficiente multinomial indica o número de formas possíveis de colocar `a_1 + ... + a_n` objectos distintos em n caixas com *a_k* elementos na k'ésima caixa. Em geral, `multinomial_coeff (a_1, ..., a_n)` calcula `(a_1 + ... + a_n)!/(a_1! ... a_n!)`.

`multinomial_coeff()` (sem argumentos) produz 1.

`minfactorial` poderá conseguir simplificar o valor calculado por `multinomial_coeff`.

Exemplos:

```
(%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*) [Função]
`num_distinct_partitions` (*n*, *list*) [Função]

Calcula o n;umero de partições de inteiros distintos de n quando n for um inteiro não negativo. De outra forma, `num_distinct_partitions` retorna uma expressão substantiva.

`num_distinct_partitions(n, list)` retorna uma lista do número de partições distintas de 1, 2, 3, ..., n.

Uma partição distinta de n é uma lista de inteiros positivos distintos k_1, ..., k_m tais que $n = k_1 + ... + k_m$.

Exemplos:

```
(%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*) [Função]
`num_partitions` (*n*, *list*) [Função]

Calcula o número das partições inteiras de n quando n for um inteiro não negativo. De outra forma, `num_partitions` retorna uma expressão substantiva.

`num_partitions(n, list)` retorna uma lista do número de partições inteiras de 1, 2, 3, ..., n.

Para um inteiro não negativo n, `num_partitions(n)` é igual a `cardinality(integer_partitions(n))`; todavia, `num_partitions` não constrói actualmente o conjunto das partições, nesse sentido `num_partitions` é mais rápida.

Exemplos:

```
(%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*) [Função]

Partições do conjunto *a* que satisfazem o predicado *f*.

`partition_set` retorna uma lista de dois conjuntos. O primeiro conjunto compreende os elementos de *a* para os quais *f* avalia para `false`, e o segundo conjunto compreende quaisquer outros elementos de *a*. `partition_set` não aplica `is` ao valor de retorno de *f*.

`partition_set` reclama se *a* não for um conjunto literal.

Veja também `subset`.

Exemplos:

```
(%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*) [Função]

Retorna um conjunto todas as permutações distintas dos elementos da lista ou do conjunto *a*. Cada permutação é uma lista, não um conjunto.

Quando *a* for uma lista, elementos duplicados de *a* são incluídos nas permutações.

`permutations` reclama se *a* não for um conjunto literal ou uma lista literal.

Exemplos:

```
(%i1) permutations ([a, a]);
(%o1)                       {[a, a]}
(%i2) permutations ([a, a, b]);
(%o2)           {[a, a, b], [a, b, a], [b, a, a]}
```

`powerset (a)` [Função]
`powerset (a, n)` [Função]

Retorna o conjunto de todos os dubconjuntos de *a*, ou um subconjunto de *a*.

`powerset(a)` retorna o conjunto de todos os subconjuntos do conjunto *a*. `powerset(a)` tem `2^cardinality(a)` elementos.

`powerset(a, n)` retorna o conjunto de todos os subconjuntos de *a* que possuem cardinalidade *n*.

`powerset` reclama se *a* não for um conjunto literal, ou se *n* não for um inteiro não negativo.

Exemplos:

```
(%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)                         {{}}
```

`rreduce (F, s)` [Função]
`rreduce (F, s, s_{n + 1})` [Função]

Extende a função de dois argumentos *F* para uma função de *n* argumentos usando composição de funções, onde *s* é uma lista.

`rreduce(F, s)` retorna `F(s_1, ... F(s_{n - 2}, F(s_{n - 1}, s_n)))`. Quando o argumento opcional *s_{n + 1}* estiver presente, o resultado é equivalente a `rreduce(F, endcons(s_{n + 1}, s))`.

A função *F* é primeiro aplicada à lista de elementos *mais à direita - rightmost*, daí o nome "rreduce".

Veja também `lreduce`, `tree_reduce`, e `xreduce`.

Exemplos:

`rreduce` sem o argumento opcional.

```
(%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` com o argumento opcional.

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

`rreduce` aplicada a operadores de dois argumentos internos (definidos por padrão) ao Maxima. `/` é o operadro de divisão.

```
(%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*) [Função]

Retorna um conjunto contendo os elementos no conjunto *a* que não estãono conjunto *b*.

`setdifference` reclama se ou *a* ou *b* não for um conjunto literal.

Exemplos:

```
(%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*) [Função]

Retorna `true` se os conjuntos *a* e *b* possuirem o mesmo número de elementos e `is(`*x* `=` *y*`)` for `true` para `x` nos elementos de *a* e `y` nos elementos de *b*, considerados na ordem determinada por `listify`. De outra forma, `setequalp` retorna `false`.

Exemplos:

```
(%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)` [Função]

Constrói um conjunto de elementos a partir da lista *a*. Elementos duplicados da lista *a* são apagados e os elementos são ordenados de acordo com o predicado `orderlessp`.

`setify` reclama se *a* não for uma lista literal.

Exemplos:

```
(%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)` [Função]

Retorna `true` se e somente se *a* for um conjunto na interpretação do Maxima.

`setp` retorna `true` para conjuntos não simplificados (isto é, conjuntos com elementos redundantes) e também para conjuntos simplificados.

`setp` é equivalente à função do Maxima `setp(a) := not atom(a) and op(a) = 'set`.

Exemplos:

```
(%i1) simp : false;
(%o1)                           false
(%i2) {a, a, a};
(%o2)                         {a, a, a}
(%i3) setp (%);
(%o3)                           true
```

`set_partitions (a)` [Função]
`set_partitions (a, n)` [Função]

Retorna o conjunto de todas as partições de *a*, ou um subconjunto daquele conjunto de partições.

`set_partitions(a, n)` retorna um conjunto de todas as decomposições de *a* em *n* subconjutnos disjuntos não vazios.

`set_partitions(a)` retorna o conjunto de todas as partições.

`stirling2` retorna a cardinalidade de um conjuntode partições de um conjunto.

Um conjunto de conjuntos P é uma partição de um conjunto S quando

1. cada elemento de P é um conjunto não vazio,
2. elementos distintos de P são disjuntos,
3. a união dos elementos de P é igual a S.

Exemplos:

O conjunto vazio é uma partição de si mesmo, as ondições 1 e 2 são "vaziamente" verdadeiras.

```
(%i1) set_partitions ({});
```

```
(%o1)                          {{}}
```

A cardinalidade do conjunto de partições de um conjunto pode ser encontrada usando `stirling2`.

```
(%i1) s: {0, 1, 2, 3, 4, 5}$
(%i2) p: set_partitions (s, 3)$
(%i3) cardinality(p) = stirling2 (6, 3);
(%o3)                       90 = 90
```

Cada elemento de `p` pode ter $n = 3$ elementos; vamos verificar.

```
(%i1) s: {0, 1, 2, 3, 4, 5}$
(%i2) p: set_partitions (s, 3)$
(%i3) map (cardinality, p);
(%o3)                          {3}
```

Finalmente, para cada elementos de `p`, a união de seus elementos possivelmente será igua a `s`; novamente vamos comprovar.

```
(%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*) [Função]

`some` (*f*, *L_1*, ..., *L_n*) [Função]

Retorna `true` se o predicado *f* for `true` para um ou mais argumentos dados.

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.

Dadas uma ou mais listas como argumentos, `some(f, L_1, ..., L_n)` retorna `true` se `is(f(x_1, ..., x_n))` retornar `true` para um ou mais *x_1*, ..., *x_n* em *L_1*, ..., *L_n*, respectivamente. `some` pode ou não avaliar *f* para algumas combinações *x_1*, ..., *x_n*. `some` avalia listas na ordem do índice de incremento.

Dado um conjunto vazio `{}` ou uma lista vazia `[]` como argumentos, `some` retorna `false`.

Quando o sinalizador global `maperror` for `true`, todas as listas *L_1*, ..., *L_n* devem ter obrigatóriamente comprimentos iguais. Quando `maperror` for `false`, argumentos do tipo lista são efectivamente truncados para o comprimento da menor lista.

Retorna o valor de um predicado *f* o qual avalia (por meio de `is`) para alguma coisa outra que não `true` ou `false` e são governados pelo sinalizador global `prederror`. Quando `prederror` for `true`, tais valores são tratados como `false`. Quando `prederror` for `false`, tais valores são tratados como `unknown` (desconhecidos).

Exemplos:

`some` aplicado a um conjunto simples. O predicado é uma função de um argumento.

```
(%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` aplicada a duas listas. O predicado é uma função de dois argumentos.

```
(%i1) some ("=", [a, b, c], [a, b, c]);
(%o1)                          true
(%i2) some ("#", [a, b, c], [a, b, c]);
(%o2)                          false
```

Retorna o valor do predicado f o qual avalia para alguma coisa que não `true` ou `false` e são governados através do sinalizador global `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*`)` [Função]

Representa o número de Stirling de primeiro tipo.

Quando n e m forem não negativos inteiros, a magnitude de `stirling1 (`*n*`, `*m*`)` é o número de permutações de um conjunto com n elementos que possui m ciclos. Para detalhes, veja Graham, Knuth e Patashnik *Concrete Mathematics*. Maxima utiliza uma relação recursiva para definir `stirling1 (`*n*`, `*m*`)` para m menor que 0; `stirling1` não é definida para n menor que 0 e para argumentos não inteiros.

`stirling1` é uma função de simplificação. Maxima conhece as seguintes identidades:

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])

Essas identidades são aplicadas quando os argumentos forem inteiros literais ou símbolos declarados como inteiros, e o primeiro argumento for não negativo. `stirling1` não simplififca para argumentos não inteiros.

Referências:

[1] Donald Knuth, *The Art of Computer Programming,* terceira edição, Volume 1, Seção 1.2.6, Equações 48, 49, e 50.

Exemplos:

```
(%i1) declare (n, integer)$
```

```
(%i2) assume (n >= 0)$
(%i3) stirling1 (n, n);
(%o3)                            1
```

`stirling1` não simplifica para argumentos não inteiros.

```
(%i1) stirling1 (sqrt(2), sqrt(2));
(%o1)              stirling1(sqrt(2), sqrt(2))
```

Maxima aplica identidades a `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*`)` [Função]

Representa o número de Stirling de segundo tipo.

Quando n e m forem inteiros não negativos, `stirling2 (`*n*`, `*m*`)` é o número de maneiras através dos quais um conjunto com cardinalidade n pode ser particionado em m subconjuntos disjuntos. Maxima utiliza uma relação recursiva para definir `stirling2 (`*n*`, `*m*`)` para m menor que 0; `stirling2` é indefinida para n menor que 0 e para argumentos não inteiros.

`stirling2` é uma função de simplificação. Maxima conhece as seguintes identidades.

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. $stirling2(n, m) = sum((-1)^(m-k) binomial(mk) k^n, i, 1, m)/m!$ onde m e n são inteiros, e n é não negativo. (Ref. [3])

Essas identidades são aplicadas quando os argumentos forem inteiros literais ou símbolos declarados como inteiros, e o primeiro argumento for não negativo. `stirling2` não simplifica para argumentos não inteiros.

Referências:

[1] Donald Knuth. *The Art of Computer Programming*, terceira edição, Volume 1, Seção 1.2.6, Equações 48, 49, e 50.

[2] Graham, Knuth, e Patashnik. *Concrete Mathematics*, Tabela 264.

[3] Abramowitz e Stegun. *Handbook of Mathematical Functions*, Seção 24.1.4.

Exemplos:

```
(%i1) declare (n, integer)$
```

```
(%i2) assume (n >= 0)$
(%i3) stirling2 (n, n);
(%o3)                            1
```

`stirling2` não simplifica para argumentos não inteiros.

```
(%i1) stirling2 (%pi, %pi);
(%o1)                  stirling2(%pi, %pi)
```

Maxima aplica identidades a `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*) [Função]

Retorna o subconjuntode um conjunto *a* que satisfaz o predicado *f*.

`subset` returns um conjunto 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` reclama se *a* não for um conjunto literal.

See also `partition_set`.

Exemplos:

```
(%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*) [Função]

Retorna `true` se e somente se o conjunto *a* for um subconjunto de *b*.

`subsetp` reclama se ou *a* ou *b* não forem um conjunto literal.

Exemplos:

```
(%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*) [Função]

Retorna a diferença simétrica, isto é, o conjunto dos elemetnos que ocorrem em exactamente um conjunto *a_k*.

Given two arguments, `symmdifference(a, b)` is the same as `union(setdifference(a, b), setdifference(b, a))`.

`symmdifference` reclama se any argument não for um conjunto literal.

Exemplos:

```
(%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*) [Função]
`tree_reduce` (*F*, *s*, *s_0*) [Função]

Extende a função binária *F* a uma função enária através de composição, onde *s* é um conjunto ou uma lista.

`tree_reduce` é equivalente ao seguinte: Aplicar *F* a sucessivos pares de elementos para formar uma nova lista `[F(s_1, s_2), F(s_3, s_4), ...]`, mantendo o elemento final inalterado caso haja um número ímpar de elementos. Repetindo então o processo até que a lista esteja reduzida a um elemento simples, o qual é o valor de retorno da função.

Quando o argumento opcional *s_0* estiver presente, o resultado é equivalente a `tree_reduce(F, cons(s_0, s)`.

Para adições em ponto flutuante, `tree_reduce` pode retornar uma soma que possui um menor ero de arredondamento que `rreduce` ou `lreduce`.

Os elementos da lista *s* e os resultados parciais podem ser arranjados em uma árvore binária de profundidade mínima, daí o nome "tree_reduce".

Exemplos:

`tree_reduce` aplicada a uma lista com um número par de elementos.

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

`tree_reduce` aplicada a uma lista com um número ímpar de elementos.

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

`union` (*a_1*, ..., *a_n*) [Função]

Retorna a união dos conjuntos de *a_1* a *a_n*.

`union()` (sem argumentos) retorna o conjunto vazio.

`union` reclama se qualquer argumento não for um conjunto literal.

Exemplos:

```
(%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*) [Função]
`xreduce` (*F*, *s*, *s_0*) [Função]

Extendendo a função *F* para uma função enária por composição, ou, se *F* já for enária, aplica-se *F* a *s*. Quando *F* não for enária, `xreduce` funciona da mesma forma que `lreduce`. O argumento *s* é uma lista.

Funções sabidamente enárias inclui adição `+`, multiplicação `*`, `and`, `or`, `max`, `min`, e `append`. Funções podem também serem declaradas enárias por meio de `declare(F, nary)`. Para essas funções, é esperado que `xreduce` seja mais rápida que ou `rreduce` ou `lreduce`.

Quando o argumento opcional *s_0* estiver presente, o resultado é equivalente a `xreduce(s, cons(s_0, s))`.

Adições em ponto flutuante não são exactamente associativas; quando a associatividade ocorrer, `xreduce` aplica a adição enária do Maxima quando *s* contiver números em ponto flutuante.

Exemplos:

`xreduce` aplicada a uma função sabidamente enária. `F` é chamada uma vez, com todos os argumentos.

```
(%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` aplicada a uma função não sabidamente enária. `G` é chamada muitas vezes, com dois argumentos de cada vez.

```
(%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 Definição de Função

39.1 Introdução a Definição de Função

39.2 Função

39.2.1 Ordinary functions

Para definir uma função no Maxima usa-se o operador :=. Por exemplo,

```
f(x) := sin(x)
```

define uma função `f`. Funções anônimas podem também serem criadas usando `lambda`. Por exemplo

```
lambda ([i, j], ...)
```

pode ser usada em lugar de `f` onde

```
f(i,j) := block ([], ...);
map (lambda ([i], i+1), l)
```

retornará uma lista com 1 adicionado a cada termo.

Pode também definir uma função com um número variável de argumentos, usando um argumento final que seja uma lista, na qual serão inseridos todos os argumentos adicionais:

```
(%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]]
```

O lado direito na definição de uma função é uma expressão. Assim, quando quiser que a definição seja uma sequência de expressões, poderá usar a forma

```
f(x) := (expr1, expr2, ...., exprn);
```

e o valor de *exprn* é que é retornado pela função.

Se quiser introduzir um ponto de `retorno` em alguma expressão dentro da função, deverá usar `block` e `return`.

```
block ([], expr1, ..., if (a > 10) then return(a), ..., exprn)
```

é em si mesma uma expressão, e então poderá ocupar o lugar do lado direito de uma definição de função. Aqui pode acontecer que o retorno aconteça mais facilmente que no exemplo anterior a essa última expressão.

O primeiro `[]` no bloco, pode conter uma lista de variáveis e atribuições de variáveis, tais como `[a: 3, b, c: []]`, que farão com que as três variáveis `a`,`b`,e `c` não se refiram a seus valores globais, mas ao contrário tenham esses valores especiais enquanto o código estiver executando a parte dentro do bloco `block`, ou dentro da funções chamadas de dentro do bloco `block`. Isso é chamado associação *dynamic*, uma vez que as variáveis permanecem do início do bloco pelo tempo que ele existir. Quando regressar do bloco `block`, ou o descartar,

os valores antigos (quaisquer que sejam) das variáveis serão restaurados. É certamente uma boa idéia para proteger as suas variáveis nesse caminho. Note que as atribuições em variáveis do bloco, são realizadas em paralelo. Isso significa, que se tivesse usado `c: a` acima, o valor de `c` seria o valor que `a` tinha antes do bloco, antes de ter obtido o seu novo valor atribuído no bloco. Dessa forma fazendo alguma coisa como

```
block ([a: a], expr1, ...  a: a+3, ..., exprn)
```

protegerá o valor externo de `a` de ser alterado, mas impedirá aceder ao valor antigo. Assim, o lado direito de atribuições, é avaliado no contexto inserido, antes que qualquer avaliação ocorra. Usando apenas `block ([x], ...` faremos com que o `x` tenho como valor a si próprio; esse é o mesmo valor que teria no início de uma sessão do **Maxima**.

Os actuais argumentos para uma função são tratados exactamente da mesma que as variáveis em um bloco. Dessa forma em

```
f(x) := (expr1, ..., exprn);
```

e

```
f(1);
```

teremos um contexto similar para avaliação de expressões como se tivéssemos concluído

```
block ([x: 1], expr1, ..., exprn)
```

Dentro de funções, quando o lado direito de uma definição, pode ser calculado em tempo de execução, isso é úti para usar `define` e possivelmente `buildq`.

39.2.2 Função de Array

Uma função de Array armazena o valor da função na primeira vez que ela for chamada com um argumento dado, e retorna o valor armazenado, sem recalcular esse valor, quando o mesmo argumento for fornecido. De modo que uma função é muitas vezes chamada uma *função de memorização*.

Nomes de funções de Array são anexados ao final da lista global `arrays` (não na lista global `functions`). O comando `arrayinfo` retorna a lista de argumentos para os quais exite valores armazenados, e `listarray` retorna os valores armazenados. Os comandos `dispfun` e `fundef` retornam a definição da função de array.

O comando `arraymake` contrói uma chamada de função de array, análogamente a `funmake` para funções comuns. O comando `arrayapply` aplica uma função de array a seus argmentos, análogamente a `apply` para funções comuns. Não existe nada exactamente análogo a `map` para funções de array, embora `map(lambda([x], a[x]), L)` ou `makelist(a[x], x, L)`, onde L é uma lista, não estejam tão longe disso.

O comando `remarray` remove uma definição de função de array (incluindo qualquer valor armazenado pela função removida), análogo a `remfunction` para funções comuns.

o comando `kill(a[x])` remove o valor da função de array a armazenado para o argumento x; a próxima vez que a foor chamada com o argumento x, o valor da função é recomputado. Todavia, não exite caminho para remover todos os valores armazenados de uma vez, excepto para `kill(a)` ou `remarray(a)`, o qual remove também remove a definição da função de array.

39.3 Macros

`buildq` (*L*, *expr*) [Função]

Substitue variáveis nomeadas pela lista *L* dentro da expressão *expr*, paralelamente, sem avaliar *expr*. A expressão resultante é simplificada, mas não avaliada, após `buildq` realizar a substituição.

Os elementos de *L* são símbolos ou expressões de atribuição *símbolo*`:` *valor*, avaliadas paralelamente. Isto é, a associação de uma variável sobre o lado direito de uma atribuição é a associação daquela variável no contexto do qual `buildq` for chamada, não a associação daquela variável na lista *L* de variáveis. Se alguma variável em *L* não dada como uma atribuição explícita, sua associação em `buildq` é a mesma que no contexto no qual `buildq` for chamada.

Então as variáveis nomeadas em *L* são substituidas em *expr* paralelamente. Isto é, a substituição para cada variável é determinada antes que qualquer substituição seja feita, então a substituição para uma variável não tem efeito sobre qualquer outra.

Se qualquer variável *x* aparecer como `splice (x)` em *expr*, então *x* deve estar associada para uma lista, e a lista recebe uma aplicação da função `splice` (é interpolada) na *expr* em lugar de substituída.

Quaisquer variáveis em *expr* não aparecendo em *L* são levados no resultado tal como foram escritos, mesmo se elas tiverem associações no contexto do qual `buildq` tiver sido chamada.

Exemplos

`a` é explicitamente associada a `x`, enquanto `b` tem a mesma associação (nomeadamente 29) como no contexto chamado, e `c` é levada do começo ao fim da forma como foi escrita. A expressão resultante não é avaliada até a avaliação explícita (com duplo apóstrofo - não com aspas - `''%`.

```
(%i1) (a: 17, b: 29, c: 1729)$
(%i2) buildq ([a: x, b], a + b + c);
(%o2)                       x + c + 29
(%i3) ''%;
(%o3)                        x + 1758
```

`e` está associado a uma lista, a qual aparece também como tal nos argumentos de `foo`, e interpolada nos argumentos de `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)
```

O resultado é simplificado após substituição. Se a simplificação for aplicada antes da substituição, esses dois resultados podem ser iguais.

```
(%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
```

As variáveis em *L* são associadas em paralelo; se associadas sequêncialmente, o primeiro resultado pode ser `foo (b, b)`. Substituições são realizadas em paralelo;

compare o segundo resultado com o resultado de `subst`, que realiza substituições sequêncialmente.

```
(%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)
```

Constrói uma lista de euqções com algumas variáveis ou expressões sobre o lado esquerdo e seus valores sobre o lado direito. `macroexpand` mostra a expressão retornada por `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 = 1729]
```

`macroexpand` (*expr*) [Função]

Retorna a expansão da macro de *expr* sem avaliar a expressão, quando `expr` for uma chamada de função de macro. De outra forma, `macroexpand` retorna *expr*.

Se a expansão de *expr* retorna outra chamada de função de macro, aquela chamada de função de macro é também expandida.

`macroexpand` coloca apóstrofo em seus argumentos, isto é, não os avalia. Todavia, se a expansão de uma chamada de função de macro tiver algum efeito, esse efeito colateral é executado.

Veja também `::=`, `macros`, e `macroexpand1`.

Exemplos

```
(%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*`)` [Função]

Retorna a expansão de macro de *expr* sem avaliar a expressão, quando `expr` for uma chamada de função de macro. De outra forma, `macroexpand1` retorna *expr*.

`macroexpand1` não avalia seus argumentos. Todavia, se a expansão de uma chamada de função de macro tiver algum efeito, esse efeito colateral é executado.

Se a expansão de *expr* retornar outra chamada de função de macro, aquela chamada de função de macro não é expandida.

Veja também `::=`, `macros`, e `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` é a lista de funções de macro definidas pelo utilizador. O operador de definição de função de macro `::=` coloca uma nova função de macro nessa lista, e `kill`, `remove`, e `remfunction` removem funções de macro da lista.

Veja também `infolists`.

`splice (a)` [Função]

Une como se fosse um elo de ligação (interpola) a lista nomeada através do átomo *a* em uma expressão, mas somente se `splice` aparecer dentro de `buildq`; de outra forma, `splice` é tratada como uma função indefinida. Se aparecer dentro de `buildq` com *a* sozinho (sem `splice`), *a* é substituido (não interpolado) como uma lista no resultado. O argumento de `splice` pode somente ser um átomo; não pode ser uma lista lateral ou uma expressão que retorna uma lista.

Tipicamente `splice` fornece os argumentos para uma função ou operador. Para uma função `f`, a expressão `f (splice (a))` dentro de `buildq` expande para `f (a[1], a[2], a[3], ...)`. Para um operador `o`, a expressão `"o" (splice (a)` dentro de `buildq` expande para `"o" (a[1], a[2], a[3], ...)`, onde `o` pode ser qualquer tipo de operador (tipicamente um que toma múltiplos argumentos). Note que o operador deve ser contido dentro de aspas duplas `"`.

Exemplos

```
(%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 Definições para Definição de Função

`apply` (*F*, [*x_1*, ..., *x_n*]) [Função]

Constrói e avalia uma expressãp *F*(*arg_1*, ..., *arg_n*).

`apply` não tenta distinguir funções de array de funções comuns; quando *F* for o nome de uma função de array, `apply` avalia *F*(...) (isto é, uma chamada de função com parêntesis em lugar de colchêtes). `arrayapply` avalia uma chamada de função com colchêtes nesse caso.

Exemplos:

`apply` avalia seus argumentos. Nesse exemplo, `min` é aplicado a `L`.

```
(%i1) L : [1, 5, -10.2, 4, 3];
(%o1)                  [1, 5, - 10.2, 4, 3]
(%i2) apply (min, L);
(%o2)                          - 10.2
```

`apply` avalia argumentos, mesmo se a função *F* disser que os argumentos não devem ser avaliados.

```
(%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` avalia o nome de função *F*. Apóstrofo ’ evita avaliação. `demoivre` é o nome de uma variável global e também de uma função.

```
(%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*) [Função]
`block` (*expr_1*, ..., *expr_n*) [Função]

`block` avalia *expr_1*, ..., *expr_n* em sequência e retorna o valor da última expressão avaliada. A sequência pode ser modificada pelas funções `go`, `throw`, e `return`. A última expressão é *expr_n* a menos que `return` ou uma expressão contendo `throw` seja avaliada. Algumas variáveis *v_1*, ..., *v_m* podem ser declaradas locais para o bloco; essas são distinguidas das variáveis globais dos mesmos nomes. Se variáveis não forem declaradas locais então a lista pode ser omitida. Dentro do bloco, qualquer variável que não *v_1*, ..., *v_m* é uma variável global.

`block` salva os valores correntes das variáveis *v_1*, ..., *v_m* (quaisquer valores) na hora da entrada para o bloco, então libera as variáveis dessa forma eles avaliam para si mesmos. As variáveis locais podem ser associadas a valores arbitrários dentro do bloco mas quando o bloco é encerrado o valores salvos são restaurados, e os valores atribuídos dentro do bloco são perdidos.

`block` pode aparecer dentro de outro `block`. Variáveis locais são estabelecidas cada vez que um novo `block` é avaliado. Variáveis locais parecem ser globais para quaisquer blocos fechados. Se uma variável é não local em um bloco, seu valor é o valor mais recentemente atribuído por um bloco fechado, quaisquer que sejam, de outra forma, seu valor é o valor da variável no ambiente global. Essa política pode coincidir com o entendimento usual de "escopo dinâmico".

Se isso for desejado para salvar e restaurar outras propriedades locais ao lado de `value`, por exemplo `array` (excepto para arrays completos), `function`, `dependencies`, `atvalue`, `matchdeclare`, `atomgrad`, `constant`, e `nonscalar` então a função `local` pode ser usada dentro do bloco com argumentos sendo o nome das variáveis.

O valor do bloco é o valor da última declaração ou o valor do argumento para a função `return` que pode ser usada para sair explicitamente do bloco. A função `go` pode ser usada para transferir o controle para a declaração do bloco que é identificada com o argumento para `go`. Para identificar uma declaração, coloca-se antes dela um

argumento atômico como outra declaração no bloco. Por exemplo: `block ([x], x:1, loop, x: x+1, ..., go(loop), ...)`. O argumento para `go` deve ser o nome de um identificador que aparece dentro do bloco. Não se deve usar `go` para transferir para um identificador em um outro bloco a não ser esse que contém o `go`.

Blocos tipicamente aparecem do lado direito de uma definição de função mas podem ser usados em outros lugares também.

`break` (*expr_1*, ..., *expr_n*) [Função]

Avalia e imprime *expr_1*, ..., *expr_n* e então causa uma parada do Maxima nesse ponto e o utilizador pode examinar e alterar seu ambiente. Nessa situação digite `exit;` para que o cálculo seja retomado.

`catch` (*expr_1*, ..., *expr_n*) [Função]

Avalia *expr_1*, ..., *expr_n* uma por uma; se qualquer avaliação levar a uma avaliação de uma expressão da forma `throw (arg)`, então o valor de `catch` é o valor de `throw (arg)`, e expressões adicionais não são avaliadas. Esse "retorno não local" atravessa assim qualquer profundidade de aninhar para o mais próximo contendo `catch`. Se não existe nenhum `catch` contendo um `throw`, uma mensagem de erro é impressa.

Se a avaliação de argumentos não leva para a avaliação de qualquer `throw` então o valor de `catch` é o valor de *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
```

A função `g` retorna uma lista de `f` de cada elemento de `l` se `l` consiste somente de números não negativos; de outra forma, `g` "captura" o primeiro elemento negativo de `l` e "arremessa-o".

`compfile` (*nomeficheiro*, *f_1*, ..., *f_n*) [Função]
`compfile` (*nomeficheiro*, *funções*) [Função]
`compfile` (*nomeficheiro*, *all*) [Função]

Traduz fuções Maxima para Lisp e escreve o código traduzido no ficheiro *nomeficheiro*.

`compfile(`*nomeficheiro*`, `*f_1*`, ..., `*f_n*`)` traduz as funções especificadas. `compfile(`*nomeficheiro*`, functions)` e `compfile(`*nomeficheiro*`, all)` traduz todas as funções definidas pelo utilizador.

As traduções Lisp não são avaliadas, nem é o ficheiro de saída processado pelo compilador Lisp. `translate` cria e avalia traduções Lisp. `compile_file` traduz Maxima para Lisp, e então executa o compilador Lisp.

Veja também `translate`, `translate_file`, e `compile_file`.

`compile` (*f_1*, ..., *f_n*) [Função]
`compile` (*funções*) [Função]
`compile` (*all*) [Função]

Traduz funções Maxima *f_1*, ..., *f_n* para Lisp, avalia a tradução Lisp, e chama a função Lisp `COMPILE` sobre cada função traduzida. `compile` retorna uma lista de nomes de funções compiladas.

`compile (all)` ou `compile (funções)` compila todas as funções definidas pelo utilizador.

`compile` não avalia seus argumentos; o operador apóstrofo-apóstrofo `''` faz com que ocorra avaliação sobrepondo-se ao apóstrofo.

`define` (*f*(*x_1*, ..., *x_n*), *expr*) [Função]
`define` (*f*[*x_1*, ..., *x_n*], *expr*) [Função]
`define` (*funmake* (*f*, [*x_1*, ..., *x_n*]), *expr*) [Função]
`define` (*arraymake* (*f*, [*x_1*, ..., *x_n*]), *expr*) [Função]
`define` (*ev* (*expr_1*), *expr_2*) [Função]

Define uma função chamada *f* com argumentos *x_1*, ..., *x_n* e corpo da função *expr*. `define` sempre avalia seu segundo argumento (a menos que explícitamente receba um apostrofo de forma a evitar a avaliação). A função então definida pode ser uma função comum do Maxima (com argumentos contidos entre parêtesis) ou uma função de array (com argumentos contidos entre colchêtes).

Quando o último ou único argumento da função *x_n* for uma lista de um elemento, a função definida por `define` aceita um número variável de argumentos. Os argumentos actuais são atribuídos um a um a argumentos formais *x_1*, ..., *x_(n - 1)*, e quaisquer argumentos adicionais actuais, se estiverem presentes, são atribuídos a *x_n* como uma lista.

Quando o primeiro argumento de `define` for uma expressão da forma `f(x_1, ..., x_n)` or `f[x_1, ..., x_n]`, os argumentos são avaliados mas *f* não é avaliada, mesmo se já existe anteriormente uma função ou variável com aquele nome. Quando o primeiro argumento for uma expressão com operador `funmake`, `arraymake`, ou `ev`, o primeiro argumento será avaliado; isso permite para o nome da função seja calculado, também como o corpo.

Todas as definições de função aparecem no mesmo nível de escopo e visibilidade; definindo uma função `f` dentro de outra função `g` não limita o escopo de `f` a `g`.

Se algum argumento formal *x_k* for um símbolo com apóstrofo (após ter sido feita uma avaliação), a função definida por `define` não avalia o correspondente actual argumento. de outra forma todos os argumentos actuais são avaliados.

Veja também `:=` and `::=`.

Exemplos:

`define` sempre avalia seu segundo argumento (a menos que explícitamente receba um apostrofo de forma a evitar a avaliação).

```
(%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)
```

A função definida por `define` pode ser uma função comum do Maxima ou uma função de array.

```
(%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
```

Quando o último ou único argumento da função *x_n* for uma lista de um único elemento, a função definida por `define` aceita um número variável de argumentos.

```
(%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*) [Função]

Introduz uma variável global dentro do ambiente Maxima. `define_variable` é útil em pacotes escritos pelo utilizador, que são muitas vezes traduzidos ou compilados.

`define_variable` realiza os seguintes passos:

1. `mode_declare (`*name*`, `*mode*`)` declara o modo de *name* para o tradutor. Veja `mode_declare` para uma lista dos modos possíveis.
2. Se a variável é não associada, *default_value* é atribuído para *name*.
3. `declare (`*name*`, special)` declara essa variável especial.
4. Associa *name* com uma função de teste para garantir que a *name* seja somente atribuído valores do modo declarado.

A propriedade `value_check` pode ser atribuída a qualquer variável que tenha sido definida via `define_variable` com um outro modo que não `any`. A propriedade `value_check` é uma expressão lambda ou o nome de uma função de uma variável, que é chamada quando uma tentativa é feita para atribuir um valor a uma variável. O argumento da função `value_check` é o valor que será atribuído.

`define_variable` avalia `default_value`, e não avalia `name` e `mode`. `define_variable` retorna o valor corrente de `name`, que é `default_value` se `name` não tiver sido associada antes, e de outra forma isso é o valor prévio de `name`.

Exemplos:

`foo` é uma variável Booleana, com o valor inicial `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` é uma variável inteira, que deve ser um número primo.

```
(%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 é not prime.
#0: prime_test(y=1440)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i6) bar;
(%o6)                          1439
```

`baz_quux` é uma variável que não pode receber a atribuição de um valor. O modo `any_check` é como `any`, mas `any_check` habilita o mecanismo `value_check`, e `any` não habilita.

```
(%i1) define_variable (baz_quux, 'baz_quux, any_check);
(%o1)                        baz_quux
(%i2) F: lambda ([y], if y # 'baz_quux then error ("Cannot assign to ‘baz_quux’."
(%o2) lambda([y], if y # 'baz_quux

                         then error(Cannot assign to ‘baz_quux’.))
(%i3) qput (baz_quux, ''F, value_check);
(%o3) lambda([y], if y # 'baz_quux
```

```
                              then error(Cannot assign to ‘baz_quux’.))
(%i4) baz_quux: ’baz_quux;
(%o4)                          baz_quux
(%i5) baz_quux: sqrt(2);
Cannot assign to ‘baz_quux’.
#0: lambda([y],if y # ’baz_quux then error("Cannot assign to ‘baz_quux’."))(y=sqr
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i6) baz_quux;
(%o6)                          baz_quux
```

`dispfun` (*f_1*, ..., *f_n*) [Função]

`dispfun` (*all*) [Função]

Mostra a definição de funções definidas pelo utilizador *f_1*, ..., *f_n*. Cada argumento pode ser o nome de uma macro (definida com `::=`), uma função comum (definida com `:=` ou `define`), uma função array (definida com `:=` ou com `define`, mas contendo argumentos entre colchêtes `[ ]`), uma função subscrita, (definida com `:=` ou `define`, mas contendo alguns argumentos entre colchêtes e outros entre parêntesis `( )`) uma da família de funções subscritas seleccionadas por um valor subscrito particular, ou uma função subscrita definida com uma constante subscrita.

`dispfun (all)` mostra todas as funções definidas pelo utilizador como dadas pelas `functions`, `arrays`, e listas de `macros`, omitindo funções subscritas definidas com constantes subscritas.

`dispfun` cria um Rótulo de expressão intermédia (`%t1`, `%t2`, etc.) para cada função mostrada, e atribui a definição de função para o rótulo. Em contraste, `fundef` retorna a definição de função.

`dispfun` não avalia seus argumentos; O operador apóstrofo-apóstrofo `’’` faz com que ocorra avaliação.

`dispfun` retorna a lista de rótulos de expressões intermédias correspondendo às funções mostradas.

Exemplos:

```
(%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` [Variável de sistema]

Valor por omissão: `[]`

`functions` é uma lista de todas as funções comuns do Maxima na sessão corrente. Uma função comum é uma função construída através de `define` ou de `:=` e chamada com parêntesis `()`. Uma função pode ser definida pela linha de comando do Maxima

de forma interativa com o utilizador ou em um ficheiro Maxima chamado por `load` ou `batch`.

Funções de array (chamadas com colchêtes, e.g., `F[x]`) e funções com subscritos (chamadas com colchêtes e parêntesis, e.g., `F[x](y)`) são lsitados através da variável global `arrays`, e não por meio de `functions`.

Funções Lisp não são mantidas em nenhuma lista.

Exemplos:

```
(%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)` [Função]

Retorna a definição da função *f*.

O argumento pode ser o nome de uma macro (definida com `::=`), uma função comum (definida com `:=` ou `define`), uma função array (definida com `:=` ou `define`, mas contendo argumentos entre colchêtes `[ ]`), Uma função subscrita, (definida com `:=` ou `define`, mas contendo alguns argumentos entre colchêtes e parêntesis `( )`) uma da família de funções subscritas seleccionada por um valor particular subscrito, ou uma função subscrita definida com uma constante subscrita.

`fundef` não avalia seu argumento; o operador apóstrofo-apóstrofo `''` faz com que ocorra avaliação.

`fundef (f)` retorna a definição de *f*. Em contraste, `dispfun (f)` cria um rótulo de expressão intermédia e atribui a definição para o rótulo.

`funmake` (*F*, [*arg_1*, ..., *arg_n*]) [Função]

Retorna uma expressão *F*(*arg_1*, ..., *arg_n*). O valor de retorno é simplificado, mas não avaliado, então a função F não é chamada, mesmo se essa função F existir.

`funmake` não tenta distinguir funções de array de funções comuns; quando F for o nome de uma função de array, `funmake` retorna *F*(...) (isto é, uma chamada de função com parêntesis em lugar de colchêtes). `arraymake` retorna uma chamada de função com colchêtes nesse caso.

`funmake` avalia seus argumentos.

Exemplos:

`funmake` aplicada a uma função comum do Maxima.

```
(%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` aplicada a uma 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` aplicada a uma função subscrita.

```
(%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` aplicada a um símbolo que não é uma função definida de qualquer tipo.

```
(%i1) funmake (A, [u]);
(%o1)                           A(u)
(%i2) ''%;
(%o2)                           A(u)
```

`funmake` avalia seus argumentos, mas não o valor de retorno.

```
(%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 simplifica o valor de retorno de `funmake`.

```
(%i1) funmake (sin, [%pi / 2]);
(%o1)                             1
```

`lambda` ([*x_1*, ..., *x_m*], *expr_1*, ..., *expr_n*) [Função]
`lambda` ([[*L*]], *expr_1*, ..., *expr_n*) [Função]
`lambda` ([*x_1*, ..., *x_m*, [*L*]], *expr_1*, ..., *expr_n*) [Função]

Define e retorna uma expressão lambda (que é, uma função anônima) A função pode ter argumentos que sejam necessários *x_1*, ..., *x_m* e/ou argumentos opcionais *L*, os quais aparecem dentro do corpo da função como uma lista. O valor de retorno da função é *expr_n*. Uma expressão lambda pode ser atribuída para uma variável e avaliada como uma função comum. Uma expressão lambda pode aparecer em alguns contextos nos quais um nome de função é esperado.

Quando a função é avaliada, variáveis locais não associadas *x_1*, ..., *x_m* são criadas. `lambda` pode aparecer dentro de `block` ou outra função `lambda`; variáveis locais são estabelecidas cada vez que outro `block` ou função `lambda` é avaliada. Variáveis locais parecem ser globais para qualquer coisa contendo `block` ou `lambda`. Se uma variável é não local, seu valor é o valor mais recentemente atribuído em alguma coisa contendo `block` ou `lambda`, qualquer que seja, de outra forma, seu valor é o valor da variável no ambiente global. Essa política pode coincidir com o entendimento usual de "escopo dinâmico".

Após variáveis locais serem estabelecidas, *expr_1* até *expr_n* são avaliadas novamente. a variável especial `%%`, representando o valor da expressão precedente, é reconhecida. `throw` e `catch` pode também aparecer na lista de expressões.

`return` não pode aparecer em uma expressão lambda a menos que contendo `block`, nesse caso `return` define o valor de retorno do bloco e não da expressão lambda, a

menos que o bloco seja *expr_n*. Da mesma forma, `go` não pode aparecer em uma expressão lambda a menos que contendo `block`.

`lambda` não avalia seus argumentos; o operador apóstrofo-apóstrofo `''` faz com que ocorra avaliação.

Exemplos:

- A expressão lambda pode ser atribuída para uma variável e avaliada como uma função comum.

```
(%i1) f: lambda ([x], x^2);
                                        2
(%o1)                     lambda([x], x )
(%i2) f(a);
                                 2
(%o2)                           a
```

- Uma expressão lambda pode aparecer em contextos nos quais uma avaliação de função é esperada como resposta.

```
(%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 ]
```

- Variáveis argumento são variáveis locais. Outras variáveis aparecem para serem variáveis globais. Variáveis globais são avaliadas ao mesmo tempo em que a expressão lambda é avaliada, a menos que alguma avaliação especial seja forçada por alguns meios, tais como `''`.

```
(%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
```

- Expressões lambda podem ser aninhadas. Variáveis locais dentro de outra expressão lambda parece ser global para a expressão interna a menos que mascarada por variáveis locais de mesmos nomes.

```
(%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
```

- Uma vez que `lambda` não avalia seus argumentos, a expressão lambda `i` abaixo não define uma função "multiplicação por `a`". Tanto uma função pode ser definida via `buildq`, como na expressão lambda `i2` abaixo.

```
(%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
```

- Uma expressão lambda pode receber um número variável de argumentos, os quais são indicados por meio de `[L]` como o argumento único ou argumento final. Os argumentos aparecem dentro do corpo da função como uma lista.

```
(%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*) [Função]

Declara as variáveis *v_1*, ..., *v_n* para serem locais com relação a todas as propriedades na declaração na qual essa função é usada.

`local` não avalia seus argumentos. `local` retorna `done`.

`local` pode somente ser usada em `block`, no corpo de definições de função ou expressões `lambda`, ou na função `ev`, e somente uma ocorrêcia é permitida em cada.

`local` é independente de `context`.

`macroexpansion` [Variável de opção]

Valor por omissão: `false`

`macroexpansion` controla recursos avançados que afectam a eficiência de macros. Escolhas possíveis:

- `false` – Macros expandem normalmente cada vez que são chamadas.
- `expand` – A primeira vez de uma chamada particular é avaliada, a expansão é lembrada internamente, dessa forma não tem como ser recalculada em chamadas subsequênte rapidamente. A macro chama ainda chamadas `grind` e `display` normalmente. Todavia, memória extra é requerida para lembrar todas as expansões.
- `displace` – A primeira vez de uma chamada particular é avaliada, a expansão é substituída pela chamada. Isso requer levemente menos armazenagem que quando `macroexpansion` é escolhida para `expand` e é razoávelmente rápido, mas tem a desvantagem de a macro original ser lentamente lembrada e daí a expansão será vista se `display` ou `grind` for chamada. Veja a documentação para `translate` e `macros` para maiores detalhes.

`mode_checkp` [Variável de opção]

Valor por omissão: `true`

Quando `mode_checkp` é `true`, `mode_declare` verifica os modos de associação de variáveis.

`mode_check_errorp` [Variável de opção]

Valor por omissão: `false`

Quando `mode_check_errorp` é `true`, `mode_declare` chama a função "error".

`mode_check_warnp` [Variável de opção]

Valor por omissão: `true`

Quando `mode_check_warnp` é `true`, modo "errors" são descritos.

`mode_declare` (*y_1*, *mode_1*, ..., *y_n*, *mode_n*) [Função]

`mode_declare` é usado para declarar os modos de variáveis e funções para subsequênte tradução ou compilação das funções. `mode_declare` é tipicamente colocada no início de uma definição de função, no início de um script Maxima, ou executado através da linha de comando de forma interativa.

Os argumentos de `mode_declare` são pares consistindo de uma variável e o modo que é um de `boolean`, `fixnum`, `number`, `rational`, ou `float`. Cada variável pode também ser uma lista de variáveis todas as quais são declaradas para ter o mesmo modo.

Se uma variável é um array, e se todo elemento do array que é referenciado tiver um valor então `array (yi, complete, dim1, dim2, ...)` em lugar de

```
array(yi, dim1, dim2, ...)
```

deverá ser usado primeiro declarando as associações do array. Se todos os elementos do array estão no modo `fixnum` (`float`), use `fixnum` (`float`) em lugar de `complete`. Também se todo elemento do array está no mesmo modo, digamos `m`, então

```
mode_declare (completearray (yi), m))
```

deverá ser usado para uma tradução eficiente.

Código numéricos usando arrays podem rodar mais rápidamente se for decladado o tamanho esperado do array, como em:

```
mode_declare (completearray (a [10, 10]), float)
```

para um array numérico em ponto flutuante que é 10 x 10.

Pode-se declarar o modo do resultado de uma função usando `function (f_1, f_2, ...)` como um argumento; aqui `f_1`, `f_2`, ... são nomes de funções. Por exemplo a expressão,

```
mode_declare ([function (f_1, f_2, ...)], fixnum)
```

declara que os valores retornados por `f_1`, `f_2`, ... são inteiros palavra simples.

`modedeclare` é um sinônimo para `mode_declare`.

`mode_identity` (*arg_1*, *arg_2*) [Função]

Uma forma especial usada com `mode_declare` e `macros` para declarar, e.g., uma lista de listas de números em ponto flutuante ou outros objectos de dados. O primeiro argumento para `mode_identity` é um valor primitivo nome de modo como dado para `mode_declare` (i.e., um de `float`, `fixnum`, `number`, `list`, ou `any`), e o segundo argumento é uma expressão que é avaliada e retornada com o valor de `mode_identity`. No entanto, se o valor de retorno não é permitido pelo modo declarado no primeiro argumento, um erro ou alerta é sinalizado. Um ponto importante é que o modo da expressão como determinado pelo Maxima para o tradutor Lisp, será aquele dado como o primeiro argumento, independente de qualquer coisa que vá no segundo argumento. E.g., `x: 3.3; mode_identity (fixnum, x);` retorna um erro. `mode_identity (flonum, x)` returns 3.3 . Isto tem númerosas utilidades, por exemplo, se souber que `first (l)` retornou um número então poderá escrever `mode_identity (number, first (l))`. No entanto, um caminho mais eficiente para fazer a mesma coisa é definir uma nova primitiva,

```
firstnumb (x) ::= buildq ([x], mode_identity (number, first(x)));
```

e usar `firstnumb` sempre que obtiver o primeiro de uma lista de números.

`transcompile` [Variável de opção]

Valor por omissão: `true`

Quando `transcompile` é `true`, `translate` e `translate_file` geram declarações para fazer o código traduzido mais adequado para compilação.

`compfile` escolhe `transcompile: true` para a duração.

`translate` (*f_1*, *...*, *f_n*) [Função]
`translate` (*funções*) [Função]
`translate` (*all*) [Função]

Traduz funções definidas pelo utilizador *f_1*, ..., *f_n* da linguagem de Maxima para Lisp e avalia a tradução Lisp. Tipicamente as funções traduzidas executam mais rápido que as originais.

`translate (all)` ou `translate (funções)` traduz todas as funções definidas pelo utilizador.

Funções a serem traduzidas incluir˜ao uma chamada para `mode_declare` no início quando possível com o objectivo de produzir um código mais eficiente. Por exemplo:

```
f (x_1, x_2, ...) := block ([v_1, v_2, ...],
    mode_declare (v_1, mode_1, v_2, mode_2, ...), ...)
```

quando *x_1*, *x_2*, ... são parâmetros para a função e *v_1*, *v_2*, ... são variáveis locais.

Os nomes de funções traduzidas são removidos da lista `functions` se `savedef` é `false` (veja abaixo) e são adicionados nas listas `props`.

Funções não poderão ser traduzidas a menos que elas sejam totalmente depuradas.

Expressões são assumidas simplificadas; se não forem, um código correcto será gerado mas não será um código óptimo. Dessa forma, o utilizador não poderá escolher o comutador `simp` para `false` o qual inibe simplificação de expressões a serem traduzidas.

O comutador `translate`, se `true`, causa tradução automatica de uma função de utilizador para Lisp.

Note que funções traduzidas podem não executar identicamente para o caminho que elas faziam antes da tradução como certas incompatabilidades podem existir entre o Lisp e versões do Maxima. Principalmente, a função `rat` com mais de um argumento e a função `ratvars` não poderá ser usada se quaisquer variáveis são declaradas com `mode_declare` como sendo expressões rotacionais canónicas(CRE). Também a escolha `prederror: false` não traduzirá.

`savedef` - se `true` fará com que a versão Maxima de uma função utilizador permaneça quando a função é traduzida com `translate`. Isso permite a que definição seja mostrada por `dispfun` e autoriza a função a ser editada.

`transrun` - se `false` fará com que a versão interpretada de todas as funções sejam executadas (desde que estejam ainda disponíveis) em lugar da versão traduzida.

O resultado retornado por `translate` é uma lista de nomes de funções traduzidas.

`translate_file` (*maxima_nomeficheiro*) [Função]
`translate_file` (*maxima_nomeficheiro*, *lisp_nomeficheiro*) [Função]

Traduz um ficheiro com código Maxima para um ficheiro com código Lisp. `translate_file` retorna uma lista de três nomes de ficheiro: O nome do ficheiro Maxima, o nome do ficheiro Lisp, e o nome do ficheiro contendo informações adicionais sobre a tradução. `translate_file` avalia seus argumentos.

`translate_file ("foo.mac"); load("foo.LISP")` é o mesmo que `batch ("foo.mac")` excepto por certas restrições, o uso de `''` e `%`, por exemplo.

`translate_file` (*maxima_nomeficheiro*) traduz um ficheiro Maxima *maxima_nomeficheiro* para um similarmente chamado ficheiro Lisp. Por exemplo, `foo.mac` é traduzido em `foo.LISP`. O nome de ficheiro Maxima pod incluir nome ou nomes de directório(s), nesse caso o ficheiro de saída Lisp é escrito para o mesmo directório que a entrada Maxima.

`translate_file` (*maxima_nomeficheiro*, *lisp_nomeficheiro*) traduz um ficheiro Maxima *maxima_nomeficheiro* em um ficheiro Lisp *lisp_nomeficheiro*. `translate_file` ignora a extensão do nome do ficheiro, se qualquer, de `lisp_nomeficheiro`; a extensão do ficheiro de saída Lisp é sempre `LISP`. O nome de ficheiro Lisp pode incluir

um nome ou nomes de directórios), nesse caso o ficheiro de saída Lisp é escrito para o directório especificado.

`translate_file` também escreve um ficheiro de mensagens de alerta do tradutor em vários graus de severidade. A extensão do nome de ficheiro desse ficheiro é `UNLISP`. Esse ficheiro pode conter informação valiosa, apesar de possivelmente obscura, para rastrear erros no código traduzido. O ficheiro `UNLISP` é sempre escrito para o mesmo directório que a entrada Maxima.

`translate_file` emite código Lisp o qual faz com que algumas definições tenham efeito tão logo o código Lisp é compilado. Veja `compile_file` para mais sobre esse tópico.

Veja também `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`, e `tr_warn_undefined_variable`.

`transrun` [Variável de opção]

Valor por omissão: `true`

Quando `transrun` é `false` fará com que a versão interpretada de todas as funções sejam executadas (desde que estejam ainda disponíveis) em lugar de versão traduzidas.

`tr_array_as_ref` [Variável de opção]

Valor por omissão: `true`

Se `translate_fast_arrays` for `false`, referências a arrays no Código Lisp emitidas por `translate_file` são afectadas por `tr_array_as_ref`. Quando `tr_array_as_ref` é `true`, nomes de arrays são avaliados, de outra forma nomes de arrays aparecem como símbolos literais no código traduzido.

`tr_array_as_ref` não terão efeito se `translate_fast_arrays` for `true`.

`tr_bound_function_applyp` [Variável de opção]

Valor por omissão: `true`

Quando `tr_bound_function_applyp` for `true`, Maxima emite um alerta se uma associação de variável (tal como um argumento de função) é achada sendo usada como uma função. `tr_bound_function_applyp` não afecta o código gerado em tais casos.

Por exemplo, uma expressão tal como `g (f, x) := f (x+1)` irá disparar a mensagem de alerta.

`tr_file_tty_messagesp` [Variável de opção]

Valor por omissão: `false`

Quando `tr_file_tty_messagesp` é `true`, messagens geradas por `translate_file` durante a tradução de um ficheiro são mostradas sobre o console e inseridas dentro do ficheiro UNLISP. Quando `false`, messagens sobre traduções de ficheiros são somente inseridas dentro do ficheiro UNLISP.

`tr_float_can_branch_complex` [Variável de opção]

Valor por omissão: `true`

Diz ao tradutor Maxima-para-Lisp assumir que as funções `acos`, `asin`, `asec`, e `acsc` podem retornar resultados complexos.

O efeito ostensivo de `tr_float_can_branch_complex` é mostrado adiante. Todavia, parece que esse sinalizador não tem efeito sobre a saída do tradutor.

Quando isso for `true` então `acos(x)` será do modo `any` sempre que `x` for do modo `float` (como escolhido por `mode_declare`). Quando `false` então `acos(x)` será do modo `float` se e somente se `x` for do modo `float`.

`tr_function_call_default` [Variável de opção]

Valor por omissão: `general`

`false` significa abandonando e chamando `meval`, `expr` significa que Lisp assume função de argumento fixado. `general`, o código padrão dado como sendo bom para `mexprs` e `mlexprs` mas não `macros`. `general` garante que associações de variável são correctas em códigos compilados. No modo `general`, quando traduzindo F(X), se F for uma variável associada, então isso assumirá que `apply (f, [x])` é significativo, e traduz como tal, com o alerta apropriado. Não é necessário desabilitar isso. Com as escolhas padrão, sem mensagens de alerta implica compatibilidade total do código traduzido e compilado com o interpretador Maxima.

`tr_numer` [Variável de opção]

Valor por omissão: `false`

Quando `tr_numer` for `true` propriedades `numer` são usadas para átomos que possuem essa propriedade, e.g. `%pi`.

`tr_optimize_max_loop` [Variável de opção]

Valor por omissão: 100

`tr_optimize_max_loop` é número máximo de vezes do passo de macro-expansão e optimização que o tradutor irá executar considerando uma forma. Isso é para capturar erros de expansão de macro, e propriedades de optimização não terminadas.

`tr_semicompile` [Variável de opção]

Valor por omissão: `false`

Quando `tr_semicompile` for `true`, as formas de saída de `translate_file` e `compfile` serão macroexpandidas mas não compiladas em código de máquina pelo compilador Lisp.

`tr_state_vars` [Variável de sistema]

Valor por omissão:

```
[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]
```

A lista de comutadores que afectam a forma de saída da tradução. Essa informação é útil para sistemas populares quando tentam depurar o tradutor. Comparando o produto traduzido para o qual pode ter sido produzido por um dado estado, isso é possível para rastrear erros.

`tr_warnings_get ()` [Função]
Imprime uma lista de alertas que podem ter sido dadas pelo tradutor durante a tradução corrente.

`tr_warn_bad_function_calls` [Variável de opção]
Valor por omissão: `true`

- Emite um alerta quando chamadas de função estão sendo feitas por um caminho que pode não ser correcto devido a declarações impróprias que foram feitas em tempo de tradução.

`tr_warn_fexpr` [Variável de opção]
Valor por omissão: `compfile`

- Emite um alerta se quaisquer FEXPRs forem encontradas. FEXPRs não poderão normalmente ser saída em código traduzido, todas as formas de programa especial legítimo são traduzidas.

`tr_warn_meval` [Variável]
Valor por omissão: `compfile`

- Emite um alerta se a função `meval` recebe chamadas. Se `meval` é chamada isso indica problemas na tradução.

`tr_warn_mode` [Variável]
Valor por omissão: `all`

- Emite um alerta quando a variáveis forem atribuídos valores inapropriados para seu modo.

`tr_warn_undeclared` [Variável de opção]
Valor por omissão: `compile`

- Determina quando enviar alertas sobre variáveis não declaradas para o TTY.

`tr_warn_undefined_variable` [Variável de opção]
Valor por omissão: `all`

- Emite um alerta quando variáveis globais indefinidas forem vistas.

`compile_file` (*`nomeficheiro`*) [Função]
`compile_file` (*`nomeficheiro`*, *`nomeficheiro_compilado`*) [Função]
`compile_file` (*`nomeficheiro`*, *`nomeficheiro_compilado`*, *`lisp_nomeficheiro`*) [Função]
Traduz o ficheiro Maxima *nomeficheiro* para Lisp, executa o compilador Lisp, e, se a tradução e a compilação obtiverem sucesso, chama o código compilado dentro do Maxima.

`compile_file` retorna uma lista dos nomes de quatro ficheiros: o ficheiro original do Maxima, o nome da tradução Lisp, uma ficheiro de notas sobre a tradução, e o nome do ficheiro que contém o código compilado. Se a compilação falhar, o quarto item é `false`.

Algumas declarações e definições passam a ter efeito tão logo o código Lisp seja compilado (sem que seja necessário chamar o código compilado). Isso inclui funções

definidas com o operador `:=`, macros definidas com o operador `::=`, `alias`, `declare`, `define_variable`, `mode_declare`, e `infix`, `matchfix`, `nofix`, `postfix`, `prefix`, e `compfile`.

Atribuições e chamadas de função não serão avaliadas até que o código compilado seja carregado. Em particular, dentro do ficheiro Maxima, atribuições para sinalizadores traduzidos (`tr_numer`, etc.) não têm efeito sobre a tradução.

nomeficheiro pode não conter declarações `:lisp`.

`compile_file` avalia seus argumentos.

`declare_translated` (*f_1*, *f_2*, ...) [Função]

Quando traduzindo um ficheiro do código Maxima para Lisp, é importante para o programa tradutor saber quais funções no ficheiro são para serem chamadas como funções traduzidas ou compiladas, e quais outras são apenas funções Maxima ou indefinidas. Colocando essa declaração no topo do ficheiro, faremos conhecido que embora um símbolo diga que não temos ainda um valor de função Lisp, teremos uma em tempo de chamada. `(MFUNCTION-CALL fn arg1 arg2 ...)` é gerado quando o tradutor n˜ao sabe que `fn` está sendo compilada para ser uma função Lisp.

40 Fluxo de Programa

40.1 Introdução a Fluxo de Programa

Maxima fornece um `do` para ciclos iterativos, também contruções mais primitivas tais como `go`.

40.2 Definições para Fluxo de Programa

`backtrace ()` [Função]
`backtrace (n)` [Função]

Imprime a pilha de chamadas, que é, a lista de funções que foram chamadas pela função correntemente activa.

`backtrace()` imprime toda a pilha de chamadas.

`backtrace (n)` imprime as n mais recentes chamadas a funções, incluindo a função correntemente activa.

`backtrace` pode ser chamada por um script, uma função, ou a partir da linha de comando interativa (não somente em um contexto de depuração).

Exemplos:

- `backtrace()` imprime toda a pilha de chamadas.

```
(%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)` imprime as n mais recentes chamadas a funções, incluindo a função correntemente activa.

```
(%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` [Operador especial]

A declaração `do` é usada para executar iteração. Devido à sua grande generalidade a declaração `do` será descrita em duas partes. Primeiro a forma usual será dada que é análoga à forma que é usada em muitas outras linguagens de programação (Fortran, Algol, PL/I, etc.); em segundo lugar os outros recursos serão mencionados.

Existem três variantes do operador especial `do` que diferem somente por suas condições de encerramento. São elas:

- `for` *`Variável`*`:` *`valor_inicial`* `step` *`incremento`* `thru` *`limite`* `do` *`corpo`*
- `for` *`Variável`*`:` *`valor_inicial`* `step` *`incremento`* `while` *`condition`* `do` *`corpo`*
- `for` *`Variável`*`:` *`valor_inicial`* `step` *`incremento`* `unless` *`condition`* `do` *`corpo`*

(Alternativamente, o `step` pode ser dado após a condição de encerramento ou limite.)

valor_inicial, *incremento*, *limite*, e *corpo* podem ser quaisquer expressões. Se o incremento for 1 então `"step 1"` pode ser omitido.

A execução da declaração `do` processa-se primeiro atribuindo o valor_inicial para a variável (daqui em diante chamada a variável de controle). Então: (1) Se a variável de controle excede o limite de uma especificação `thru`, ou se a condição de `unless` for `true`, ou se a condição de `while` for `false` então o `do` será encerrado. (2) O corpo é avaliado. (3) O incremento é adicionado à variável de controle. O processo de (1) a (3) é executado repetidamente até que a condição de encerramento seja satisfeita. Pode-se também dar muitas condições de encerramento e nesse caso o `do` termina quando qualquer delas for satisfeita.

Em geral o teste `thru` é satisfeito quando a variável de controle for maior que o limite se o incremento for não negativo, ou quando a variável de controle for menor que o limite se o incremento for negativo. O incremento e o limite podem ser expressões não numéricas enquanto essa desigualdade puder ser determinada. Todavia, a menos que o incremento seja sintaticamente negativo (e.g. for um número negativo) na hora em que a declaração `do` for iniciada, Maxima assume que o incremento e o limite serão positivos quando o `do` for executado. Se o limite e o incremento não forem positivos, então o `do` pode não terminar propriamente.

Note que o limite, incremento, e condição de encerramento são avaliados cada vez que ocorre um ciclo. Dessa forma se qualquer desses for responsável por muitos cálculos, e retornar um resultado que não muda durante todas as execuções do corpo, então é mais eficiente escolher uma variável para seu valor anterior para o `do` e usar essa variável na forma `do`.

O valor normalmente retornado por uma declaração `do` é o átomo `done`. Todavia, a função `return` pode ser usada dentro do corpo para sair da delcaração `do` prematuramente e dar a isso qualquer valor desejado. Note todavia que um `return` dentro de um `do` que ocorre em um `block` encerrará somente o `do` e não o `block`. Note também que a função `go` não pode ser usada para sair de dentro de um `do` dentro de um `block` que o envolve.

A variável de controle é sempre local para o `do` e dessa forma qualquer variável pode ser usada sem afectar o valor de uma variável com o mesmo nome fora da declaração `do`. A variável de controle é liberada após o encerramento da declaração `do`.

```
(%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 que a condição `while i <= 10` é equivalente a `unless i > 10` e também `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
```

que fornece 8 termos da série de Taylor para `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
```

Esse exemplo calcula a raíz quadrada negativa de 10 usando a iteração de Newton-Raphson um maximum de 10 vezes. Caso o critério de convergêcia não tenha sido encontrado o valor retornado pode ser `done`. Em lugar de sempre adicionar uma

quantidade à variável de controle pode-se algumas vezes desejar alterar isso de alguma outra forma para cada iteração. Nesse caso pode-se usar `next` *`expressão`* em lugar de `step` *`incremento`*. Isso fará com que a variável de controle seja escolhida para o resultado da expressão de avaliação cada vez que o ciclo de repetição for executado.

```
(%i6) for count: 2 next 3*count thru 20 do display (count)$
                              count = 2

                              count = 6

                             count = 18
```

Como uma alternativa para `for` *`Variável`*`:` *`valor`* `...do...` a sintaxe `for` *`Variável`* `from` *`valor`* `...do...` pode ser usada. Isso permite o `from` *`valor`* ser colocado após o `step` ou proximo valor ou após a condição de encerramento. Se `from` *`valor`* for omitido então 1 é usado como o valor inicial.

Algumas vezes se pode estar interessado em executar uma iteração onde a variável de controle nunca seja usada. Isso é permissível para dar somente as condições de encerramento omitindo a inicialização e a informação de actualização como no exemplo seguinte para para calcular a raíz quadrada de 5 usando uma fraca suposição inicial.

```
(%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
```

Se isso for desejado pode-se sempre omitir as condições de encerramento inteiramente e apenas dar o corpo `do` *`corpo`* que continuará a ser avaliado indefinidamente. Nesse caso a função `return` será usada para encerrar a execução da declaração `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 que `return`, quando executado, faz com que o valor corrente de `x` seja retornado como o valor da declaração `do`. O `block` é encerrado e esse valor da declaração `do` é retornado como o valor do `block` porque o `do` é a última declaração do `block`.)

Uma outra forma de `do` é disponível no Maxima. A sintaxe é:

`for` *`Variável`* `in` *`list end_tests`* `do` *`corpo`*

Os elementos de *list* são quaisquer expressões que irão sucessivamente ser atribuídas para a variável a cada iteração do corpo. O teste opcional *end_tests* pode ser usado para encerrar a execução da declaração `do`; de outra forma o `do` terminará quando a lista for exaurida ou quando um `return` for executado no corpo. (De facto, a lista pode ser qualquer expressão não atômica, e partes sucessivas são usadas.)

```
(%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*) [Função]

Avalia *expr_1*, ..., *expr_n* uma por uma e retorna `[`*expr_n*`]` (uma lista) se nenhum erro ocorrer. Se um erro ocorrer na avaliação de qualquer argumento, `errcatch` evita que o erro se propague e retorna a lista vazia `[]` sem avaliar quaisquer mais argumentos.

`errcatch` é útil em ficheiros `batch` onde se suspeita que um erro possa estar ocorrendo o `errcatch` terminará o `batch` se o erro não for detectado.

`error` (*expr_1*, ..., *expr_n*) [Função]

`error` [Variável de sistema]

Avalia e imprime *expr_1*, ..., *expr_n*, e então causa um retorno de erro para o nível mais alto do Maxima ou para o mais próximo contendo `errcatch`.

A variável `error` é escolhida para uma lista descrevendo o erro. O primeiro elemento de `error` é uma sequência de caracteres de formato, que junta todas as sequências de caracteres entre os argumentos *expr_1*, ..., *expr_n*, e os elementos restantes são os valores de quaisquer argumentos que não são sequências de caracteres.

`errormsg()` formata e imprime `error`. Isso efectivamente reimprime a mais recente mensagem de erro.

`errormsg` () [Função]

Reimprime a mais recente mensagem de erro. A variável `error` recebe a mensagem, e `errormsg` formata e imprime essa mensagem.

`for` [Operador especial]

Usado em iterações. Veja `do` para uma descrição das facilidades de iteração do Maxima.

`go` (*tag*) [Função]

é usada dentro de um `block` para transferir o controle para a declaração do bloco que for identificada com o argumento para `go`. Para identificar uma declaração, coloque antes dessa declaração um argumento atômico como outra declaração no `block`. Por exemplo:

```
block ([x], x:1, loop, x+1, ..., go(loop), ...)
```

O argumento para `go` deve ser o nome de um identificardor aparecendo no mesmo `block`. Não se pode usar `go` para transferir para um identificador em um outro `block` que não seja o próprio contendo o `go`.

`if` [Operador especial]

Representa avaliação condicional. Várias formas de expressões `if` são reconhecidas. `if` *cond_1* `then` *expr_1* `else` *expr_0* avalia para *expr_1* se *cond_1* avaliar para `true`, de outra forma a expressão avalia para `expr_0`.

`if` *cond_1* `then` *expr_1* `elseif` *cond_2* `then` *expr_2* `elseif ... else` *expr_0*
avalia para *expr_k* se *cond_k* for `true` e todas as condições precedentes forem `false`. Se nenhuma das condições forem `true`, a expressão avalia para `expr_0`.

O comportamento `else false` é assumido se `else` for omitido. Isso é, `if` *cond_1* `then` *expr_1* é equivalente a `if` *cond_1* `then` *expr_1* `else false`, e `if` *cond_1* `then` *expr_1* `elseif ... elseif` *cond_n* `then` *expr_n* é equivalente a `if` *cond_1* `then` *expr_1* `elseif ... elseif` *cond_n* `then` *expr_n* `else false`.

As alternativas *expr_0*, ..., *expr_n* podem ser quaisquer expressões do Maxima, incluíndo expressões `if` aninhadas (if dentro de if). As alternativas não são nem simplificadas nem avaliadas a menos que a correspondente condição seja `true`.

As condições *cond_1*, ..., *cond_n* são expressões tais que `is(`*cond_k*`)` avaliem para `true` ou para `false`; de outra forma é um erro. Entre outros elementos, condições podem compreender operadores lógicos e relacionais como segue.

Operação	Símbolo	Tipo
menor que	<	infixo relacional
menor que ou igual a	<=	infixo relacional
igualdade (sintática)	=	infixo relacional
negação de =	#	infixo relacional
igualdade (valor)	equal	função relacional
negação de igualdade	notequal	função relacional
maior que ou igual a	>=	infixo relacional
maior que	>	infixo relacional
e	and	infixo lógico
ou	or	infixo lógico
não	not	prefixo lógico

`map` (*f*, *expr_1*, ..., *expr_n*) [Função]
Retorna uma expressão cujo operador principal é o mesmo que o das expressões *expr_1*, ..., *expr_n* mas cujas subpartes são os resultados da aplicação de *f* nas correspondentes subpartes das expressões. *f* é ainda o nome de uma função de *n* argumentos ou é uma forma `lambda` de *n* argumentos.

`maperror` - se `false` fará com que todas as funções mapeadas (1) parem quando elas terminarem retornando a menor expi se não forem todas as expi do mesmo comprimento e (2) aplique fn a [exp1, exp2,...] se expi não forem todas do mesmo tipo de objecto. Se `maperror` for `true` então uma mensagem de erro será dada nas duas instâncias acima.

Um dos usos dessa função é para mapear (`map`) uma função (e.g. `partfrac`) sobre cada termo de uma expressão muito larga onde isso comumente não poderia ser possível usar a função sobre a expressão inteira devido a uma exaustão de espaço da lista de armazenamento no decorrer da computação.

```
(%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*) [Função]

Retorna `true` se e somente se *expr* for tratada pelas rotinas de mapeamento como um átomo. "Mapatoms" são átomos, números (incluíndo números racioanais), e variáveis subscritas.

`maperror` [Variável de opção]

Valor por omissão: `true`

Quando `maperror` é `false`, faz com que todas as funções mapeadas, por exemplo

```
map (f, expr_1, expr_2, ...))
```

(1) parem quando elas terminarem retornando a menor expi se não forem todas as expi do mesmo comprimento e (2) aplique `f` a `[expr_1, expr_2, ...]` se `expr_i` não forem todas do mesmo tipo de objecto.

Se `maperror` for `true` então uma ,mensagem de erro é mostrada nas duas instâncias acima.

`maplist` (*f*, *expr_1*, ..., *expr_n*) [Função]

Retorna uma lista de aplicações de *f* em todas as partes das expressões *expr_1*, ..., *expr_n*. *f* é o nome de uma função, ou uma expressão lambda.

`maplist` difere de `map (f, expr_1, ..., expr_n)` que retorna uma expressão com o mesmo operador principal que *expr_i* tem (excepto para simplificações e o caso onde `map` faz um `apply`).

`prederror` [Variável de opção]

Valor por omissão: `true`

Quando `prederror` for `true`, uma mensagem de erro é mostrada sempre que o predicado de uma declaração `if` ou uma função `is` falha em avaliar ou para `true` ou para `false`.

Se `false`, `unknown` é retornado no lugar nesse caso. O modo `prederror: false` não é suportado no código traduzido; todavia, `maybe` é suportado no código traduzido.

Veja também `is` e `maybe`.

`return` (*valor*) [Função]

Pode ser usada para sair explicitamente de um bloco, levando seu argumento. Veja `block` para mais informação.

`scanmap` (*f*, *expr*) [Função]
`scanmap` (*f*, *expr*, *bottomup*) [Função]

Recursivamente aplica *f* a *expr*, de cima para baixo. Isso é muito útil quando uma factoração completa é desejada, por exemplo:

```
(%i1) exp:(a^2+2*a+1)*y + x^2$
(%i2) scanmap(factor,exp);
                                        2      2
(%o2)                            (a + 1)  y + x
```

Note o caminho através do qual `scanmap` aplica a dada função `factor` para as subexpressões constituintes de *expr*; se outra forma de *expr* é apresentada para `scanmap` então o resultado pode ser diferente. Dessa forma, `%o2` não é recuperada quando `scanmap` é aplicada para a forma expandida de exp:

```
(%i3) scanmap(factor,expand(exp));
                              2                   2
(%o3)                         a  y + 2 a y + y + x
```

Aqui está um outro exemplo do caminho no qual `scanmap` aplica recursivamente uma função dada para todas as subexpressões, incluindo expoentes:

```
(%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`) aplica *f* a *expr* de baixo para cima. E.g., para `f` indefinida,

```
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))
```

Neste caso, obtém-se a mesma resposta pelos dois métodos.

`throw` (*expr*) [Função]

Avalia *expr* e descarta o valor retornado para o mais recente `catch`. `throw` é usada com `catch` como um mecanismo de retorno não local.

`while` [Operador especial]

Veja `do`.

`outermap` (*f*, *a_1*, ..., *a_n*) [Função]

Aplica a função *f* para cada um dos elementos do produto externo *a_1* vezes *a_2* ... vezes *a_n*.

f é o nome de uma função de *n* argumentos ou uma expressão lambda de *n* argumentos. Cada argumento *a_k* pode ser uma lista simples ou lista aninhada (lista contendo listas como elementos), ou uma matrz, ou qualquer outro tip de expressão.

O valor de retorno de `outermap` é uma estrutura aninhada. Tomemos x como sendo o valor de retorno. Então x tem a mesma estrutura da primeira lista, lista aninhada, ou argumento matriz, `x[i_1]...[i_m]` tem a mesma estrutura que a segunda lista, lista aninhada, ou argumento matriz, `x[i_1]...[i_m][j_1]...[j_n]` tem a mesma estrutura que a terceira lista, lista aninhada, ou argumento matriz, e assim por diante, onde m, n, ... são os números dos índices requeridos para acessar os elementos de cada argumento (um para uma lista, dois para uma matriz, um ou mais para uma lista aninhada). Argumentos que não forem listas ou matrizes não afectam a estrutura do valor de retorno.

Note que o efeito de `outermap` é diferente daquele de aplicar f a cada um dos elementos do produto externo retornado por `cartesian_product`. `outermap` preserva a estrutura dos argumentos no valor de retorno, enquanto `cartesian_product` não reserva essa mesma estrutura.

`outermap` avalia seus argumentos.

Veja também `map`, `maplist`, e `apply`.

Exemplos: Exemplos elementares de `outermap`. Para mostrar a a combinação de argumentos mais claramente, `F` está indefinida à esquerda.

```
(%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]]
```

Uma explanação final do valor de retorno de `outermap`. Os argumentos primeiro, segundo, e terceiro são matriz, lista, e matriz, respectivamente. O valor de retorno é uma matriz. Cada elementos daquela matriz é uma lista, e cada elemento de cada lista é uma matriz.

```
(%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 Depuração

41.1 Depuração do Código Fonte

Maxima tem um depurador interno de código fonte. O utilizador pode escolher um ponto de parada numa função, e executar o programa linha por linha a partir daí. A pilha de chamadas pode ser examinada, juntamente com as variáveis existentes nesse nível.

O comando `:help` ou `:h` mostra a lista de comandos de depuração. (Em geral, os comandos podem ser abreviados se a abreviação for única. Se não for única, as alternativas podem ser listadas.) Dentro do depurador, o utilizador pode também usar quaisquer funções comuns do Maxima para examinar, definir, e manipular variáveis e expressões.

Um ponto de parada é escolhido através do comando `:br` na linha de comando do Maxima. Dentro do depurador, o utilizador pode avançar uma linha de cada vez usando o comando `:n` ("next"). o comando `:bt` ("backtrace") mostra uma lista da pilha de frames. O comando `:r` ("resume") sai do depurador e continua com a execução. Esses comandos são demostrados no exemplo a seguir.

```
(%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 digitado aqui lista os frames
#0: foo(y=5)(foobar.mac line 1)
#1: bar(x=2,y=3)(foobar.mac line 9)

(dbm:1) :n                           <-- Aqui digite :n para avançar linha
(foobar.mac 2)
/tmp/foobar.mac:2::

(dbm:1) :n                           <-- Aqui digite :n para avançar linha
(foobar.mac 3)
/tmp/foobar.mac:3::

(dbm:1) u;                           <-- Investiga o valor de u
28

(dbm:1) u: 33;                       <-- Altera u para ser 33
33
```

```
(dbm:1) :r                          <-- Digite :r para retomar a computação

(%o2)                                 1094
```

O ficheiro `/tmp/foobar.mac` é o seguinte:

```
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);
```

USO DO DEPURADOR ATRAVÉS DO EMACS E DE XMAXIMA

Se o utilizador estiver a executar o código sob o GNU Emacs numa janela shell (shel dbl), ou estiver usando a interface gráfica, `xmaxima`, então quando parar num ponto de parada, verá a sua posição actual no ficheiro fonte apresentada na outra metade da janela, ou em vermelho brilhante, ou com uma pequena seta apontando na direita da linha. Poderá avançar uma linha por vez digitando M-n (Alt-n).

No Emacs pode executar o Maxima numa shell `dbl`, o qual requer o ficheiro `dbl.el` no directório elisp. Verifique que tenha instalado os ficheiros elisp ou adicionado o directório elisp do Maxima ao seu caminho: e.g., adicione o seguinte ao seu ficheiro `.emacs` ou ao seu ficheiro `site-init.el`

```
(setq load-path (cons "/usr/share/maxima/5.9.1/emacs" load-path))
(autoload 'dbl "dbl")
```

então no Emacs

```
M-x dbl
```

pode iniciar uma janela shell na qual pode executar programas, por exemplo Maxima, gcl, gdb etc. Essa janela de shell também reconhece informações sobre depuração de código fonte, e mostra o código fonte em outra janela.

O utilizador pode escolher um ponto de parada em certa linha do ficheiro digitando `C-x space`. Isso encontra qual a função onde o cursor está posicionado, e mostra qual a linha daquela função onde o cursor está habilitado. Se o cursor estiver habilitado, digamos, na linha 2 de `foo`, então isso irá inserir na outra janela o comando, "`:br foo 2`", para parar `foo` nessa segunda linha. Para ter isso habilitado, o utilizador deve ter maxima-mode.el habilitado na janela na qual o ficheiro `foobar.mac` estiver interagindo. Existe comandos adicional disponíveis naquela janela de ficheiro, tais como avaliando a função dentro do Maxima, através da digitação de `Alt-Control-x`.

41.2 Comandos Palavra Chave

Comandos palavra chave são palavras chaves especiais que não são interpretadas como expressões do Maxima. Um comando palavra chave pode ser inserido na linha de comando do Maxima ou na linha de comando do depurador, embora não possa ser inserido na linha

de comando de parada. Comandos palavra chave iniciam com um dois pontos Keyword commands start with a colon, ':'. Por exemplo, para avaliar um comando do Lisp, pode escrever `:lisp` seguido pelo comando a ser avaliado.

```
(%i1) :lisp (+ 2 3)
5
```

O número de argumentos necessários depende do comando em particular. Também, não precisa escrever o comando completo, apenas o suficiente para ser único no meio das palavras chave de parada. Dessa forma `:br` será suficiente para `:break`.

Os comandos de palavra chave são listados abaixo.

`:break F n`
: Escolhe um ponto de parada em uma função `F` na linha `n` a partir do início da função. Se `F` for dado como uma sequência de caracteres, então essa sequência de caracteres é assumida referir-se a um ficheiro, e `n` é o deslocamente a partir do início do ficheiro. O deslocamento é opcional. Se for omitido, é assumido ser zero (primeira linha da função ou do ficheiro).

`:bt` Imprime na tela uma lista da pilha de frames

`:continue`
: Continua a computação

`:delete` Remove o ponto de parada seleccionado, ou todos se nenum for especificado

`:disable` Desabilita os pontos de parada seleccionados, ou todos se nenhum for especificado

`:enable` Habilita os pontos de de parada especificados, ou todos se nenhum for especificado

`:frame n` Imprime na tela a pilha de frame `n`, ou o corrente frame se nenhum for especificado

`:help` Imprime na tela a ajuda sobre um comando do depurador, ou todos os comandos se nenhum for especificado

`:info` Imprime na tela informações sobre um item

`:lisp alguma-forma`
: Avalia `alguma-forma` como uma forma Lisp

`:lisp-quiet alguma-forma`
: Avalia a forma Lisp `alguma-forma` sem qualquer saída

`:next` Como `:step`, excepto `:next` passos sobre chamadas de fução

`:quit` Sai do nível corrente do depurador sem concluir a computação

`:resume` Continua a computação

`:step` Continua a computação até encontraruma nova linha de códico

`:top` Retorne para a linha de comando do Maxima (saindo de qualquer nível do depurador) sem completar a computação

41.3 Definições para Depuração

`refcheck` [Variável de opção]

Valor por omissão: `false`

Quando `refcheck` for `true`, Maxima imprime uma mensagem cada vez que uma variável associada for usada pela primeira vez em uma computação.

`setcheck` [Variável de opção]

Valor por omissão: `false`

Se `setcheck` for escolhido para uma lista de variáveis (as quais podem ser subscritas), Maxima mostra uma mensagem quando as variáveis, ou ocorrências subscritas delas, forem associadas com o operador comum de atribuição `:`, o operador `::` de atribuição, ou associando argumentos de função, mas não com o operador de atribuição de função `:=` nem o operador de atribuição `::=` de macro. A mensagem compreende o nome das variáveis e o valor associado a ela.

`setcheck` pode ser escolhida para `all` ou `true` incluindo desse modo todas as variáveis.

Cada nova atribuição de `setcheck` estabelece uma nova lista de variáveis para verificar, e quaisquer variáveis previamente atribuídas a `setcheck` são esquecidas.

Os nomes atribuídos a `setcheck` devem ter um apóstrofo no início se eles forem de outra forma avaliam para alguma outra coisa que não eles mesmo. Por exemplo, se `x`, `y`, e `z` estiverem actualmente associados, então digite

```
setcheck: ['x, 'y, 'z]$
```

para colocá-los na lista de variáveis monitoradas.

Nenhuma saída é gerada quando uma variável na lista `setcheck` for atribuída a sí mesma, e.g., `X: 'X`.

`setcheckbreak` [Variável de opção]

Valor por omissão: `false`

Quando `setcheckbreak` for `true`, Maxima mostrará um ponto de parada quando uma variável sob a lista `setcheck` for atribuída a um novo valor. A parada ocorre antes que a atribuíção seja concluída. Nesse ponto, `setval` retém o valor para o qual a variável está para ser atribuída. Consequentemente, se pode atribuir um valor diferente através da atribuição a `setval`.

Veja também `setcheck` e `setval`.

`setval` [Variável de sistema]

Mantém o valor para o qual a variável está para ser escolhida quando um `setcheckbreak` ocorrer. Consequentemente, se pode atribuir um valor diferente através da atribuição a `setval`.

Veja também `setcheck` e `setcheckbreak`.

`timer` (*f_1*, ..., *f_n*) [Função]

`timer` () [Função]

Dadas as funções *f_1*, ..., *f_n*, `timer` coloca cada uma na lista de funções para as quais cronometragens estatísticas são colectadas. `timer(f)$ timer(g)$` coloca `f` e então `g` sobre a lista; a lista acumula de uma chamada para a chamada seguinte.

Sem argumentos, `timer` retorna a lista das funções tempo estatisticamente monitoradas.

Maxima armazena quanto tempo é empregado executando cada função na lista de funções tempo estatisticamente monitoradas. `timer_info` retorna a coronometragem estatística, incluindo o tempo médio decorrido por chamada de função, o número de chamadas, e o tempo total decorrido. `untimer` remove funções da lista de funções tempo estatisticamente monitoradas.

`timer` não avalia seus argumentos. `f(x) := x^2$ g:f$ timer(g)$` não coloca `f` na lista de funções estatisticamente monitoradas.

Se `trace(f)` está vigorando, então `timer(f)` não tem efeito; `trace` e `timer` não podem ambas atuarem ao mesmo tempo.

Veja também `timer_devalue`.

`untimer (`*f_1*, ..., *f_n*`)` [Função]
`untimer ()` [Função]

Dadas as funções *f_1*, ..., *f_n*, `untimer` remove cada uma das funções listadas da lista de funções estatisticamente monitoradas.

Sem argumentos, `untimer` remove todas as funções actualmente na lista de funções estatisticamente monitoradas.

Após `untimer (f)` ser executada, `timer_info (f)` ainda retorna estatisticas de tempo previamente colectadas, embora `timer_info()` (sem argumentos) não retorna informações sobre qualquer função que não estiver actualmente na lista de funções tempo estatisticamente monitoradas. `timer (f)` reposiciona todas as estatisticas de tempo para zero e coloca `f` na lista de funções estatisticamente monitoradas novamente.

`timer_devalue` [Variável de opção]

Valor Padrão: `false`

Quando `timer_devalue` for `true`, Maxima subtrai de cada função estatisticamente monitorada o tempo empregado em ou funções estatisticamente monitoradas. De outra forma, o tempo reportado para cada função inclui o tempo empregado em outras funções. Note que tempo empregado em funções não estatisticamente monitoradas não é subtraído do tempo total.

Veja também `timer` e `timer_info`.

`timer_info (`*f_1*, ..., *f_n*`)` [Função]
`timer_info ()` [Função]

Dadas as funções *f_1*, ..., *f_n*, `timer_info` retorna uma matriz contendo informações de cronometragem para cada função. Sem argumentos, `timer_info` retorna informações de cronometragem para todas as funções actualmente na lista de funções estatisticamente monitoradas.

A matriz retornada através de `timer_info` contém o nome da função, tempo por chamda de função, número de chamadas a funções,tempo total, e `gctime`, cujja forma "tempo de descarte" no Macsyma original mas agora é sempre zero.

Os dados sobre os quais `timer_info` constrói seu valor de retorno podem também serem obtidos através da função `get`:

```
get(f, 'calls);  get(f, 'runtime);  get(f, 'gctime);
```

Veja também `timer`.

`trace` (*f_1*, ..., *f_n*) [Função]
`trace` () [Função]

Dadas as funções *f_1*, ..., *f_n*, `trace` instrui Maxima para mostrar informações de depuração quando essas funções forem chamadas. `trace(f)$ trace(g)$` coloca `f` e então `g` na lista de funções para serem colocadas sob a ação de `trace`; a lista acumula de uma chamada para a seguinte.

Sem argumentos, `trace` retorna uma lista de todas as funções actualmente sob a ação de `trace`.

A função `untrace` desabilita a ação de `trace`. Veja também `trace_options`.

`trace` não avalia seus argumentos. Dessa forma, `f(x) := x^2$ g:f$ trace(g)$` não coloca `f` sobre a lista de funções monitoradas por `trace`.

Quando uma função for redefinida, ela é removida da lista de `timer`. Dessa forma após `timer(f)$ f(x) := x^2$`, a função `f` não mais está na lista de `timer`.

Se `timer (f)` estiver em efeito, então `trace (f)` não está agindo; `trace` e `timer` não podem ambas estar agindo para a mesma função.

`trace_options` (*f*, *option_1*, ..., *option_n*) [Função]
`trace_options` (*f*) [Função]

Escolhe as opções de `trace` para a função *f*. Quaisquer opções anteriores são substituídas. `trace_options (f, ...)` não tem efeito a menos que `trace (f)` tenha sido também chamada (ou antes ou após `trace_options`).

`trace_options (f)` reposiciona todas as opções para seus valores padrão.

As opções de palavra chave são:

- `noprint` Não mostre uma mensagem na entrada da função e saia.
- `break` Coloque um ponto de parada antes da função ser inserida, e após a funçãos er retirada. Veja `break`.
- `lisp_print` Mostre argumentos e valores de retorno com objectos Lisp.
- `info` Mostre `-> true` na entrada da funçào e saia.
- `errorcatch` Capture os erros, fornecendo a opção para sinalizar um erro, tentar novamente a chamada de função, ou especificar um valor de retorno.

Opções para `trace` são especificadas em duas formas. A presença da palavra chave de opção sozinha coloca a opção para ter efeito incondicionalmente. (Note que opção *foo* não coloca para ter efeito especificando *`foo`*`: true` ou uma forma similar; note também que palavras chave não precisam estar com apóstrofo.) Especificando a opção palavra chave com uma função predicado torna a opção condicional sobre o predicado.

A lista de argumentos para a função predicado é sempre `[level, direction, function, item]` onde `level` é o nível rerecursão para a função, `direction` é ou `enter` ou `exit`, `function` é o nome da função, e `item` é a lista de argumentos (sobre entrada) ou o valor de retorno (sobre a saída).

Aqui está um exemplo de opções incondicionais de `trace`:

```
(%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);
```

Aqui está a mesma função, com a opção `break` condicional sobre um predicado:

```
(%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*) [Função]

`untrace` () [Função]

Dadas as funções *f_1*, ..., *f_n*, `untrace` desabilita a a monitoração habilitada pela função `trace`. Sem argumentos, `untrace` desabilita a atuação da função `trade` para todas as funções.

`untrace` retorne uma lista das funções para as quais `untrace` desabilita a atuação de `trace`.

42 augmented_lagrangian

42.1 Definições para augmented_lagrangian

`augmented_lagrangian_method` (*FOM*, *xx*, *C*, *yy*) [Função]
`augmented_lagrangian_method` (*FOM*, *xx*, *C*, *yy*, *args_opcionais*) [Função]

Retorna um mínimo aproximado da expressão *FOM* com relação às variáveis *xx*, mantendo restrito o valor de *C* a zero. *yy* é uma lista de suposições iniciais para *xx*. O método utilizado é o método do Lagrangiano aumentado (veja referências [1] e [2]).

`args_opcionais` representam argumentos adicionais, especificados como *`símbolo = valor`*. Os argumentos opcionais que podem ser colocados no lugar de *símbolo*:

`niter` Número de iterações do algoritmo do Langrangiano aumentado

`lbfgs_tolerance`
Tolerância forneceida a LBFGS (Limited-memory, Broyden, Fletcher, Goldfarb, Shanno)

`iprint` parâmetro IPRINT (uma lista de dois inteiros que controlam o nível de informação) fornecido a LBFGS

`%lambda` valor inicial de `%lambda` a ser usado durante o cálculo do Lagrangiano aumentado

Essa implementação minimiza o Lagrangiano aumentado pela pela aplicação do algoritmo de memória limitada BFGS (LBFGS), que é um algoritmo quasi-Newton.

`load("augmented_lagrangian")` chama essa função.

Veja também `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

Exemplo:

```
(%i1) load ("lbfgs");
(%o1) /home/robert/tmp/maxima-release-branch/maxima/share/lbfgs/\
lbfgs.mac
(%i2) load ("augmented_lagrangian");
(%o2) /home/robert/tmp/maxima-release-branch/maxima/share/contri\
b/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]
(%o6)                          [1, 1]
(%i7) augmented_lagrangian_method (FOM, xx, C, yy, iprint = [-1, 0]);
```

```
(%o7) [[x = 0.6478349888525, y = 0.32391749442625],
                                  %lambda = [- 1.267422460983745]]
```

43 bode

43.1 Definições para bode

bode_gain (*H*, *range*, *...plot_opts...*) [Função]

Função para desenhar gráficos de ganho para Bode.

Exemplos (1 a 7 provenientes de

http://www.swarthmore.edu/NatSci/echeeve1/Ref/Bode/BodeHow.html,

8 proveniente de 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])$
```

Para usar essa função escreva primeiramente `load("bode")`. Veja também `bode_phase`

`bode_phase` (*H*, *range*, *...plot_opts...*) [Função]

Função para desenhar gráficos de fase para Bode

Exemplos (1 a 7 provenientes de

`http://www.swarthmore.edu/NatSci/echeeve1/Ref/Bode/BodeHow.html`,

8 proveniente de 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]));
```

Para usar essa função escreva primeiramente `load("bode")`. Veja também `bode_gain`

44 descriptive

44.1 Introdução ao pacote descriptive

O pacote `descriptive` contém um conjunto de funções para fazer cálculos de estatística descritiva e desenhar gráficos. Juntamente com o código fonte três conjuntos de dados em suar árvore do Maxima: `pidigits.data`, `wind.data` e `biomed.data`. Eles também podem ser baixados a partir de `www.biomates.net`.

Qualque manual de estatística pode ser usado como referência para as funções no pacote `descriptive`.

Para comentários, erros ou sugestões, por favor entre em contato comigo em *'mario AT edu DOT xunta DOT es'*.

Aqui está um exemplo sobre como as funções de estatística descritiva no pacote `descriptive` fazem esse trabalho, dependendo da natureza de seus argumentos, listas e matrizes,

```
(%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) /* amostra de várias variáveis */ mean (%);
                        e + c + a  f + d + b
(%o4)                  [---------, ---------]
                            3          3
```

Note que em amostras de várias variáveis a média é calculada em cada coluna.

No caso de muitas amostras amostras com possíveis tamanhos diferentes, A função do Maxima `map` pode ser usada para pegar os resultados desejados de cada amostra,

```
(%i1) load ("descriptive")$
(%i2) map (mean, [[a, b, c], [d, e]]);
                         c + b + a  e + d
(%o2)                   [---------, -----]
                            3        2
```

Nesse caso, duas amostras de tamanhos 3 e 2 foram armazenadas em uma lista.

Amostras de uma única variável devem ser armazenadas em listas como

```
(%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]
```

e amostras de várias variáveis em matrizes como em

```
(%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  ]
```

Nesse caso, o número de colunas é igual à dimensão (ao número) de variáveis e o n;umero de linhas é o tamano da amostra.

Dados podem ser introduzidos manualmente, mas grandes amostras são usualmente armazenadas em ficheiros no formato texto plano. Por exemplo, o ficheiro `pidigits.data` contém os primeiros 100 dígitos do número `%pi`:

```
3
1
4
1
5
9
2
6
5
3 ...
```

Com o objectivo de chamar esses dígitos no Maxima,

```
(%i1) load ("numericalio")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) length (s1);
(%o3)                          100
```

Por outro lado, o ficheiro `wind.data` contém a média diária de velocidades do ventoem 5 estações meteorológicas na República da Irlanda (Esses dados são parte de um conjunto de dados tomados em 12 estações meteorológicas. O ficheiro original está disponivel livremente para download no Repositório de Dados StatLib e sua análise é discutida em 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). As linhas seguintes mostram como tornar os dados disponíveis para o Maxima:

```
(%i1) load ("numericalio")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) length (s2);
(%o3)                          100
(%i4) s2 [%]; /* last record */
(%o4)            [3.58, 6.0, 4.58, 7.62, 11.25]
```

Algumas amostras possuem dados não numéricos. Como um exemplo, o ficheiro `biomed.data` (que é parte de outro grande ficheiro tomado do Repósitório de Dados StatLib) contém quatro medidas sanguíneas tomadas de dois grupos de pacientes, `A` e `B`, de diferentes idades,

```
(%i1) load ("numericalio")$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i3) length (s3);
(%o3)                          100
(%i4) s3 [1]; /* first record */
(%o4)            [A, 30, 167.0, 89.0, 25.6, 364]
```

O primeiro indivíduo pertence ao grupo `A`, com 30 anos de idade e suas medidas sanguíneas foram 167.0, 89.0, 25.6 e 364.

Se deve tomar cuidado quando se trabalha com dados divididos por categorias. no exemplo seguinte, ao símbolo `a` é atribuído um valor em algum momento anterior e então a amostra com valores divididos por categoria `a` é interpretada como,

```
(%i1) a : 1$
(%i2) matrix ([a, 3], [b, 5]);
                             [ 1  3 ]
(%o2)                        [      ]
                             [ b  5 ]
```

44.2 Definições para manipulação da dados

`continuous_freq` (*list*) [Função]
`continuous_freq` (*list*, *m*) [Função]

O argumento de `continuous_freq` deve ser uma lista de números, que serão então agrupadas em intervalos e contado quantos desses dados pertencem a cada grupo. Opcionalmente, a função `continuous_freq` admite um segundo argumento indicando o número de classes, 10 é o valor padrão,

```
(%i1) load ("numericalio")$
(%i2) load ("descriptive")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) continuous_freq (s1, 5);
(%o4) [[0, 1.8, 3.6, 5.4, 7.2, 9.0], [16, 24, 18, 17, 25]]
```

A primeira lista contém os limites de intervalos e o segundo a correspondente contagem: existem 16 algarismos da parte decimal de `%pi` dentro do intervalo `[0, 1.8]`, isto é 0's e 1's, 24 algarismos em `(1.8, 3.6]`, isto é 2's e 3's, e assim por diante.

`discrete_freq` (*list*) [Função]

Conta as frequências absolutas em amostras discretas, em amostras numéricas e em amostras divididas em categorias. Seu único argumento é uma lista,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"));
(%o3) [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8,
```

```
4, 6, 2, 6, 4, 3, 3, 8, 3, 2, 7, 9, 5, 0, 2, 8, 8, 4, 1, 9, 7,
1, 6, 9, 3, 9, 9, 3, 7, 5, 1, 0, 5, 8, 2, 0, 9, 7, 4, 9, 4, 4,
5, 9, 2, 3, 0, 7, 8, 1, 6, 4, 0, 6, 2, 8, 6, 2, 0, 8, 9, 9, 8,
6, 2, 8, 0, 3, 4, 8, 2, 5, 3, 4, 2, 1, 1, 7, 0, 6, 7]
(%i4) discrete_freq (s1);
(%o4) [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
                              [8, 8, 12, 12, 10, 8, 9, 8, 12, 13]]
```

A primeira lista fornece os valores da amostra e a segunda as suas frequências absolutas. Os comandos `? col` e `? transpose` podem ajudá-lo a entender o último comando de entrada.

`subsample` (*matriz_de_dados*, *expressão_lógica*) [Função]
`subsample` (*matriz_de_dados*, *expressão_lógica*, *col_num*, *col_num*, ...) [Função]

Essas funções são um tipo de variação da função `submatrix` do Maxima. O primeiro argumento é o nome da matriz de dados, o segundo argumento é uma expressão lógica que recebeu apóstrofo e os argumentos opcionais adicionais são o número de colunas a serem tomadas. Esse comportamento é melhor entendido com exemplos,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) subsample (s2, '(%c[1] > 18));
              [ 19.38  15.37  15.12  23.09  25.25 ]
              [                                    ]
              [ 18.29  18.66  19.08  26.08  27.63 ]
(%o4)         [                                    ]
              [ 20.25  21.46  19.95  27.71  23.38 ]
              [                                    ]
              [ 18.79  18.96  14.46  26.38  21.84 ]
```

Existem registros de várias variáveis nos quais a velocidade do vento na primeira estação meteorológica foram maiores que 18. Veja que na expressão lógica que recebeu apóstrofo o *i*-ésimo componente é referenciado como `%c[i]`. O símbolo `%c[i]` é usado dentro da função `subsample`, portanto quando usado como uma variável de uma categoria, Maxima fica confuso. No seguinte exemplo, requisitamos somente o primeiro, o segundo e o quinto componentes desses registro com velocidades de vento maiores que ou igual a 16 nós na estação meteorológica número 1 e menor que 25 nós na estação meteorológica número 4,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) subsample (s2, '(%c[1] >= 16 and %c[4] < 25), 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 ]
```

Aqui está um exemplo com as variáveis divididas em categorias do ficheiro `biomed.data`. Queremos os registros correspondentes a aqueles pacientes no grupo `B` que possuem idade maior que 38 anos,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s3 : read_matrix (file_search ("biomed.data"))$
(%i4) subsample (s3, '(%c[1] = B and %c[2] > 38));
                [ 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 ]
```

Probavelmente, a análise estatística irá envolver somente as medidas sanguíneas,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s3 : read_matrix (file_search ("biomed.data"))$
(%i4) subsample (s3, '(%c[1] = B and %c[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 ]
(%o4)              [                       ]
                   [ 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 ]
```

Essa é a média de várias variáveis de `s3`,

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

```
(%i2) load ("numericalio")$
(%i3) s3 : read_matrix (file_search ("biomed.data"))$
(%i4) mean (s3);
       65 B + 35 A  317          6 NA + 8145.0
(%o4) [-----------, ---, 87.178, -------------, 18.123,
            100      10                100
                                                  3 NA + 19587
                                                  ------------]
                                                      100
```

Aqui, a primeira componente é sem sentido, uma vez que `A` e `B` são categorias, o segundo componente é a idade média dos indivíduos na forma racional, e o quarto eo último valores exibem um comportamento estranho. Isso ocorre porque o símbolo `NA` é usado aqui para indicar dado não disponível (*non available* em inglês), e as duas médias são certamente sem sentido. Uma solução possível pode ser jogar fora a matriz cujas linhas possuam símbolos `NA`, embora isso cause alguma perda de informação,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s3 : read_matrix (file_search ("biomed.data"))$
(%i4) mean (subsample (s3, '(%c[4] # NA and %c[6] # NA), 3, 4, 5, 6));
(%o4) [79.4923076923077, 86.2032967032967, 16.93186813186813,
                                                        2514
                                                        ----]
                                                         13
```

44.3 Definições para estatística descritiva

`mean` (*`lista`*) [Função]
`mean` (*`matriz`*) [Função]

Essa função calcula a média de uma amostra, definida como

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) mean (s1);
                                 471
(%o4)                            ---
                                 100
(%i5) %, numer;
(%o5)                           4.71
(%i6) s2 : read_matrix (file_search ("wind.data"))$
(%i7) mean (s2);
(%o7)     [9.9485, 10.1607, 10.8685, 15.7166, 14.8441]
```

`var` (*list*) [Função]
`var` (*matrix*) [Função]

This is the sample variance, defined as

$$\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) var (s1), numer;
(%o4)                   8.425899999999999
```

See also function `var1`.

`var1` (*lista*) [Função]
`var1` (*matriz*) [Função]

Essa função calcula a variância da amostra, definida como

$$\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) var1 (s1), numer;
(%o4)                    8.5110101010101
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) var1 (s2);
(%o6) [17.39586540404041, 15.13912778787879, 15.63204924242424,
                            32.50152569696971, 24.66977392929294]
```

See also function `var`.

`std` (*lista*) [Função]
`std` (*matriz*) [Função]

A raíz quadrada da função `var`, a variância com denominador n.

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) std (s1), numer;
(%o4)                   2.902740084816414
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) std (s2);
(%o6) [4.149928523480858, 3.871399812729241, 3.933920277534866,
                            5.672434260526957, 4.941970881136392]
```

Veja também as funções `var` e `std1`.

std1 (*lista*) [Função]
std1 (*matriz*) [Função]

É a raíz quadrada da função `var1`, a variância com denominador $n-1$.

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) std1 (s1), numer;
(%o4)                   2.917363553109228
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) std1 (s2);
(%o6) [4.17083509672109, 3.89090320978032, 3.953738641137555,
                              5.701010936401517, 4.966867617451963]
```

Veja também as funções `var1` e `std`.

noncentral_moment (*lista*, *k*) [Função]
noncentral_moment (*matriz*, *k*) [Função]

O momento não central de ordem k, definido como

$$\frac{1}{n}\sum_{i=1}^{n} x_i^k$$

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) noncentral_moment (s1, 1), numer; /* the mean */
(%o4)                          4.71
(%i6) s2 : read_matrix (file_search ("wind.data"))$
(%i7) noncentral_moment (s2, 5);
(%o7) [319793.8724761506, 320532.1923892463, 391249.5621381556,
                             2502278.205988911, 1691881.797742255]
```

Veja também a função `central_moment`.

central_moment (*lista*, *k*) [Função]
central_moment (*matriz*, *k*) [Função]

O momento central de ordem k, definido como

$$\frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^k$$

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) central_moment (s1, 2), numer; /* a variância */
```

```
(%o4)                    8.425899999999999
(%i6) s2 : read_matrix (file_search ("wind.data"))$
(%i7) central_moment (s2, 3);
(%o7) [11.29584771375004, 16.97988248298583, 5.626661952750102,
                                37.5986572057918, 25.85981904394192]
```

Veja também as funções `central_moment` e `mean`.

`cv` (*lista*) [Função]
`cv` (*matriz*) [Função]

O coeficiente de variação é o quociente entre o desvio padrão da amostra (`std`) e a média `mean`,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) cv (s1), numer;
(%o4)                    .6193977819764815
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) cv (s2);
(%o6) [.4192426091090204, .3829365309260502, 0.363779605385983,
                               .3627381836021478, .3346021393989506]
```

Veja também as funções `std` e `mean`.

`mini` (*lista*) [Função]
`mini` (*matriz*) [Função]

É o valor mínimo da amostra *lista*,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) mini (s1);
(%o4)                             0
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) mini (s2);
(%o6)             [0.58, 0.5, 2.67, 5.25, 5.17]
```

Veja também função `maxi`.

`maxi` (*lista*) [Função]
`maxi` (*matriz*) [Função]

É o valor máximo da amostra *lista*,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) maxi (s1);
(%o4)                             9
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) maxi (s2);
(%o6)          [20.25, 21.46, 20.04, 29.63, 27.63]
```

Veja também a função `mini`.

range (*lista*) [Função]
range (*matriz*) [Função]

A amplitude é a diferença entre os valores de maximo e de mínimo.

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) range (s1);
(%o4)                          9
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) range (s2);
(%o6)          [19.67, 20.96, 17.37, 24.38, 22.46]
```

quantile (*lista*, *p*) [Função]
quantile (*matriz*, *p*) [Função]

É o *p*-quantile (quantil de ordem p), com p sendo um número em $[0, 1]$ (intervalo fechado), da amostra *lista*. Embora exista muitas Definições para quantil de uma amostra (Hyndman, R. J., Fan, Y. (1996) *Sample quantiles in statistical packages*. American Statistician, 50, 361-365), aquela que se baseia em interpolação linear é a que foi implementada no pacote `descriptive`.

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) /* 1st and 3rd quartiles */ [quantile (s1, 1/4), quantile (s1, 3/4)], numer
(%o4)                       [2.0, 7.25]
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) quantile (s2, 1/4);
(%o6)    [7.2575, 7.477500000000001, 7.82, 11.28, 11.48]
```

median (*lista*) [Função]
median (*matriz*) [Função]

Uma vez que a amostra está ordenada, se o tamanho da amostra for ímpar a mediana é o valor central, de outra forma a mediana será a média dos dois valores centrais.

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) median (s1);
                                9
(%o4)                           -
                                2
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) median (s2);
(%o6)         [10.06, 9.855, 10.73, 15.48, 14.105]
```

A mediana é o 1/2-quantil.

Veja também function `quantile`.

`qrange` (*`lista`*) [Função]
`qrange` (*`matriz`*) [Função]

A amplitude do interquartil é a diferença entre o terceiro e o primeiro quartil, `quantile(`*`lista`*`,3/4) - quantile(`*`lista`*`,1/4)`,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) qrange (s1);
                                21
(%o4)                           --
                                4
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) qrange (s2);
(%o6) [5.385, 5.572499999999998, 6.0225, 8.729999999999999,
                                                  6.650000000000002]
```

Veja também a função `quantile`.

`mean_deviation` (*`lista`*) [Função]
`mean_deviation` (*`matriz`*) [Função]

O desvio médio, definido como

$$\frac{1}{n}\sum_{i=1}^{n}|x_i - \bar{x}|$$

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) mean_deviation (s1);
                                51
(%o4)                           --
                                20
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) mean_deviation (s2);
(%o6) [3.287959999999999, 3.075342, 3.23907, 4.715664000000001,
                                                  4.028546000000002]
```

Veja também a função `mean`.

`median_deviation` (*`lista`*) [Função]
`median_deviation` (*`matriz`*) [Função]

O desvio da mediana, definido como

$$\frac{1}{n}\sum_{i=1}^{n}|x_i - med|$$

onde `med` é a mediana da *lista*.

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) median_deviation (s1);
                                5
(%o4)                           -
                                2
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) median_deviation (s2);
(%o6)          [2.75, 2.755, 3.08, 4.315, 3.31]
```

Veja também a função `mean`.

`harmonic_mean` (*`lista`*) [Função]
`harmonic_mean` (*`matriz`*) [Função]

A média harmônica, definida como

$$\frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) y : [5, 7, 2, 5, 9, 5, 6, 4, 9, 2, 4, 2, 5]$
(%i4) harmonic_mean (y), numer;
(%o4)                   3.901858027632205
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) harmonic_mean (s2);
(%o6) [6.948015590052786, 7.391967752360356, 9.055658197151745,
                            13.44199028193692, 13.01439145898509]
```

Veja também as funções `mean` e `geometric_mean`.

`geometric_mean` (*`lista`*) [Função]
`geometric_mean` (*`matriz`*) [Função]

A média geométrica, definida como

$$\left(\prod_{i=1}^{n} x_i \right)^{\frac{1}{n}}$$

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) y : [5, 7, 2, 5, 9, 5, 6, 4, 9, 2, 4, 2, 5]$
(%i4) geometric_mean (y), numer;
(%o4)                   4.454845412337012
(%i5) s2 : read_matrix (file_search ("wind.data"))$
```

```
(%i6) geometric_mean (s2);
(%o6) [8.82476274347979, 9.22652604739361, 10.0442675714889,
                              14.61274126349021, 13.96184163444275]
```

Veja também as funções `mean` e `harmonic_mean`.

`kurtosis` (*lista*) [Função]
`kurtosis` (*matriz*) [Função]

O coeficiente de curtose, definido como

$$\frac{1}{ns^4}\sum_{i=1}^{n}(x_i - \bar{x})^4 - 3$$

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) kurtosis (s1), numer;
(%o4)                   - 1.273247946514421
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) kurtosis (s2);
(%o6) [- .2715445622195385, 0.119998784429451,
   - .4275233490482866, - .6405361979019522, - .4952382132352935]
```

Veja também as funções `mean`, `var` e `skewness`.

`skewness` (*lista*) [Função]
`skewness` (*matriz*) [Função]

O coeficiente de assimetria, definido como

$$\frac{1}{ns^3}\sum_{i=1}^{n}(x_i - \bar{x})^3$$

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) skewness (s1), numer;
(%o4)                   .009196180476450306
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) skewness (s2);
(%o6) [.1580509020000979, .2926379232061854, .09242174416107717,
                              .2059984348148687, .2142520248890832]
```

Veja também as funções `mean`, `var` e `kurtosis`.

`pearson_skewness` (*lista*) [Função]
`pearson_skewness` (*matriz*) [Função]

O coeficiente de assimetria de pearson, definido como

$$\frac{3\ (\bar{x} - med)}{s}$$

onde *med* é a mediana de *lista*.

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) pearson_skewness (s1), numer;
(%o4)                    .2159484029093895
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) pearson_skewness (s2);
(%o6) [- .08019976629211892, .2357036272952649,
          .1050904062491204, .1245042340592368, .4464181795804519]
```

Veja também as funções `mean`, `var` e `median`.

`quartile_skewness` (*lista*) [Função]
`quartile_skewness` (*matriz*) [Função]

O coeficiented de assimetria do quartil, definido como

$$\frac{c_{\frac{3}{4}} - 2\,c_{\frac{1}{2}} + c_{\frac{1}{4}}}{c_{\frac{3}{4}} - c_{\frac{1}{4}}}$$

onde c_p é o quartil de ordem p da amostra *lista*.

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) quartile_skewness (s1), numer;
(%o4)                   .04761904761904762
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) quartile_skewness (s2);
(%o6) [- 0.0408542246982353, .1467025572005382,
       0.0336239103362392, .03780068728522298, 0.210526315789474]
```

Veja também a função `quantile`.

44.4 Definições específicas para estatística descritiva de várias variáveis

`cov` (*matriz*) [Função]

A matriz de covariância da amostra de várias variáveis, definida como

$$S = \frac{1}{n}\sum_{j=1}^{n} \left(X_j - \bar{X}\right)\,\left(X_j - \bar{X}\right)'$$

onde X_j é a j-ésima linha da matriz de amostra.

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
```

```
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) fpprintprec : 7$  /* modifique a precisão para obter uma saída melhor */
(%i5) cov (s2);
      [ 17.22191  13.61811  14.37217  19.39624  15.42162 ]
      [                                                  ]
      [ 13.61811  14.98774  13.30448  15.15834  14.9711  ]
      [                                                  ]
(%o5) [ 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 ]
```

Veja também a função `cov1`.

`cov1` (*matriz*) [Função]

A matriz de covariância da amostra de várias variáveis, definida como

$$\frac{1}{n-1}\sum_{j=1}^{n}\left(X_j-\bar{X}\right)\,\left(X_j-\bar{X}\right)'$$

where X_j is the j-th row of the sample matrix.

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) fpprintprec : 7$ /* modifique a precisão para obter uma saída melhor */
(%i5) cov1 (s2);
      [ 17.39587  13.75567  14.51734  19.59216  15.5774  ]
      [                                                  ]
      [ 13.75567  15.13913  13.43887  15.31145  15.12232 ]
      [                                                  ]
(%o5) [ 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 ]
```

Veja também a função `cov`.

`global_variances` (*matriz*) [Função]
`global_variances` (*matriz*, *valor_lógico*) [Função]

A função `global_variances` retorna uma lista de medidas de variância global:

- *variância total*: `trace(S_1)`,
- *variância média*: `trace(S_1)/p`,
- *variância generalizada*: `determinant(S_1)`,
- *desvio padrão generalizado*: `sqrt(determinant(S_1))`,

- *variância efectiva* `determinant(S_1)^(1/p)`, (defined in: Peña, D. (2002) *Análisis de datos multivariantes*; McGraw-Hill, Madrid.)
- *desvio padrão efectivo*: `determinant(S_1)^(1/(2*p))`.

onde p é a dimensão das várias variáveis aleatórias e S_1 a matriz de covariância retornada por `cov1`.

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) global_variances (s2);
(%o4) [105.338342060606, 21.06766841212119, 12874.34690469686,
          113.4651792608502, 6.636590811800794, 2.576158149609762]
```

A função `global_variances` tem um argumento lógico opcional: `global_variances(x,true)` diz ao Maxima que `x` é a matriz de dados, fazendo o mesmo que `global_variances(x)`. Por outro lado, `global_variances(x,false)` significa que `x` não é a matriz de dados, mas a matriz de covariância, evitando a repetição seu cálculo,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) s : cov1 (s2)$
(%i5) global_variances (s, false);
(%o5) [105.338342060606, 21.06766841212119, 12874.34690469686,
          113.4651792608502, 6.636590811800794, 2.576158149609762]
```

Veja também `cov` e `cov1`.

`cor` (*matriz*) [Função]
`cor` (*matriz*, *valor_lógico*) [Função]

A matriz de correlação da maostra de várias variáveis.

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) fpprintprec:7$
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) cor (s2);
      [   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    ]
```

A função `cor` tem um argumento lógico opcional: `cor(x,true)` diz ao Maxima que `x` é a matriz de dados, fazendo o mesmo que `cor(x)`. Por outro lado, `cor(x,false)` significa que `x` não é a matriz de dados, mas a matriz de covariância, evitando a repetição de seu cálculo,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) fpprintprec:7$
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) s : cov1 (s2)$
(%i6) cor (s, false); /* this is faster */
      [   1.0     .8476339  .8803515  .8239624  .7519506 ]
      [                                                   ]
      [ .8476339    1.0     .8735834  .6902622  0.782502 ]
      [                                                   ]
(%o6) [ .8803515  .8735834    1.0     .7764065  .8323358 ]
      [                                                   ]
      [ .8239624  .6902622  .7764065    1.0     .7293848 ]
      [                                                   ]
      [ .7519506  0.782502  .8323358  .7293848    1.0    ]
```

Veja também `cov` e `cov1`.

`list_correlations` (*matriz*) [Função]
`list_correlations` (*matriz*, *valor_lógico*) [Função]

A função `list_correlations` retorna uma lista de medidas de correlação:

- *matriz de precisão*: o inverso da matriz de covariância S_1,

$$S_1^{-1} = \left(s^{ij}\right)_{i,j=1,2,...,p}$$

- *vector de correlação múltipla*: $(R_1^2, R_2^2, ..., R_p^2)$, com

$$R_i^2 = 1 - \frac{1}{s^{ii} s_{ii}}$$

  sendo um indicador do melhor do ajuste do modelo de regressão linear de várias variáveis dobre X_i quando o resto das variáveis são usados como regressores.

- *matriz de correlação parcial*: como elemento (i, j) sendo

$$r_{ij.rest} = -\frac{s^{ij}}{\sqrt{s^{ii} s^{jj}}}$$

Exemplo:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) z : list_correlations (s2)$
(%i5) fpprintprec : 5$ /* for pretty output */
(%i6) z[1];  /* precision matrix */
```

```
        [  .38486   - .13856   - .15626   - .10239    .031179  ]
        [                                                       ]
        [ - .13856   .34107    - .15233    .038447   - .052842 ]
        [                                                       ]
(%o6)   [ - .15626  - .15233    .47296    - .024816  - .10054  ]
        [                                                       ]
        [ - .10239   .038447   - .024816   .10937    - .034033 ]
        [                                                       ]
        [ .031179   - .052842  - .10054   - .034033   .14834   ]
(%i7) z[2];  /* multiple correlation vector */
(%o7)       [.85063, .80634, .86474, .71867, .72675]
(%i8) z[3];  /* partial correlation matrix */
       [  - 1.0     .38244   .36627   .49908   - .13049 ]
       [                                                 ]
       [  .38244    - 1.0    .37927  - .19907   .23492  ]
       [                                                 ]
(%o8)  [  .36627    .37927   - 1.0    .10911    .37956  ]
       [                                                 ]
       [  .49908   - .19907  .10911   - 1.0     .26719  ]
       [                                                 ]
       [ - .13049   .23492   .37956   .26719    - 1.0   ]
```

A função `list_correlations` também tem um argumento lógico opcional: `list_correlations(x,true)` diz ao Maxima que `x` é a matriz de dados, fazendo o mesmo que `list_correlations(x)`. Por outro lado, `list_correlations(x,false)` significa que `x` não é a matriz de correlação, mas a matriz de covariancia, evitando a repetição de seu cálculo.

Veja também `cov` e `cov1`.

44.5 Definições para gráficos estatísticos

`dataplot` (*lista*) [Função]
`dataplot` (*lista*, *opção_1*, *opção_2*, ...) [Função]
`dataplot` (*matriz*) [Função]
`dataplot` (*matriz*, *opção_1*, *opção_2*, ...) [Função]

A função `dataplot` permite visualização directa de dados de amostra, ambas d uma única variável (*lista*) e de várias variáveis (*matriz*). Fornecendo valores para as seguintes *opções* que são alguns aspéctos de impressão que podem ser controlados:

- `'outputdev`, o valor padrão é `"x"`, indica o formato de dispositivo/ficheiro da figura de saída; valores correctos são `"x"`, `"eps"` e `"png"`, para a tela, formato de ficheiro postscript e formato de ficheiro png, respectivamente.
- `'maintitle`, o valor padrão é `""`, é o título principal entre aspas duplas.
- `'axisnames`, o valor padrão é `["x","y","z"]`, é uma lista de nomes dos eixos `x`, `y` e `z`.
- `'joined`, o valor padrão é `false`, um valor lógico para seleccionar pontos em 2D para serem unidos ou isolados.

- `'picturescales`, o valor padrão é `[1.0, 1.0]`, factor de proporcionalidade para o tamanho do gráfico.
- `'threedim`, o valor padrão é `true`, diz ao Maxima se ou monta-se o gráfico de uma matriz de três colunas como um diagrama 3D ou se monta-se o gráfico como um diagrama de dispersão de várias variáveis. Veja exemplos abaixo.
- `'axisrot`, o valor padrão é `[60, 30]`, modifica o ponto de visualização quando `'threedim` for escolhido para `true` dados forem armazenados em uma matriz de três colunas. O primeiro número é o ângulo de rotação do eixo x, e o segundo número é o angulo de rotação do eixo z-axis, ambas as medidas em graus.
- `'nclasses`, o valor padrão é `10`, é o número de classes para histogramas na diagonal de gráficos de dispersão de várias variáveis.
- `'pointstyle`, o valor padrão é `1`, é um inteiro que indica como mostrar pontos de amostra.

Por exemplo, com a seguite entrada um gráfico simples dos primeiros vinte dígitos de `%pi` é requisitado e a saída é armazenada em um ficheiro no formato eps.

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) dataplot (makelist (s1[k], k, 1, 20), 'pointstyle = 3)$
```

Note que dados unidimensionais são colocados no gráfico como uma série de tempo. No caso seguinte, ocorre a mesma coisa só que com mais dados e com mais configurações,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) dataplot (makelist (s1[k], k, 1, 50), 'maintitle = "Primeiros dígitos de pi
 'axisnames = ["ordem do dígito", "valor do dígito"], 'pointstyle = 2,
 'joined = true)$
```

A função `dataplot` pode ser usada para montar gráficos de pontos no plano. O exemplo seguinte é gráfico de dispersão de pares de pontos de velocidades de vento para o primeira e para o quinta estação meteorológica,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) dataplot (submatrix (s2, 2, 3, 4), 'pointstyle = 2,
 'maintitle = "Pares de medidas de velocidade do vento em nós",
 'axisnames = ["Velocidade do vento em A", "Velocidade do vento em E"])$
```

Se pontos forem armazenados em uma matriz de duas colunas, `dataplot` pode montar o gráfico desses pontos directamente, mas se eles forem formatados em uma lista de pares, essa lista deve ser transformada em uma matriz como no seguinte exemplo.

```
(%i1) load ("descriptive")$
(%i2) x : [[-1, 2], [5, 7], [5, -3], [-6, -9], [-4, 6]]$
(%i3) dataplot (apply ('matrix, x), 'maintitle = "Pontos",
 'joined = true, 'axisnames = ["", ""], 'picturescales = [0.5, 1.0])$
```

Pontos no espaço tridimensional podem ser vistos como uma projeção no plano. Nesse exemplo, o gráfico de velocidades do vento correspondendo a três estações meteorológicas são requisitados, primeiramente em um gráfico em 3D e a seguir em um gráfico de dispersào de várias variáveis.

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) /* 3D plot */ dataplot (submatrix (s2, 4, 5), 'pointstyle = 2,
 'maintitle = "Pares de medidas de velocidades do vento em nós",
 'axisnames = ["Estação A", "Estação B", "Estação C"])$
(%i5) /* Gráfico de dispersão de várias variáveis */ dataplot (submatrix (s2, 4,
 'nclasses = 6, 'threedim = false)$
```

Note que no último exemplo, o número de classes no histogramas da diagonal é escolhido para 6, e aquela opção `'threedim` for escolhida para `false`.

Para mais que três dimensões somente gráficos de dispersão de várias variáveis são possível, como em

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) dataplot (s2)$
```

`histogram` (*lista*) [Função]
`histogram` (*lista*, *opção_1*, *opção_2*, ...) [Função]
`histogram` (*one_column_matrix*) [Função]
`histogram` (*one_column_matrix*, *opção_1*, *opção_2*, ...) [Função]

Essa função monta um gráfico de um histograma. Dados de amostras devem ser armazenados em uma lista de números ou em uma matriz de uma coluna. Fornecendo valores para as seguintes *opções* alguns aspéctos do gráfico podem ser controlados:

- `'outputdev`, o valor padrão é `"x"`, indica o formato de ficheiro da figura de saída; valores correctos são `"x"`, `"eps"` e `"png"`, para a tela, formato de ficheiro postscript e formato de ficheiro png, respectivamente.
- `'maintitle`, o valor padrão é `""`, é o título principal entre aspas duplas.
- `'axisnames`, o valor padrão é `["x", "Fr."]`, é uma lista de nomes dos eixos `x` e `y`.
- `'picturescales`, o valor padrão é `[1.0, 1.0]`, factor de proporcionalidade para o tamanho do gráfico.
- `'nclasses`, o valor padrão é `10`, é o número de classes ou o número de barras.
- `'relbarwidth`, o valor padrão é `0.9`, um número decimao entre 0 e 1 para controlar a largura das barras.
- `'barcolor`, o valor padrão é `1`, um inteiro para indicar a cor das barras.
- `'colorintensity`, o valor padrão é `1`, um número decimal entre 0 e 1 para estabelecer a intensidade da cor.

Nos próximos dois exemplos, histogramas são requisitados para os primeiros 100 dígitos do número `%pi` e para velocidades do vento na terceira estação meteorológica.

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

```
(%i2) load ("numericalio")$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) histogram (s1, 'maintitle = "dígitos de pi", 'axisnames = ["", "Frequência
 'relbarwidth = 0.2, 'barcolor = 3, 'colorintensity = 0.6)$
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) histogram (col (s2, 3), 'colorintensity = 0.3)$
```

Note tque no primeiro caso, `s1` é uma lista e o segundo exemplo, `col(s2,3)` é uma matriz.

Veja também a função `barsplot`.

`barsplot` (*`lista`*) [Função]
`barsplot` (*`lista`*, *`opção_1`*, *`opção_2`*, ...) [Função]
`barsplot` (*`one_column_matrix`*) [Função]
`barsplot` (*`one_column_matrix`*, *`opção_1`*, *`opção_2`*, ...) [Função]

Similar a `histogram` mas para variáveis estatísticas, numéricas ou divididas em categorias. As opções estão abaixo,

- `'outputdev`, o valor padrão é `"x"`, indica o formato de ficheiro da figura de saída; valores correctos são `"x"`, `"eps"` e `"png"`, para a tela, formato de ficheiro postscript e formato de ficheiro png, respectivamente.
- `'maintitle`, o valor padrão é `""`, é o título principal entre aspas duplas.
- `'axisnames`, o valor padrão é `["x", "Fr."]`, é uma lista de nomes dos eixos `x` e `y`.
- `'picturescales`, o valor padrão é `[1.0, 1.0]`, factor de proporcionalidade para o tamanho do gráfico.
- `'relbarwidth`, o valor padrão é `0.9`, um número decimao entre 0 e 1 para controlar a largura das barras.
- `'barcolor`, o valor padrão é `1`, um inteiro para indicar a cor das barras.
- `'colorintensity`, o valor padrão é `1`, um número decimal entre 0 e 1 para estabelecer a intensidade da cor.

Esse exemplo monta um gráfico de barras para os grupos `A` e `B` de pacientes na amostra `s3`,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s3 : read_matrix (file_search ("biomed.data"))$
(%i4) barsplot (col (s3, 1), 'maintitle = "Grupos de pacientes",
 'axisnames = ["Grupo", "# de indivíduos"], 'colorintensity = 0.2)$
```

A primeira coluna na amostra `s3` armazena os valores das categorias `A` e `B`, também conhecidos algumas vezes como factores. Por outro lado, os números inteiros positivos na segunda coluna sào idades, em anos, que se comportam como variável discreta, então podemos montar um gráfico as freqyuências absolutas para esses valores,

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s3 : read_matrix (file_search ("biomed.data"))$
(%i4) barsplot (col (s3, 2), 'maintitle = "Idades",
```

```
'axisnames = ["Anos", "# dos indivíduos"], 'colorintensity = 0.2,
'relbarwidth = 0.6)$
```

Veja também a função `histogram`.

`boxplot` (*data*) [Função]
`boxplot` (*data*, *opção_1*, *opção_2*, ...) [Função]

Essa função monta diagramas em caixas. O argumento *data* pode ser uma lista, que não é de grande interesse, uma vez que esses diagramas são principalmente usados para comparação entre diferentes amostras, ou uma matriz, eentão é possível comparar dois ou mais componentes de uma variável estatística de várias variáveis. Mas é também permitido *data* se uma lista de amostras com tamanhos diferentes de amostra, de facto essa é aa única função no pacote `descriptive` que admite esse tipo de estrutura de dados. Veja o exemplo abaixo. Abaixo etão as opções,

- `'outputdev`, o valor padrão é `"x"`, indica o formato de ficheiro da figura de saída; valores correctos são `"x"`, `"eps"` e `"png"`, para a tela, formato de ficheiro postscript e formato de ficheiro png, respectivamente.
- `'maintitle`, o valor padrão é `""`, é o título principal entre aspas duplas.
- `'axisnames`, o valor padrão é `["sample", "y"]`, é uma lista de nomes dos eixos `x` e `y`.
- `'picturescales`, o valor padrão é `[1.0, 1.0]`, factor de proporcionalidade para o tamanho do gráfico.

Examples:

```
(%i1) load ("descriptive")$
(%i2) load ("numericalio")$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) boxplot (s2, 'maintitle = "Velocidade do vento em nós",
 'axisnames = ["Estação do ano", ""])$
(%i5) 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]]$
(%i6) boxplot (A)$
```

45 diag

45.1 Definições para diag

`diag` (*lm*) [Função]

Constrói a matriz quadrada com as matrizes de *lm* na diagonal. *lm* é uma lista de matrizes ou escalares.

Exemplo:

```
(%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 ]
```

Para usar essa função escreva primeiramente `load("diag")`.

`JF` (*lambda*,*n*) [Função]

Retorna a célula de Jordan de ordem *n* com autovalor *lambda*.

Exemplo:

```
(%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 ]
```

Para usar essa função escreva primeiramente `load("diag")`.

`jordan` (*mat*) [Função]

Retorna a forma de Jordan da matriz *mat*, mas codificada em uma lista do Maxima. Para pegar a matriz correspondente à codificação, chame a função `dispJordan` sando como argumento a saída de `JF`.

Exemplo:

```
(%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 ]
```

Para usar essa função escreva primeiramente `load("diag")`. Veja também `dispJordan` e `minimalPoly`.

`dispJordan` (*l*) [Função]

Retorna a matriz de Jordan associada à codificação fornecida pela lista do Maxima *l*, que é a saída fornecida pela função `jordan`.

Exemplo:

```
(%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 ]
```

Para usar essa função escreva primeiramente `load("diag")`. Veja também `jordan` e `minimalPoly`.

`minimalPoly` (*l*) [Função]

Retorna o menor polinómio associado à codificação fornecida pela lista do Maxima *l*, que é a saída fornecida pela função `jordan`.

Exemplo:

```
(%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)
```

Para usar essa função escreva prmeiramente `load("diag")`. Veja também `jordan` e `dispJordan`.

`ModeMatrix` (*A*,*l*) [Função]

Retorna a matriz M tal que $(Mm1).A.M = J$, onde J é a forma de Jordan de A. A lista do Maxima *l* é a codificação da forma de Jordan como retornado pela função `jordan`.

Exemplo:

```
(%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 que `dispJordan(%o3)` é a forma de Jordan da matriz `a`.

Para usa essa função escreva primeiramente `load("diag")`. Veja também `jordan` e `dispJordan`.

`mat_function` (*f*,*mat*) [Função]

Retorna $f(mat)$, onde f é uma função analítica e *mat* uma matriz. Essa computação é baseada na fórmula da integral de Cauchy, que estabelece que se `f(x)` for analítica e

```
mat=diag([JF(m1,n1),...,JF(mk,nk)]),
```

então

```
f(mat)=ModeMatrix*diag([f(JF(m1,n1)),...,f(JF(mk,nk))])*ModeMatrix^^(-1)
```

Note que existem entre 6 ou 8 outros métodos para esse cálculo.

Segue-se alguns exemplos.

Exemplo 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           ]
```

Exemplo 2:

```
(%i5) b1:matrix([0,0,1,1,1],
                [0,0,0,1,1],
                [0,0,0,0,1],
                [0,0,0,0,0],
                [0,0,0,0,0])$

(%i6) mat_function(exp,t*b1);
                  [              2     ]
                  [             t      ]
                  [ 1  0  t  t  -- + t ]
                  [             2      ]
                  [                    ]
(%o6)             [ 0  1  0  t    t    ]
                  [                    ]
                  [ 0  0  1  0    t    ]
                  [                    ]
                  [ 0  0  0  1    0    ]
                  [                    ]
                  [ 0  0  0  0    1    ]
(%i7) minimalPoly(jordan(b1));
                             3
(%o7)                       x
(%i8) ident(5)+t*b1+1/2*(t^2)*b1^^2;
                  [              2     ]
                  [             t      ]
```

```
                    [ 1  0  t  t  -- + t ]
                    [             2      ]
                    [                    ]
(%o8)               [ 0  1  0  t    t    ]
                    [                    ]
                    [ 0  0  1  0    t    ]
                    [                    ]
                    [ 0  0  0  1    0    ]
                    [                    ]
                    [ 0  0  0  0    1    ]
(%i9) mat_function(exp,%i*t*b1);
                    [                          2 ]
                    [                         t  ]
                    [ 1  0  %i t  %i t  %i t - -- ]
                    [                         2  ]
                    [                            ]
(%o9)               [ 0  1   0    %i t    %i t    ]
                    [                            ]
                    [ 0  0   1     0      %i t    ]
                    [                            ]
                    [ 0  0   0     1        0     ]
                    [                            ]
                    [ 0  0   0     0        1     ]
(%i10) mat_function(cos,t*b1)+%i*mat_function(sin,t*b1);
                     [                          2 ]
                     [                         t  ]
                     [ 1  0  %i t  %i t  %i t - -- ]
                     [                         2  ]
                     [                            ]
(%o10)               [ 0  1   0    %i t    %i t    ]
                     [                            ]
                     [ 0  0   1     0      %i t    ]
                     [                            ]
                     [ 0  0   0     1        0     ]
                     [                            ]
                     [ 0  0   0     0        1     ]
```

Exemplo 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  ]
```

Para usar essa função escreva primeiramente `load("diag")`.

46 distrib

46.1 Introdução a distrib

Pacote `distrib` contém um conjunto de funções para fazer cálculos envolvendo probabilidades de modelos de uma única variável estatística e de ambos os tipos discreta e contínua.

O que segue é um curto resumo de definiçoes básicas relacionadas à teoria das probabilidades.

Seja $f(x)$ a *função densidade de probabilidade* absoluta de uma variável aleatória contínua X. A *função distribuição de probabilidade* é definida como

$$F\left(x\right) = \int_{-\infty}^{x} f\left(u\right)\ du$$

que é igual à probabilidade *Pr(X <= x)*.

O valor *médio* é um parâmetro de localização e está definido como

$$E\left[X\right] = \int_{-\infty}^{\infty} x\, f\left(x\right)\ dx$$

A *variância* é uma medida de variação,

$$V\left[X\right] = \int_{-\infty}^{\infty} f\left(x\right)\ \left(x - E\left[X\right]\right)^2\ dx$$

que é um número real positivo. A raíz quadrada da variância é o *desvio padrão*, $D[X] = sqrt(V[X])$, e esse *desvio padrão* é outra medida de variação.

O *coeficiente de assimetria* é uma medida de não simetria,

$$SK\left[X\right] = \frac{\int_{-\infty}^{\infty} f\left(x\right)\ \left(x - E\left[X\right]\right)^3\ dx}{D\left[X\right]^3}$$

E o *coeficiente de curtose* mede o grau de achatamento de uma distribuição,

$$KU\left[X\right] = \frac{\int_{-\infty}^{\infty} f\left(x\right)\ \left(x - E\left[X\right]\right)^4\ dx}{D\left[X\right]^4} - 3$$

Se X for gaussiana, $KU[X] = 0$. De facto, ambos assimetria e curtose são parâmetros de ajuste usados para medir a não gaussianidade de uma distribuição.

Se a variável aleatória X for discreta, a função densidade de probabilidade, ou simplesmente *probabilidade*, $f(x)$ toma valores positivos dentro de certos conjuntos contáveis de números x_i, e zero em caso contrário. Nesse caso, a função distribuição de probabilidade é

$$F\left(x\right) = \sum_{x_i \leq x} f\left(x_i\right)$$

A média, variância, desvio padrão, coeficiente de assimetria e coeficiente de curtose tomam a forma

$$E\left[X\right] = \sum_{x_i} x_i f\left(x_i\right),$$

$$V\left[X\right] = \sum_{x_i} f\left(x_i\right)\left(x_i - E\left[X\right]\right)^2,$$

$$D\left[X\right] = \sqrt{V\left[X\right]},$$

$$SK\left[X\right] = \frac{\sum_{x_i} f\left(x\right)\ \left(x - E\left[X\right]\right)^3\ dx}{D\left[X\right]^3}$$

and

$$KU\left[X\right] = \frac{\sum_{x_i} f\left(x\right)\ \left(x - E\left[X\right]\right)^4\ dx}{D\left[X\right]^4} - 3,$$

respectively.

O Pacote `distrib` inclui funções para simulação de variáveis estatísticas pseudo-aleatórias. Algumas dessas funções fazem uso de variáveis opcionais que indicam o algoritmo a ser usado. O método inverso genérico (baseado no facto que se u for um número aleatório uniforme no intervalo $(0, 1)$, então *F^(-1)(u)* é uma variável estatística pseudo-aleatória com distribuição F) está implementada para a maioria dos casos; isso é um método subóptimo em termos de cronometragem, mas útil para fazer comparações com outros algoritmos. Nesse exemplo, a `perandom_formance` dos algoritmos `ahrens_cheng` e `inverse` em simular variáveis chi-quadradas (letra grega "chi") são comparadas por meio de seus histogramas:

```
(%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.
```

Com o objectivo de fazer comparações visuais entre algoritmos para uma variável estatística discreta, a função `barsplot` do pacote `descriptive` pode ser usada.

Note que algum trabalho resta para ser realizado, uma vez que essas funções de simulação não foram ainda verificadas pelos mais rigorosamente melhores dos testes de ajuste.

Por favor, consulte um manual introdutório sobre probabilidade e estatística para maiores informações sobre todo esse material matemático.

Existe uma convenção de nome no pacote `distrib`. Todo nome de função tem duas partes, a primeira faz referência à função ou ao parâmetro que queremos calcular,

```
Funções:
   função densidade de probabilidade            (pdf_*)
```

```
função distribuição de probabilidade       (cdf_*)
Quartil                     (quantile_*)
Média                         (mean_*)
Variância                     (var_*)
Desvio padrão          (std_*)
Coeficiente de assimetria        (skewness_*)
Coeficiente de curtose         (kurtosis_*)
Variável estatística pseudo-aleatória              (random_*)
```

A segunda parte é uma referência explícita ao modelo probabilístico,

```
Distribuíções contínuas:
   Normal              (*normal)
   Student             (*student_t)
   Chi^2               (*chi2)
   F                   (*f)
   Exponencial         (*exp)
   Lognormal           (*lognormal)
   Gama               (*gamma)
   Beta                (*beta)
   contínua uniforme  (*continuous_uniform)
   Logística            (*logistic)
   Pareto              (*pareto)
   Weibull             (*weibull)
   Rayleigh            (*rayleigh)
   Laplace             (*laplace)
   Cauchy              (*cauchy)
   Gumbel              (*gumbel)

Distribuições discretas:
   Binomial             (*binomial)
   Poisson              (*poisson)
   Bernoulli            (*bernoulli)
   Geométrica            (*geometric)
   discreta uniforme     (*discrete_uniform)
   hipergeométrica       (*hypergeometric)
   Binomial Negativa    (*negative_binomial)
```

Por exemplo, `pdf_student_t(x,n)` é a função densidade de probabilidade da distribuição de Student com n graus de liberdade, `std_pareto(a,b)` é o desvio padrão da distribuição de Pareto com parâmetros a e b e `kurtosis_poisson(m)` é o coeficiente de curtose da distribuição de Poisson com média m.

Para poder usar o pacote `distrib` precisa primeiro carregá-lo escrevendo

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

Para comentários, melhorias ou sugestões, por favor contacte o autor em *'mario AT edu DOT xunta DOT es'*.

46.2 Definições para distribuições contínuas

`pdf_normal (x,m,s)` [Função]
Retorna o valor em x da função densidade de probabilidade de uma variável aleatória $Normal(m, s)$, com $s > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_normal (x,m,s)` [Função]
Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $Normal(m, s)$, com $s > 0$. Essa função é definida em termos de funções de erro internas do Maxima, `erf`.

```
(%i1) load ("distrib")$
(%i2) assume(s>0)$ cdf_normal(x,m,s);
                                x - m
                           erf(---------)
                               sqrt(2) s    1
(%o3)                      -------------- + -
                                 2          2
```

Veja também `erf`.

`quantile_normal (q,m,s)` [Função]
Retorna o q-quantil de uma variável aleatória $Normal(m, s)$, com $s > 0$; em outras palavras, isso é o inverso de `cdf_normal`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_normal (m,s)` [Função]
Retorna a média de uma variável aleatória $Normal(m, s)$, com $s > 0$, a saber m. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_normal (m,s)` [Função]
Retorna a variância de uma variável aleatória $Normal(m, s)$, com $s > 0$, a saber $s\hat{}2$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_normal (m,s)` [Função]
Retorna o desvio padrão de uma variável aleatória $Normal(m, s)$, com $s > 0$, a saber s. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_normal (m,s)` [Função]
Retorna o coeficiente de assimetria de uma variável aleatória $Normal(m, s)$, com $s > 0$, que é sempre igual a 0. Para fazer uso dessa função,escreva primeiramente `load("distrib")`.

`kurtosis_normal (m,s)` [Função]
Retorna o coeficiente de curtose de uma variável aleatória $Normal(m, s)$, com $s > 0$, que é sempre igual a 0. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_normal_algorithm` [Variável de opção]
Valor por omissão: `box_mueller`

Esse é o algoritmo seleccionado para simular variáveis aleatórias normais. O algoritmos implementados são `box_mueller` e `inverse`:

- `box_mueller`, Baseado no algoritmo descrito em Knuth, D.E. (1981) *Seminumerical Algorithms. The Art of Computer Programming.* Addison-Wesley.
- `inverse`, baseado no método inverso genérico.

Veja também `random_normal`.

`random_normal (m,s)` [Função]
`random_normal (m,s,n)` [Função]

Retorna uma variável estatística pseudo-aleatória $Normal(m, s)$, com $s > 0$. Chamando `random_normal` com um terceiro argumento n, uma amostra aleatória de tamanho n será simulada.

Existem dois algoritmos implementados para essa função, e o algoritmo a ser usado pode ser seleccionado fornecendo um certo valor para a variável global `random_normal_algorithm`, cujo valor padrão é `box_mueller`.

Veja também `random_normal_algorithm`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_student_t (x,n)` [Função]

Retorna o valor em x da função densidade de probabilidade de uma variável aleatória de Student $t(n)$, com $n > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_student_t (x,n)` [Função]

Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória de Student $t(n)$, com $n > 0$. Essa função não tem uma forma definitiva e é calculada numericamente se a variável global `numer` for igual a `true`, de outra froma `cdf_student_t` retorna uma expressão nominal.

```
(%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)` [Função]

Retorna o q-quantil de uma variável aleatória de Student $t(n)$, com $n > 0$; em outras palavras, `quantile_student_t` é o inverso de `cdf_student_t`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_student_t (n)` [Função]

Retorna a média de uma variável aleatória de Student $t(n)$, com $n > 0$, que é sempre igual a 0. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_student_t (n)` [Função]

Retorna a variância de uma variável aleatória de Student $t(n)$, com $n > 2$.

```
(%i1) load ("distrib")$
(%i2) assume(n>2)$  var_student_t(n);
                                  n
(%o3)                            -----
                                 n - 2
```

`std_student_t (n)` [Função]

Retorna o desvio padrão de uma variável aleatória de Student $t(n)$, com $n > 2$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_student_t (n)` [Função]

Retorna o coeficiente de assimetria de uma variável aleatória de Student $t(n)$, com $n > 3$, que é sempre igual a 0. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_student_t (n)` [Função]

Retorna o coeficiente de curtose de uma variável aleatória de Student $t(n)$, com $n > 4$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_student_t_algorithm` [Variável de opção]

Valor por omissão: `ratio`

Esse é o algoritmo seleccionado para simular variáveis estatísticas pseudo-aleatórias de Student. Algorítmos implementados são `inverse` e `ratio`:

- `inverse`, baseado no método inverso genérico.
- `ratio`, baseado no facto que se Z for uma variável aleatória normal $N(0,1)$ e S^2 for uma variável aleatória chi quadrada com n graus de liberdade, $Chi^2(n)$, então

$$X = \frac{Z}{\sqrt{\frac{S^2}{n}}}$$

é uma variável aleatória de Student com n graus de liberdade, $t(n)$.

Veja também `random_student_t`.

`random_student_t (n)` [Função]
`random_student_t (n,m)` [Função]

Retorna uma variável estatística pseudo-aleatória de Student $t(n)$, com $n > 0$. Chamando `random_student_t` com um segundo argumento m, uma amostra aleatória de tamanho m será simulada.

Existem dois algoritmos implementados para essa função, se pode seleccionar o algoritmo a ser usado fornecendo um certo valor à variável global `random_student_t_algorithm`, cujo valor padrão é `ratio`.

Veja também `random_student_t_algorithm`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_chi2 (x,n)` [Função]

Retorna o valor em x da função densidade de probabilidade de uma variável aleatória Chi-quadrada $Chi^2(n)$, com $n > 0$.

A variável aleatória $Chi^2(n)$ é equivalente a $Gamma(n/2, 2)$, portanto quando Maxima não tiver informação para pegar o resultado, uma forma nomial baseada na função de densidade densidade de probabilidade da função gama é retornada.

```
(%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)` [Função]

Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória Chi-quadrada $Chi^2(n)$, com $n > 0$.

Essa função não possui uma forma fechada e é calculada numericamante se a variável global `numer` for igual a `true`, de outra forma essa função retorna uma expressão nominal baseada na distribuição gama, uma vez que a variável aleatória $Chi^2(n)$ é equivalente a é equivalente a $Gamma(n/2, 2)$.

```
(%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)` [Função]

Retorna o q-quantilede uma variável aleatória Chi-quadrada $Chi^2(n)$, com $n > 0$; em outras palavras, essa função é a inversa da função `cdf_chi2`. O argumento q deve ser um elemento de $[0, 1]$.

This função não possui uma forma fechada e é calculada numericamante se a variável global `numer` for igual a `true`, de outra forma essa função retorna uma expressão nominal baseada no quantil da função gama, uma vez que a variável aleatória $Chi^2(n)$ é equivalente a $Gamma(n/2, 2)$.

```
(%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)` [Função]

Retorna a média de uma variável aleatória Chi-quadrada $Chi^2(n)$, com $n > 0$.

A variável aleatória $Chi^2(n)$ é equivalente a $Gamma(n/2, 2)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada na média da função gama é retornada.

```
(%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)` [Função]

Retorna a variância de uma variável aleatória Chi-quadrada $Chi^2(n)$, com $n > 0$.

A variável aleatória $Chi^2(n)$ é equivalente a $Gamma(n/2, 2)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada na variância da função gama é retornada.

```
(%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)` [Função]

Retorna o desvio padrão de uma variável aleatória Chi-quadrada $Chi^2(n)$, com $n > 0$.

A variável aleatória $Chi^2(n)$ é equivalente a $Gamma(n/2, 2)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada no desvio padrão da função gama é retornada.

```
(%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)` [Função]

Retorna o coeficiente de assimetria de uma variável aleatória Chi-quadrada $Chi^2(n)$, com $n > 0$.

A variável aleatória $Chi^2(n)$ é equivalente a $Gamma(n/2, 2)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada no coeficiente de assimetria da função gama é retornada.

```
(%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)` [Função]

Retorna o coeficiente de curtose de uma variável aleatória Chi-quadrada $Chi^2(n)$, com $n > 0$.

A variável aleatória $Chi^2(n)$ é equivalente a $Gamma(n/2, 2)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada no coeficiente de curtose da função gama é retornada.

```
(%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` [Variável de opção]

Valor por omissão: `ahrens_cheng`

Esse é o algoritmo seleccionado para simular variáveis estatística pseudo-aleatórias Chi-quadradas. Os algoritmos implementados são `ahrens_cheng` e `inverse`:

- `ahrens_cheng`, baseado na simulação aleatória de variáveis gama. Veja `random_gamma_algorithm` para mais detalhes.
- `inverse`, baseado no método inverso genérico.

Veja também `random_chi2`.

`random_chi2 (n)` [Função]
`random_chi2 (n,m)` [Função]

Retorna uma variável estatística pseudo-aleatória Chi-square $Chi^2(n)$, com $n > 0$. Chamando `random_chi2` com um segundo argumento m, uma amostra aleatória de tamanho m será simulada.

Existem dois algoritmos implementados para essa função, se pode seleccionar o algoritmo a ser usado fornecendo um certo valor à variável global `random_chi2_algorithm`, cujo valor padrão é `ahrens_cheng`.

Veja também `random_chi2_algorithm`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

pdf_f (*x*,*m*,*n*) [Função]

Retorna o valor em x da função densidade de probabilidade de uma variável aleatória F, $F(m,n)$, com $m,n > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

cdf_f (*x*,*m*,*n*) [Função]

Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória F, $F(m,n)$, com $m,n > 0$. Essa função não possui uma forma definitiva e é calculada numericamente se a variável global `numer` for igual a `true`, de outra forma retorna uma expressão nominal.

```
(%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*) [Função]

Retorna o q-quantil de uma variável aleatória F, $F(m,n)$, com $m,n > 0$; em outras palavras, essa função é o inverso de `cdf_f`. O argumento q deve ser um elemento de $[0,1]$.

Essa função não possui uma forma fechada e é calculada numericamante se a variável global `numer` for igual a `true`, de outra forma essa função retorna uma expressão nominal.

```
(%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*) [Função]

Retorna a média de uma variável aleatória F, $F(m,n)$, com $m > 0, n > 2$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

var_f (*m*,*n*) [Função]

Retorna a variância de uma variável aleatória F, $F(m,n)$, com $m > 0, n > 4$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

std_f (*m*,*n*) [Função]

Retorna o desvio padrão de uma variável aleatória F, $F(m,n)$, com $m > 0, n > 4$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

skewness_f (*m*,*n*) [Função]

Retorna o coeficiente de assimetria de uma variável aleatória F, $F(m,n)$, com $m > 0, n > 6$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_f (m,n)` [Função]

Retorna o coeficiente de curtose de uma variável aleatória F, $F(m,n)$, com $m > 0, n > 8$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_f_algorithm` [Variável de opção]

Valor por omissão: `inverse`

Esse é o algoritmo seleccionado para simular variáveis estatísticas pseudo-aleatórias F. Os algoritmos implementados são `ratio` e `inverse`:

- `ratio`, baseado no facto de que se X for uma variável aleatória $Chi^2(m)$ e Y for uma variável aleatória $Chi^2(n)$, então

$$F = \frac{nX}{mY}$$

  é uma variável aleatória F com m e n graus de liberdade, $F(m,n)$.

- `inverse`, baseado no método inverso genérico.

Veja também `random_f`.

`random_f (m,n)` [Função]
`random_f (m,n,k)` [Função]

Retorna uma variável estatística pseudo-aleatória F, $F(m,n)$, com $m,n > 0$. Chamando `random_f` com um terceiro argumento k, uma amostra aleatória de tamanho k será simulada.

Existem dois algoritmos implementados para essa função, se pode seleccionar o algoritmo a ser usado fornecendo um certo valor à variável global `random_f_algorithm`, cujo valor padrão é `inverse`.

Veja também `random_f_algorithm`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_exp (x,m)` [Função]

Retorna o valor em x da função densidade de probabilidade variável aleatória $Exponential(m)$, com $m > 0$.

A variável aleatória $Exponential(m)$ é equivalente a $Weibull(1, 1/m)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada na função de densidade de probabilidade de Weibull éretornada.

```
(%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)` [Função]

Retorna o valor em x da função distribuição de probabilidade variável aleatória $Exponential(m)$, com $m > 0$.

A variável aleatória $Exponential(m)$ é equivalente a $Weibull(1, 1/m)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada na distribuição de Weibull é retornada.

```
(%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*) [Função]

Retorna o q-quantil variável aleatória $Exponential(m)$, com $m > 0$; em outras palavras, essa função é inversa da função `cdf_exp`. O argumento q deve ser um elemento de $[0, 1]$.

A variável aleatória $Exponential(m)$ é equivalente a $Weibull(1, 1/m)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada no qualtil de Weibull é retornada.

```
(%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*) [Função]

Retorna a média de uma variável aleatória $Exponential(m)$, com $m > 0$.

A variável aleatória $Exponential(m)$ é equivalente a $Weibull(1, 1/m)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada na média de Weibull é reornada.

```
(%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*) [Função]

Retorna a variância de uma variável aleatória $Exponential(m)$, com $m > 0$.

A variável aleatória $Exponential(m)$ é equivalente a $Weibull(1, 1/m)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada na variância de Weibull é retornada.

```
(%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*) [Função]

Retorna o desvio padrão de uma variável aleatória $Exponential(m)$, com $m > 0$.

A variável aleatória $Exponential(m)$ é equivalente a $Weibull(1, 1/m)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada no desvio padrão de Weibull é retornada.

```
(%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*) [Função]

Retorna o coeficiente de assimetria de uma variável aleatória $Exponential(m)$, com $m > 0$.

A variável aleatória $Exponential(m)$ é equivalente a $Weibull(1, 1/m)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada no coeficiente de assimetria de Weibull é retornada.

```
(%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*) [Função]

Retorna o coeficiente de curtose de uma variável aleatória $Exponential(m)$, com $m > 0$.

A variável aleatória $Exponential(m)$ é equivalente a $Weibull(1, 1/m)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada no coeficiente de curtose de Weibull é retornada.

```
(%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` [Variável de opção]

Valor por omissão: `inverse`

Esse é o algoritmo seleccionado para simular variáveis exponenciais estatística pseudo-aleatórias. Os algoritmos implementados são `inverse`, `ahrens_cheng` e `ahrens_dieter`

- `inverse`, baseado no método inverso genérico.
- `ahrens_cheng`, baseado no facto de que a variável aleatória $Exp(m)$ é equivalente a $Gamma(1, 1/m)$. Veja `random_gamma_algorithm` para maiores detalhes.
- `ahrens_dieter`, baseado no algoritmo descrito em Ahrens, J.H. e Dieter, U. (1972) *Computer methods for sampling from the exponential and normal distributions.* Comm, ACM, 15, Oct., 873-882.

Veja também `random_exp`.

`random_exp (m)` [Função]
`random_exp (m,k)` [Função]

Retorna uma variável estatística pseudo-aleatória $Exponential(m)$, com $m > 0$. Chamando `random_exp` com um segundo argumento k, uma amostra aleatória de tamanho k será simulada.

Existem três algoritmos implementados para essa função, se pode seleccionar o algoritmo a ser usado fornecendo um certo valor à variável global `random_exp_algorithm`, cujo valor padrão é `inverse`.

Veja também `random_exp_algorithm`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_lognormal (x,m,s)` [Função]

Retorna o valor em x da função densidade de probabilidade de uma variável aleatória $Lognormal(m, s)$, com $s > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_lognormal (x,m,s)` [Função]

Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $Lognormal(m, s)$, com $s > 0$. Essa função é definida em termos de funções `erf`de erro internas do Maxima.

```
(%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

Veja também `erf`.

`quantile_lognormal (q,m,s)` [Função]
Retorna o q-quantil de uma variável aleatória $Lognormal(m, s)$, com $s > 0$; em outras palavras, essa função é a inversa da função `cdf_lognormal`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_lognormal (m,s)` [Função]
Retorna a média de uma variável aleatória $Lognormal(m, s)$, com $s > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_lognormal (m,s)` [Função]
Retorna a variância de uma variável aleatória $Lognormal(m, s)$, com $s > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_lognormal (m,s)` [Função]
Retorna o desvio padrão de uma variável aleatória $Lognormal(m, s)$, com $s > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_lognormal (m,s)` [Função]
Retorna o coeficiente de assimetria de uma variável aleatória $Lognormal(m, s)$, com $s > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_lognormal (m,s)` [Função]
Retorna o coeficiente de curtose de uma variável aleatória $Lognormal(m, s)$, com $s > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_lognormal (m,s)` [Função]
`random_lognormal (m,s,n)` [Função]
Retorna uma variável estatística pseudo-aleatória $Lognormal(m, s)$, com $s > 0$. Chamando `random_lognormal` com um terceiro argumento n, uma amostra aleatória de tamanho n será simulada.

Variáveis Log-normal são simuladas por meio de variáveis estatísticas normais pseudo-aleatórias. Existem dois algoritmos implementados para essa função, se pode selecionar o algoritmo a ser usado fornecendo um certo valor à variável global `random_normal_algorithm`, cujo valor padrão é `box_mueller`.

Veja também `random_normal_algorithm`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_gamma (x,a,b)` [Função]
Retorna o valor em x da função densidade de probabilidade de uma variável aleatória $Gamma(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_gamma (x,a,b)` [Função]
Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $Gamma(a, b)$, com $a, b > 0$.

Essa função não possui uma forma fechada e é calculada numericamante se a variável global `numer` for igual a `true`, de outra forma essa função retorna uma expressão nominal.

```
(%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*) [Função]

Retorna o q-quantil de uma variável aleatória $Gamma(a, b)$, com $a, b > 0$; em outras palavras, essa função é a inversa da função `cdf_gamma`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_gamma` (*a*,*b*) [Função]

Retorna a média de uma variável aleatória $Gamma(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_gamma` (*a*,*b*) [Função]

Retorna a variância de uma variável aleatória $Gamma(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_gamma` (*a*,*b*) [Função]

Retorna o desvio padrão de uma variável aleatória $Gamma(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_gamma` (*a*,*b*) [Função]

Retorna o coeficiente de assimetria de uma variável aleatória $Gamma(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_gamma` (*a*,*b*) [Função]

Retorna o coeficiente de curtose de uma variável aleatória $Gamma(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_gamma_algorithm` [Variável de opção]

Valor por omissão: `ahrens_cheng`

Esse é o algoritmo seleccionado para simular variáveis estatística gama pseudo-aleatórias. Os algoritmos implementados são `ahrens_cheng` e `inverse`

- `ahrens_cheng`, essa é uma combinação de dois processos, dependendo do valor do parâmetro *a*:

  For $a >= 1$, Cheng, R.C.H. e Feast, G.M. (1979). *Some simple gamma variate generators*. Appl. Stat., 28, 3, 290-295.

  For $0 < a < 1$, Ahrens, J.H. e Dieter, U. (1974). *Computer methods for sampling from gamma, beta, poisson and binomial cdf_tributions*. Computing, 12, 223-246.

- `inverse`, baseado no método inverso genérico.

Veja também `random_gamma`.

`random_gamma` (*a*,*b*) [Função]
`random_gamma` (*a*,*b*,*n*) [Função]

Retorna uma variável estatística pseudo-aleatória $Gamma(a, b)$, com $a, b > 0$. Chamando `random_gamma` com um terceiro argumento n, uma amostra aleatória de tamanho n será simulada.

Existem dois algoritmos implementados para essa função, se pode seleccionar o algoritmo a ser usado fornecendo um certo valor à variável global `random_gamma_algorithm`, cujo valor padrão é `ahrens_cheng`.

Veja também `random_gamma_algorithm`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_beta` (*x*,*a*,*b*) [Função]

Retorna o valor em x da função densidade de probabilidade de uma variável aleatória $Beta(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_beta` (*x*,*a*,*b*) [Função]

Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $Beta(a, b)$, com $a, b > 0$.

Essa função não possui uma forma fechada e é calculada numericamante se a variável global `numer` for igual a `true`, de outra forma essa função retorna uma expressão nominal.

```
(%i1) load ("distrib")$
(%i2) cdf_beta(1/3,15,2);
                                  1
(%o2)                  cdf_beta(-, 15, 2)
                                  3
(%i3) %,numer;
(%o3)                7.666089131388224E-7
```

`quantile_beta` (*q*,*a*,*b*) [Função]

Retorna o q-quantil de uma variável aleatória $Beta(a, b)$, com $a, b > 0$; em outras palavras, essa função é a inversa da função `cdf_beta`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_beta` (*a*,*b*) [Função]

Retorna a média de uma variável aleatória $Beta(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_beta` (*a*,*b*) [Função]

Retorna a variância de uma variável aleatória $Beta(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_beta` (*a*,*b*) [Função]

Retorna o desvio padrão de uma variável aleatória $Beta(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_beta (a,b)` [Função]
Retorna o coeficiente de assimetria de uma variável aleatória $Beta(a,b)$, com $a,b>0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_beta (a,b)` [Função]
Retorna o coeficiente de curtose de uma variável aleatória $Beta(a,b)$, com $a,b>0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_beta_algorithm` [Variável de opção]
Valor por omissão: `cheng`

Esse é o algoritmo seleccionado para simular variáveis estatísticas beta pseudo-aleatórias. Os algoritmos implementados são `cheng`, `inverse` e `ratio`

- `cheng`, esse é o algoritmo definido em Cheng, R.C.H. (1978). *Generating Beta Variates with Nonintegral Shape Parameters*. Communications of the ACM, 21:317-322
- `inverse`, baseado no método inverso genérico.
- `ratio`, baseado no facto de que se X for uma variável aleatória $Gamma(a,1)$ e Y for $Gamma(b,1)$, então a razão $X/(X+Y)$ está distribuída como $Beta(a,b)$.

Veja também `random_beta`.

`random_beta (a,b)` [Função]
`random_beta (a,b,n)` [Função]
Retorna uma variável estatística pseudo-aleatória $Beta(a,b)$, com $a,b>0$. Chamando `random_beta` com um terceiro argumento n, uma amostra aleatória de tamanho n será simulada.

Existem três algoritmos implementados para essa função, se pode seleccionar o algoritmo a ser usado fornecendo um certo valor à variável global `random_beta_algorithm`, cujo valor padrão é `cheng`.

Veja também `random_beta_algorithm`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_continuous_uniform (x,a,b)` [Função]
Retorna o valor em x da função densidade de probabilidade de uma variável aleatória $ContinuousUniform(a,b)$, com $a<b$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_continuous_uniform (x,a,b)` [Função]
Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $ContinuousUniform(a,b)$, com $a<b$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`quantile_continuous_uniform (q,a,b)` [Função]
Retorna o q-quantil de uma variável aleatória $ContinuousUniform(a,b)$, com $a<b$; em outras palavras, essa função é a inversa da função `cdf_continuous_uniform`. O argumento q deve ser um elemento de $[0,1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_continuous_uniform (a,b)` [Função]
Retorna a média de uma variável aleatória $ContinuousUniform(a, b)$, com $a < b$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_continuous_uniform (a,b)` [Função]
Retorna a variância de uma variável aleatória $ContinuousUniform(a, b)$, com $a < b$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_continuous_uniform (a,b)` [Função]
Retorna o desvio padrão de uma variável aleatória $ContinuousUniform(a, b)$, com $a < b$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_continuous_uniform (a,b)` [Função]
Retorna o coeficiente de assimetria de uma variável aleatória $ContinuousUniform(a, b)$, com $a < b$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_continuous_uniform (a,b)` [Função]
Retorna o coeficiente de curtose de uma variável aleatória $ContinuousUniform(a, b)$, com $a < b$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_continuous_uniform (a,b)` [Função]
`random_continuous_uniform (a,b,n)` [Função]
Retorna uma variável estatística pseudo-aleatória $ContinuousUniform(a, b)$, com $a < b$. Chamando `random_continuous_uniform` com um terceiro argumento n, uma amostra aleatória de tamanho n será simulada.

Essa é uma aplicação directa da função `random` interna do Maxima.

Veja também `random`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_logistic (x,a,b)` [Função]
Retorna o valor em x da função densidade de probabilidade de uma variável aleatória $Logistic(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_logistic (x,a,b)` [Função]
Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $Logistic(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`quantile_logistic (q,a,b)` [Função]
Retorna o q-quantil de uma variável aleatória $Logistic(a, b)$, com $b > 0$; em outras palavras, essa função é a inversa da função `cdf_logistic`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`meanlog (a,b)` [Função]
Retorna a média de uma $Logistic(a, b)$ variável aleatória , com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_logistic (a,b)` [Função]
Retorna a variância de uma variável aleatória $Logistic(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_logistic (a,b)` [Função]
Retorna o desvio padrão de uma variável aleatória $Logistic(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_logistic (a,b)` [Função]
Retorna o coeficiente de assimetria de uma variável aleatória $Logistic(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_logistic (a,b)` [Função]
Retorna o coeficiente de curtose de uma variável aleatória $Logistic(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_logistic (a,b)` [Função]
`random_logistic (a,b,n)` [Função]
Retorna uma variável estatística pseudo-aleatória $Logistic(a, b)$, com $b > 0$. Chamando `random_logistic` com um terceiro argumento n, uma amostra aleatória de tamanho n será simulada.

Somente o método inverso genérico está implementado. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_pareto (x,a,b)` [Função]
Retorna o valor em x da função densidade de probabilidade de uma variável aleatória $Pareto(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_pareto (x,a,b)` [Função]
Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $Pareto(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`quantile_pareto (q,a,b)` [Função]
Retorna o q-quantile de uma variável aleatória $Pareto(a, b)$, com $a, b > 0$; em outras palavras, essa função é a inversa da função `cdf_pareto`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_pareto (a,b)` [Função]
Retorna a média de uma variável aleatória $Pareto(a, b)$, com $a > 1, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_pareto (a,b)` [Função]
Retorna a variância de uma variável aleatória $Pareto(a, b)$, com $a > 2, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_pareto (a,b)` [Função]
Retorna o desvio padrão de uma variável aleatória $Pareto(a, b)$, com $a > 2, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_pareto (a,b)` [Função]
Retorna o coeficiente de assimetria de uma variável aleatória $Pareto(a, b)$, com $a > 3, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_pareto (a,b)` [Função]
Retorna o coeficiente de curtose de uma variável aleatória $Pareto(a, b)$, com $a > 4, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_pareto (a,b)` [Função]
`random_pareto (a,b,n)` [Função]
Retorna uma variável estatística pseudo-aleatória $Pareto(a, b)$, com $a > 0, b > 0$. Chamando `random_pareto` com um terceiro argumento n, uma amostra aleatória de tamanho n será simulada.

Somente o método inverso genérico está implementado. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_weibull (x,a,b)` [Função]
Retorna o valor em x da função densidade de probabilidade de uma variável aleatória $Weibull(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_weibull (x,a,b)` [Função]
Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $Weibull(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`quantile_weibull (q,a,b)` [Função]
Retorna o q-quantil de uma variável aleatória $Weibull(a, b)$, com $a, b > 0$; em outras palavras, essa função é a inversa da função `cdf_weibull`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_weibull (a,b)` [Função]
Retorna a média de uma variável aleatória $Weibull(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_weibull (a,b)` [Função]
Retorna a variância de uma variável aleatória $Weibull(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_weibull (a,b)` [Função]
Retorna o desvio padrão de uma variável aleatória $Weibull(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_weibull (a,b)` [Função]
Retorna o coeficiente de assimetria de uma variável aleatória $Weibull(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_weibull (a,b)` [Função]
Retorna o coeficiente de curtose de uma variável aleatória $Weibull(a, b)$, com $a, b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_weibull (a,b)` [Função]
`random_weibull (a,b,n)` [Função]

Retorna uma variável estatística pseudo-aleatória $Weibull(a, b)$, com $a, b > 0$. Chamando `random_weibull` com um terceiro argumento n, uma amostra aleatória de tamanho n será simulada.

Somente o método inverso genérico está implementado. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_rayleigh (x,b)` [Função]

Retorna o valor em x da função densidade de probabilidade de uma variável aleatória $Rayleigh(b)$, com $b > 0$.

A variável aleatória $Rayleigh(b)$ é equivalente a $Weibull(2, 1/b)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada na função densidade de probabilidade de Weibull é retornada.

```
(%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)` [Função]

Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $Rayleigh(b)$, com $b > 0$.

A variável aleatória $Rayleigh(b)$ é equivalente a $Weibull(2, 1/b)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada na distribuição de Weibull é retornada.

```
(%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)` [Função]

Retorna o q-quantil de uma variável aleatória $Rayleigh(b)$, com $b > 0$; em outras palavras, essa função é a inversa da função `cdf_rayleigh`. O argumento q deve ser um elemento de $[0, 1]$.

A variável aleatória $Rayleigh(b)$ é equivalente a $Weibull(2, 1/b)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada no quantil de Weibull é retornada.

```
(%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*) [Função]

Retorna a média de uma variável aleatória $Rayleigh(b)$, com $b > 0$.

A variável aleatória $Rayleigh(b)$ é equivalente a $Weibull(2, 1/b)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada na meia de Weibull é retornada.

```
(%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*) [Função]

Retorna a variância de uma variável aleatória $Rayleigh(b)$, com $b > 0$.

A variável aleatória $Rayleigh(b)$ é equivalente a $Weibull(2, 1/b)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada na variância de Weibull é retornada.

```
(%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*) [Função]

Retorna o desvio padrão de uma variável aleatória $Rayleigh(b)$, com $b > 0$.

A variável aleatória $Rayleigh(b)$ é equivalente a $Weibull(2, 1/b)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada na Weibull desvio padrão é retornada.

```
(%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*) [Função]

Retorna o coeficiente de assimetria de uma variável aleatória $Rayleigh(b)$, com $b > 0$.

A variável aleatória $Rayleigh(b)$ é equivalente a $Weibull(2, 1/b)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada no coeficiente de assimetria de Weibull é retornada.

```
(%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*) [Função]

Retorna o coeficiente de curtose de uma variável aleatória $Rayleigh(b)$, com $b > 0$.

A variável aleatória $Rayleigh(b)$ é equivalente a $Weibull(2, 1/b)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada no coeficiente de curtose de Weibull é retornada.

```
(%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*) [Função]
`random_rayleigh` (*b*,*n*) [Função]

Retorna uma variável estatística pseudo-aleatória $Rayleigh(b)$, com $b > 0$. Chamando `random_rayleigh` com um segundo argumento n, uma amostra aleatória de tamanho n será simulada.

Somente o método inverso genérico está implementado. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_laplace` (*x*,*a*,*b*) [Função]

Retorna o valor em x da função densidade de probabilidade de uma variável aleatória $Laplace(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_laplace` (*x*,*a*,*b*) [Função]

Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $Laplace(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`quantile_laplace` (*q*,*a*,*b*) [Função]

Retorna o q-quantil de uma variável aleatória $Laplace(a, b)$, com $b > 0$; em outras palavras, essa função é a inversa da função `cdf_laplace`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_laplace` (*a*,*b*) [Função]

Retorna a média de uma variável aleatória $Laplace(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_laplace` (*a*,*b*) [Função]

Retorna a variância de uma variável aleatória $Laplace(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_laplace` (*a*,*b*) [Função]

Retorna o desvio padrão de uma variável aleatória $Laplace(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_laplace` (*a*,*b*) [Função]

Retorna o coeficiente de assimetria de uma variável aleatória $Laplace(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_laplace (a,b)` [Função]

Retorna o coeficiente de curtose de uma variável aleatória $Laplace(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_laplace (a,b)` [Função]
`random_laplace (a,b,n)` [Função]

Retorna uma variável estatística pseudo-aleatória $Laplace(a, b)$, com $b > 0$. Chamando `random_laplace` com um terceiro argumento n, uma amostra aleatória de tamanho n será simulada.

Somente o método inverso genérico está implementado. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_cauchy (x,a,b)` [Função]

Retorna o valor em x da função densidade de probabilidade de uma variável aleatória $Cauchy(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_cauchy (x,a,b)` [Função]

Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $Cauchy(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`quantile_cauchy (q,a,b)` [Função]

Retorna o q-quantil de uma variável aleatória $Cauchy(a, b)$, com $b > 0$; em outras palavras, essa função é a inversa da função `cdf_cauchy`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_cauchy (a,b)` [Função]
`random_cauchy (a,b,n)` [Função]

Retorna uma variável estatística pseudo aleatória $Cauchy(a, b)$, com $b > 0$. Chamando `random_cauchy` com um terceiro argumento n, uma amostra aleatória de tamanho n será simulada.

Somente o método inverso genérico está implementado. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_gumbel (x,a,b)` [Função]

Retorna o valor em x da função densidade de probabilidade de uma variável aleatória $Gumbel(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_gumbel (x,a,b)` [Função]

Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $Gumbel(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`quantile_gumbel (q,a,b)` [Função]

Retorna o q-quantil de uma variável aleatória $Gumbel(a, b)$, com $b > 0$; em outras palavras, essa função é a inversa da função `cdf_gumbel`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_gumbel (a,b)` [Função]

Retorna a média de uma variável aleatória $Gumbel(a, b)$, com $b > 0$.

```
(%i1) load ("distrib")$
(%i2) assume(b>0)$  mean_gumbel(a,b);
(%o3)                      %gamma b + a
```

onde o símbolol `%gamma` representa a constante de Euler-Mascheroni. Veja também `%gamma`.

`var_gumbel (a,b)` [Função]

Retorna a variância de uma variável aleatória $Gumbel(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_gumbel (a,b)` [Função]

Retorna o desvio padrão de uma variável aleatória $Gumbel(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_gumbel (a,b)` [Função]

Retorna o coeficiente de assimetria de uma variável aleatória $Gumbel(a, b)$, com $b > 0$.

```
(%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
```

onde `zeta` representa a função zeta de Riemann.

`kurtosis_gumbel (a,b)` [Função]

Retorna o coeficiente de curtose de uma variável aleatória $Gumbel(a, b)$, com $b > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_gumbel (a,b)` [Função]
`random_gumbel (a,b,n)` [Função]

Retorna uma variável estatística pseudo-aleatória $Gumbel(a, b)$, com $b > 0$. Chamando `random_gumbel` com um terceiro argumento n, uma amostra aleatória de tamanho n será simulada.

Somente o método inverso genérico está implementado. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

46.3 Definições para distribuições discretas

`pdf_binomial (x,n,p)` [Função]

Retorna o valor em x da função de probabilidade de uma $Binomial(n, p)$ variável aleatória, com $0 < p < 1$ e n um inteiro positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_binomial (x,n,p)` [Função]

Retorna o valor em x da função distribuição de probabilidade de uma $Binomial(n, p)$ variável aleatória, com $0 < p < 1$ e n um inteiro positivo.

`cdf_binomial` é calculada numéricamente se a variável global `numer` for igual a `true`, de outra forma `cdf_binomial` retorna uma expressão nominal.

```
(%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)` [Função]

Retorna o q-quantil de uma variável aleatória $Binomial(n, p)$, com $0 < p < 1$ e n um inteiro positivo; em outras palavras, essa função é a inversa da função `cdf_binomial`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_binomial (n,p)` [Função]

Retorna a média de uma variável aleatória $Binomial(n, p)$, com $0 < p < 1$ e n um inteiro positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_binomial (n,p)` [Função]

Retorna a variância de uma variável aleatória $Binomial(n, p)$, com $0 < p < 1$ e n um inteiro positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_binomial (n,p)` [Função]

Retorna o desvio padrão de uma variável aleatória $Binomial(n, p)$, com $0 < p < 1$ e n um inteiro positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_binomial (n,p)` [Função]

Retorna o coeficiente de assimetria de uma variável aleatória $Binomial(n, p)$, com $0 < p < 1$ e n um inteiro positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_binomial (n,p)` [Função]

Retorna o coeficiente de curtose de uma variável aleatória $Binomial(n, p)$, com $0 < p < 1$ e n um inteiro positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_binomial_algorithm` [Variável de opção]

Valor por omissão: `kachit`

Esse é o algoritmo seleccionado para simular rvariáveis estatísticas pseudo-aleatórias binomiais. Os algoritmos implementados são `kachit`, `bernoulli` e `inverse`:

- `kachit`, baseado no algoritmo descrito em Kachitvichyanukul, V. and Schmeiser, B.W. (1988) *Binomial Random Variate Generation.* Communications of the ACM, 31, Feb., 216.

- `bernoulli`, baseado na simulação testes de Bernoulli.
- `inverse`, baseado no método inverso genérico.

Veja também `random_binomial`.

`random_binomial (n,p)` [Função]
`random_binomial (n,p,m)` [Função]

Retorna uma variável estatística pseudo-aleatória $Binomial(n, p)$, com $0 < p < 1$ e n um inteiro positivo. Chamando `random_binomial` com um terceiro argumento m, uma amostra aleatória de tamanho m será simulada.

Existem três algoritmos implementado para essa função, se pode seleccionar o algoritmo a ser usado fornecendo um certo valor à variável global `random_binomial_algorithm`, cujo valor padrão é `kachit`.

Veja também `random_binomial_algorithm`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_poisson (x,m)` [Função]

Retorna o valor em x da função de probabilidade de uma variável aleatória $Poisson(m)$, com $m > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_poisson (x,m)` [Função]

Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $Poisson(m)$, com $m > 0$.

Essa função é calculada numéricamente se a variável global `numer` for igual a `true`, de outra forma essa função retorna uma expressão nominal.

```
(%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)` [Função]

Retorna o q-quantil de uma variável aleatória $Poisson(m)$, com $m > 0$; em outras palavras, essa função é a inversa da função `cdf_poisson`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_poisson (m)` [Função]

Retorna a média de uma variável aleatória $Poisson(m)$, com $m > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_poisson (m)` [Função]

Retorna a variância de uma variável aleatória $Poisson(m)$, com $m > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_poisson (m)` [Função]

Retorna o desvio padrão de uma variável aleatória $Poisson(m)$, com $m > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_poisson (m)` [Função]
Retorna o coeficiente de assimetria de uma variável aleatória $Poisson(m)$, com $m > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_poisson (m)` [Função]
Retorna o coeficiente de curtose de uma Poisson variável aleatória $Poi(m)$, com $m > 0$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_poisson_algorithm` [Variável de opção]
Valor por omissão: `ahrens_dieter`

Esse é o algoritmo seleccionado para simular variáveis estatísticas pseudo-aleatórias de Poisson.Os algoritmos implementados são `ahrens_dieter` e `inverse`:

- `ahrens_dieter`, baseado no algoritmo descrito em 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`, baseado no método inverso genérico.

Veja também `random_poisson`.

`random_poisson (m)` [Função]
`random_poisson (m,n)` [Função]
Retorna uma variável estatística pseudo-aleatória $Poisson(m)$, com $m > 0$. Chamando `random_poisson` com um segundo argumento n, uma amostra aleatória de tamanho n será simulada.

Existem dois algoritmos implementado para essa função, se pode seleccionar o algoritmo a ser usado fornecendo um certo valor à variável global `random_poisson_algorithm`, cujo valor padrão é `ahrens_dieter`.

Veja também `random_poisson_algorithm`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_bernoulli (x,p)` [Função]
Retorna o valor em x da função de probabilidade de uma variável aleatória $Bernoulli(p)$, com $0 < p < 1$.

A variável aleatória $Bernoulli(p)$ é equivalente a $Binomial(1, p)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada na função binomial de probabilidade é retornada.

```
(%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)` [Função]
Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $Bernoulli(p)$, com $0 < p < 1$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`quantile_bernoulli (q,p)` [Função]

Retorna o q-quantil de uma variável aleatória $Bernoulli(p)$, com $0 < p < 1$; em outras palavras, essa função é a inversa da função `cdf_bernoulli`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_bernoulli (p)` [Função]

Retorna a média de uma variável aleatória $Bernoulli(p)$, com $0 < p < 1$.

A variável aleatória $Bernoulli(p)$ é equivalente a $Binomial(1, p)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada na média binomial é retornada.

```
(%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)` [Função]

Retorna a variância de uma variável aleatória $Bernoulli(p)$, com $0 < p < 1$.

A variável aleatória $Bernoulli(p)$ é equivalente a $Binomial(1, p)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada na variância binomial é retornada.

```
(%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)` [Função]

Retorna o desvio padrão de uma variável aleatória $Bernoulli(p)$, com $0 < p < 1$.

A variável aleatória $Bernoulli(p)$ é equivalente a $Binomial(1, p)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada no desvio padrão binomial é retornada.

```
(%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)` [Função]

Retorna o coeficiente de assimetria de uma variável aleatória $Bernoulli(p)$, com $0 < p < 1$.

A variável aleatória $Bernoulli(p)$ é equivalente a $Binomial(1, p)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada no coeficiente de assimetria binomial é retornada.

```
(%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*) [Função]

Retorna o coeficiente de curtose de uma variável aleatória $Bernoulli(p)$, com $0 < p < 1$.

A variável aleatória $Bernoulli(p)$ é equivalente a $Binomial(1, p)$, embora quando Maxima não tiver informação disponível para pegar o resultado, uma forma nominal baseada no coeficiente de curtose binomial é retornada.

```
(%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*) [Função]

`random_bernoulli` (*p*,*n*) [Função]

Retorna uma variável estatística pseudo-aleatória $Bernoulli(p)$, com $0 < p < 1$. Chamando `random_bernoulli` com um segundo argumento n, uma amostra aleatória de tamanho n será simulada.

Essa é uma aplicação directa da função `random` built-in função do Maxima.

Veja também `random`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_geometric` (*x*,*p*) [Função]

Retorna o valor em x da função de probabilidade de uma variável aleatória $Geometric(p)$, com $0 < p < 1$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_geometric` (*x*,*p*) [Função]

Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $Geometric(p)$, com $0 < p < 1$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`quantile_geometric` (*q*,*p*) [Função]

Retorna o q-quantil de uma variável aleatória $Geometric(p)$, com $0 < p < 1$; em outras palavras, essa função é a inversa da função `cdf_geometric`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_geometric` (*p*) [Função]

Retorna a média de uma variável aleatória $Geometric(p)$, com $0 < p < 1$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_geometric` (*p*) [Função]
Retorna a variância de uma variável aleatória *Geometric*(p), com $0 < p < 1$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_geometric` (*p*) [Função]
Retorna o desvio padrão de uma variável aleatória *Geometric*(p), com $0 < p < 1$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_geometric` (*p*) [Função]
Retorna o coeficiente de assimetria de uma variável aleatória *Geometric*(p), com $0 < p < 1$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_geometric` (*p*) [Função]
Retorna o coeficiente de curtose de uma geometric variável aleatória *Geo*(p), com $0 < p < 1$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_geometric_algorithm` [Variável de opção]
Valor por omissão: `bernoulli`

Esse é o algoritmo seleccionado para simular variáveis estatísticas pseudo-aleatórias geométricas. Algorítmos implementados são `bernoulli`, `devroye` e `inverse`:

- `bernoulli`, baseado na simulação de testes de Bernoulli.
- `devroye`, baseado no algoritmo descrito em Devroye, L. (1986) *Non-Uniform Random Variate Generation.* Springer Verlag, p. 480.
- `inverse`, baseado no método inverso genérico.

Veja também `random_geometric`.

`random_geometric` (*p*) [Função]
`random_geometric` (*p*,*n*) [Função]
Retorna um *Geometric*(p) variável estatística pseudo-aleatória, com $0 < p < 1$. Chamando `random_geometric` com um segundo argumento n, uma amostra aleatória de tamanho n será simulada.

Existem três algoritmos implementados para essa função, se pode seleccionar o algoritmo a ser usado fornecendo um certo valor à variável global `random_geometric_algorithm`, cujo valor padrão é `bernoulli`.

Veja também `random_geometric_algorithm`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_discrete_uniform` (*x*,*n*) [Função]
Retorna o valor em x da função de probabilidade de uma variável aleatória $DiscreteUniform(n)$, com n a strictly positive integer. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_discrete_uniform` (*x*,*n*) [Função]
Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $DiscreteUniform(n)$, com n inteiro estritamente positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`quantile_discrete_uniform (q,n)` [Função]
Retorna o q-quantil de uma variável aleatória $DiscreteUniform(n)$, com n um inteiro estritamente positivo; em outras palavras, essa função é a inversa da função `cdf_discrete_uniform`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_discrete_uniform (n)` [Função]
Retorna a média de uma variável aleatória $DiscreteUniform(n)$, com n um inteiro estritamente positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_discrete_uniform (n)` [Função]
Retorna a variância de uma variável aleatória $DiscreteUniform(n)$, com n um inteiro estritamente positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_discrete_uniform (n)` [Função]
Retorna o desvio padrão de uma variável aleatória $DiscreteUniform(n)$, com n um inteiro estritamente positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_discrete_uniform (n)` [Função]
Retorna o coeficiente de assimetria de uma variável aleatória $DiscreteUniform(n)$, com n um inteiro estritamente positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_discrete_uniform (n)` [Função]
Retorna o coeficiente de curtose de uma variável aleatória $DiscreteUniform(n)$, com n um inteiro estritamente positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_discrete_uniform (n)` [Função]
`random_discrete_uniform (n,m)` [Função]
Retorna uma variável estatística pseudo-aleatória $DiscreteUniform(n)$, com n um inteiro estritamente positivo. Chamando `random_discrete_uniform` com um segundo argumento m, uma amostra aleatória de tamanho m será simulada.

Isso é uma aplicação directa da função `random` built-in função do Maxima.

Veja também `random`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_hypergeometric (x,n1,n2,n)` [Função]
Retorna o valor em x da função de probabilidade de uma variável aleatória $Hypergeometric(n1, n2, n)$, com *n1*, *n2* e n inteiros não negativos e $n <= n1 + n2$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_hypergeometric (x,n1,n2,n)` [Função]
Retorna o valor em x da função distribuição de probabilidade de uma variável aleatória $Hypergeometric(n1, n2, n)$, com *n1*, *n2* e n inteiros não negativos e $n <= n1 + n2$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`quantile_hypergeometric (q,n1,n2,n)` [Função]
Retorna o q-quantil de uma variável aleatória $Hypergeometric(n1, n2, n)$, com *n1*, *n2* e n inteiros não negativos e $n <= n1 + n2$; em outras palavras, essa função é a inversa da função `cdf_hypergeometric`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_hypergeometric (n1,n2,n)` [Função]
Retorna a média de uma variável aleatória discreta univorme $Hyp(n1, n2, n)$, com *n1*, *n2* e n inteiros não negativos e $n <= n1 + n2$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_hypergeometric (n1,n2,n)` [Função]
Retorna a variância de uma variável aleatória hipergeométrica $Hyp(n1, n2, n)$, com *n1*, *n2* e n inteiros não negativos e $n <= n1+n2$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_hypergeometric (n1,n2,n)` [Função]
Retorna o desvio padrão de uma variável aleatória $Hypergeometric(n1, n2, n)$, com *n1*, *n2* e n inteiros não negativos e $n <= n1+n2$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_hypergeometric (n1,n2,n)` [Função]
Retorna o coeficiente de assimetria de uma variável aleatória $Hypergeometric(n1, n2, n)$, com *n1*, *n2* e n inteiros não negativos e $n <= n1 + n2$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_hypergeometric (n1,n2,n)` [Função]
Retorna o coeficiente de curtose de uma variável aleatória $Hypergeometric(n1, n2, n)$, com *n1*, *n2* e n inteiros não negativos e $n <= n1 + n2$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_hypergeometric_algorithm` [Variável de opção]
Valor por omissão: `kachit`

Esse é o algoritmo seleccionado para simular variáveis estatísticas pseudo aleatórias hipergeométricas.Os algoritmos implementados são `kachit` e `inverse`:

- `kachit`, baseado no algoritmo descrito em Kachitvichyanukul, V., Schmeiser, B.W. (1985) *Computer generation of hypergeometric variáveis estatística pseudo-aleatórias.* Journal of Statistical Computation and Simulation 22, 127-145.
- `inverse`, baseado no método inverso genérico.

Veja também `random_hypergeometric`.

`random_hypergeometric (n1,n2,n)` [Função]
`random_hypergeometric (n1,n2,n,m)` [Função]
Retorna uma variável estatística pseudo-aleatória $Hypergeometric(n1, n2, n)$, com *n1*, *n2* e n inteiros não negativos e $n <= n1+n2$. Chamando `random_hypergeometric` com um quarto argumento m, uma amostra aleatória de tamanho m será simulada.

Existem dois algoritmos implementados para essa função, se pode seleccionar o algoritmo a ser usado fornecendo um certo valor à variável global `random_hypergeometric_algorithm`, cujo valor padrão é `kachit`.

Veja também `random_hypergeometric_algorithm`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`pdf_negative_binomial (x,n,p)` [Função]

Retorna o valor em x da função de probabilidade de uma variável aleatória $NegativeBinomial(n, p)$, com $0 < p < 1$ e n um inteiro positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`cdf_negative_binomial (x,n,p)` [Função]

Retorna o valor em x da função distribuição de probabilidade de uma $NegativeBinomial(n, p)$ variável aleatória, com $0 < p < 1$ e n um inteiro positivo.

Essa função é calculada numéricamente se a variável global `numer` for igual a `true`, de outra forma essa função retorna uma expressão nominal.

```
(%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)` [Função]

Retorna o q-quantil de uma variável aleatória $NegativeBinomial(n, p)$, com $0 < p < 1$ e n um inteiro positivo; em outras palavras, essa função é a inversa da função `cdf_negative_binomial`. O argumento q deve ser um elemento de $[0, 1]$. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`mean_negative_binomial (n,p)` [Função]

Retorna a média de uma variável aleatória $NegativeBinomial(n, p)$, com $0 < p < 1$ e n um inteiro positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`var_negative_binomial (n,p)` [Função]

Retorna a variância de uma variável aleatória $NegativeBinomial(n, p)$, com $0 < p < 1$ e n um inteiro positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`std_negative_binomial (n,p)` [Função]

Retorna o desvio padrão de uma variável aleatória $NegativeBinomial(n, p)$, com $0 < p < 1$ e n um inteiro positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`skewness_negative_binomial (n,p)` [Função]

Retorna o coeficiente de assimetria de uma variável aleatória $NegativeBinomial(n, p)$, com $0 < p < 1$ e n um inteiro positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`kurtosis_negative_binomial (n,p)` [Função]

Retorna o coeficiente de curtose de uma variável aleatória $NegativeBinomial(n, p)$, com $0 < p < 1$ e n um inteiro positivo. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

`random_negative_binomial_algorithm` [Variável de opção]

Valor por omissão: `bernoulli`

Esse é o algoritmo seleccionado para simular variáveis estatísticas pseuso-aleatórias binomiais negativas. Os algoritmos implementados são `devroye`, `bernoulli` e `inverse`:

- `devroye`, baseado no algoritmo descrito em Devroye, L. (1986) *Non-Uniform Random Variate Generation*. Springer Verlag, p. 480.
- `bernoulli`, baseado na simulação de testes de Bernoulli.
- `inverse`, baseado no método inverso genérico.

Veja também `random_negative_binomial`.

`random_negative_binomial` (*n*,*p*) [Função]
`random_negative_binomial` (*n*,*p*,*m*) [Função]

Retorna uma variável estatística pseudo-aleatória $NegativeBinomial(n, p)$, com $0 < p < 1$ e n um inteiro positivo. Chamando `random_negative_binomial` com um terceiro argumento m, uma amostra aleatória de tamanho m será simulada.

Existem três algoritmos implementados para essa função, se pode seleccionar o algoritmo a ser usado fornecendo um certo valor à variável global `random_negative_binomial_algorithm`, cujo valor padrão é `bernoulli`.

Veja também `random_negative_binomial_algorithm`. Para fazer uso dessa função, escreva primeiramente `load("distrib")`.

47 dynamics

47.1 O pacote dynamics

O pacote adicional `dynamics` inclui várias funções para criar diversas representações gráficas de sistemas dinâmicos e fractais, para além duma implementaçõ do método numérico de Runge-Kutta de quarta ordem, para resolver sistemas de equações diferenciais.

Para usar as funções neste pacote será necessário primeiro que tudo carregá-lo com `load("dynamics")`; as funções que criam gráficos precisam que o Xmaxima esteja instalado.

47.2 Análise gráfica de sistemas dinâmicos discretos

`chaosgame` (*[[x1, y1]...[xm, ym]], [x0, y0], b, n, ...opções...*); [Função]

Usa o método designado de *jogo do caos*, para produzir fractais: desenha-se um ponto inicial (*x0*, *y0*) e logo escolhe-se aleatoriamente um dos *m* pontos `[`*x1*, *y1*`]`...`[`*xm*, *ym*`]`. A seguir, desenha-se um novo ponto que estará no segmento entre o último ponto desenhado e o ponto que se acabou de seleccionar aleatoriamente, a uma distância do ponto seleccionado que será *b* vezes o comprimento do segmento. O processo repete-se *n* vezes.

`evolution` (*F, y0, n,...opções...*); [Função]

Desenha *n+1* pontos num gráfico bidimensional (série de tempo), onde as coordenadas horizontais dos pontos são os números inteiros 0, 1, 2, ..., *n*, e as coordenadas verticais são os valores *y(n)* correspondentes, obtidos a partir da relação de recorrência

$$y_{n+1} = F(y_n)$$

Com valor inicial *y(0)* igual a *y0*. *F* deverá ser uma expressão que dependa unicamente da variável *y* (e não de *n*), *y0* deverá ser um número real e *n* um número inteiro positivo.

`evolution2d` (*[F, G], [x0, y0], n, ...opções...*); [Função]

Mostra, num gráfico bidimensional, os primeiros *n+1* pontos da sucessão definida a partir do sistema dinâmico discreto com relações de recorrência:

$$\begin{cases} x_{n+1} = F(x_n, y_n) \\ y_{n+1} = G(x_n, y_n) \end{cases}$$

Com valores iniciais *x0* e *y0*. *F* e *G* deverão ser duas expressões que dependam unicamente de *x* e *y*.

`ifs` (*[r1,...,rm],[A1,...,Am], [[x1,y1]...[xm, ym]], [x0,y0],* [Função]
n, ...opções...);

Usa o método do Sistema de Funções Iteradas (IFS, em inglês *Iterated Function System*). Esse método é semelhante ao método descrito na função `chaosgame`, mas em

vez de aproximar o último ponto para ponto seleccionado aleatoriamente, as duas coordenadas do último ponto multiplicam-se por uma matriz 2 por 2 *Ai* correspondente ao ponto que tenha sido escolhido aleatoriamente.

A selecção aleatória de um dos *m* pontos atractivos pode se realizada com uma função de probabilidade não uniforme, definida com os pesos *r1*,...,*rm*. Pesos esses que deverão ser dados em forma acumulada; por exemplo, se quiser usar 3 pontos com probabilidades 0.2, 0.5 e 0.3, os pesos *r1*, *r2* e *r3* poderiam ser 2, 7 e 10, ou qualquer outro grupo de números que estejam na mesma proporção.

`orbits` (*F*, *y0*, *n1*, *n2*, [*x*, *x0*, *xf*, *xstep*], *...opções...*); [Função]

Desenha o diagrama de órbitas duma família de sistemas dinâmicos discretos unidimensionais, com um parâmetro *x*; esse tipo de diagrama usa-se para mostrar as bifurcações dum sistema discreto unidimensional.

A função *F(y)* define uma sequência que começa com um valor inicial *y0*, igual que no caso da função `evolution`, mas neste caso a função também dependerá do parâmetro *x*, o qual terá valores compreendidos no intervalo de *x0* a *xf*, com incrementos *xstep*. Cada valor usado para o parâmetro *x* apresenta-se no eixo horizontal. No eixo vertical apresentam-se *n2* valores da sucessão *y(n1+1)*,..., *y(n1+n2+1)*, obtidos após deixá-la evoluir durante *n1* iterações iniciais.

`rk` (*EDO*, *var*, *inicial*, *dominio*) [Função]

`rk` ([*EDO1*,...,*EDOm*], [*v1*,...,*vm*], [*inic1*,...,*inicm*], *domínio*) [Função]

A primeira forma usa-se para resolver numericamente uma equação diferencial ordinária de primeira ordem (EDO), e a segunda forma resolve numericamente um sistema de *m* dessas equações, usando o método de Runge-Kutta de quarta ordem. *var* representa a variável dependente. EDO deverá ser uma expressão que dependa unicamente das variáveis independente e dependente, e define a derivada da variável dependente em função da variável independente.

A variável independente representa-se com *domínio*, que deverá ser uma lista com quatro elementos, como, por exemplo:

```
[t, 0, 10, 0.1]
```

o primeiro elemento da lista identifica a variável independente, os segundo e terceiro elementos são os valores inicial e final para essa variável, e o último elemento dá o valor dos incrementos que deverão ser usados dentro desse intervalo.

Se se estiverem a resolver *m* equações, deverá haver *m* variáveis dependentes *v1*, *v2*, ..., *vm*. Os valores iniciais para essas variáveis serão *inic1*, *inic2*, ..., *inicm*. Continuará existindo apenas uma variável independente, definida pela lista *domain*, tal como no caso anterior. *EDO1*, ..., *EDOm* são as expressões que definem as derivadas de cada uma das variáveis dependentes, em função da variável independente. As únicas variáveis que podem aparecer em cada uma dessas expressões são a variável independente e qualquer uma das variáveis dependentes. É importante que as derivadas *EDO1*, ..., *EDOm* sejam colocadas na lista na mesma ordem em que forem agrupadas as variáveis dependentes; por exemplo, o terceiro elemento da lista será interpretado como a derivada da terceira variável dependente.

O programa tenta integrar as equações desde o valor inicial da variável independente, até o valor final, usando incrementos fixos. Se em algum passo uma das variáveis

dependentes atingir um valor absoluto muito elevado, a integração será interrompida nesse ponto. O resultado será uma lista com um número de elementos igual ao número de iterações realizadas. Cada elemento na lista de resultados é também uma lista com *m*+1 elementos: o valor da variável independente, seguido dos valores das variáveis dependentes nesse ponto.

`staircase` (*F*, *y0*, *n*, *...opções...*); [Função]

Desenha um diagrama de degraus (ou diagrama de teia de aranha) para a sucessão definida pela equação de recorrência

$$y_{n+1} = F(y_n)$$

A interpretação e valores permitidos dos parâmetros de entrada é igual que para a função `evolution`. Um diagrama de degraus consiste num gráfico da função *F(y)*, junto com a recta *G(y)* = *y*. Começa-se por desenhar um segmento vertical desde o ponto (*y0*, *y0*) na recta, até o ponto de intersecção com a função *F*. A seguir, desde esse ponto desenha-se um segmento horizontal até o ponto de intersecção com a recta, (*y1*, *y1*); o processo repete-se *n* vezes até alcançar o ponto (*yn*, *yn*).

Opções

Cada opção é uma lista com dois ou mais elementos. O primeiro elemento na lista é o nome da opção e os restantes são os argumentos para essa opção.

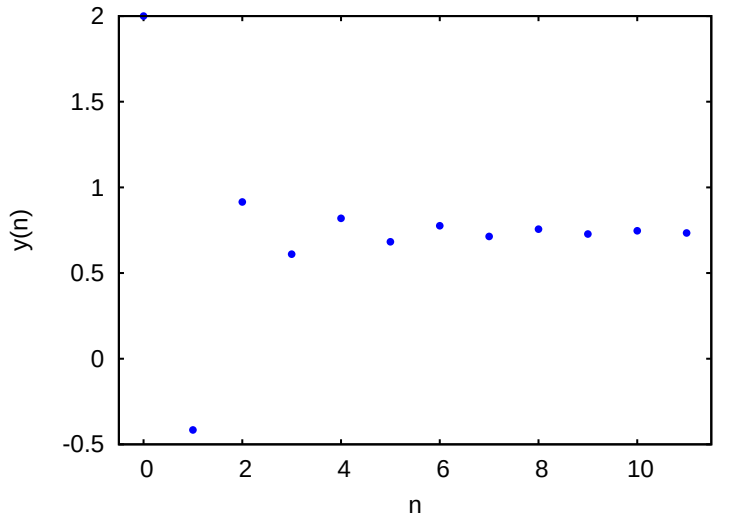
As opções aceites pelas funções evolution, evolution2, staircase, orbits, ifs e chaosgame são as seguintes:

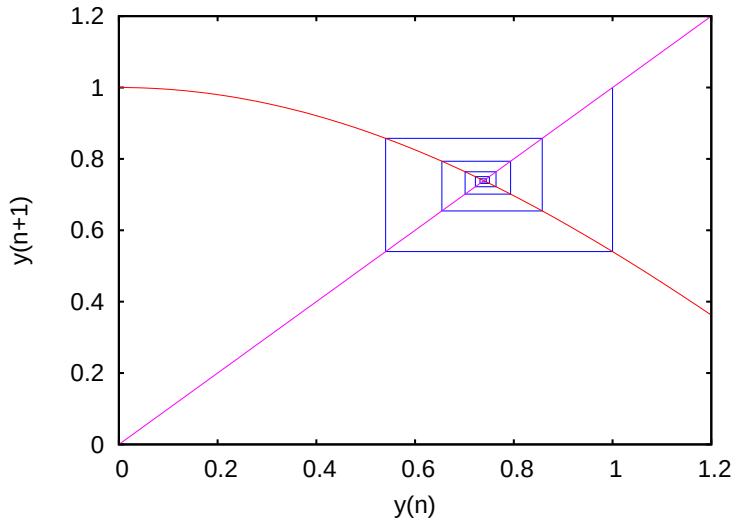
- *domain* especifica os valores mínimo y máximo da variável independente para o gráfico da função *F* representada por `staircase`.
- *pointsize* define o raio de cada ponto desenhado, em unidades de pontos. O valor por omissão é 1.
- *xaxislabel* é o nome que será dado ao eixo horizontal.
- *xcenter* é a coordenada x do ponto que deverá aparecer no centro do gráfico. Esta opção não é usada pela função `orbits`.
- *xradius* é metade do comprimento do intervalo de valores de x que serão representados. Esta opção não é usada pela função `orbits`.
- *yaxislabel* é o nome que será dado ao eixo vertical.
- *ycenter* é a coordenada y do ponto que deverá aparecer no centro do gráfico.
- *yradius* é metade do comprimento do intervalo de valores de y que serão representados.

As opções aceites pelos programas juli **Exemplos**

Representação gráfica e diagrama de degraus da sequência: 2, cos(2), cos(cos(2)),...

```
(%i1) load("dynamics")$
(%i2) evolution(cos(y), 2, 11, [yaxislabel, "y"], [xaxislabel,"n"]);
(%i3) staircase(cos(y), 1, 11, [domain, 0, 1.2]);
```



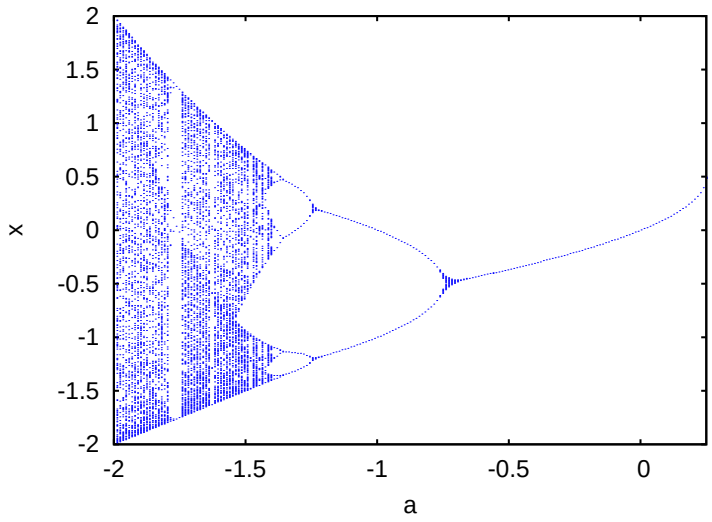


Se o seu processador for lento, terá que reduzir o número de iterações usado nos exemplos seguintes. E o valor de *pointsize* que dá os melhores resultados dependerá do monitor e da resolução usada. Terá que experimentar com diferentes valores.

Diagrama de órbitas para o mapa quadrático

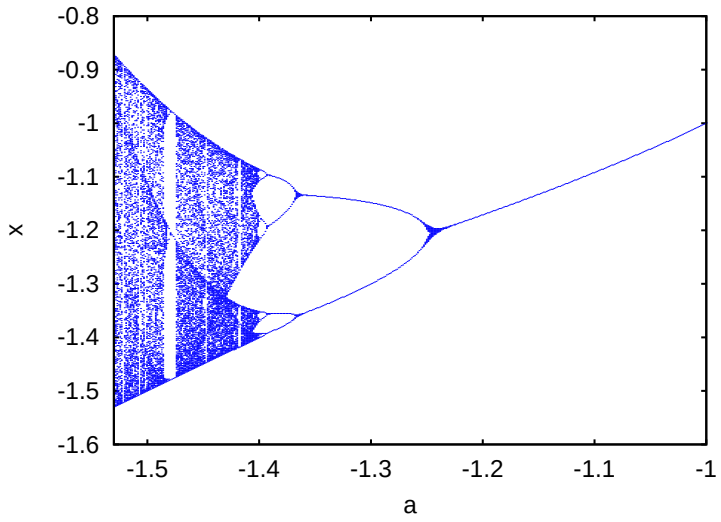
$$y_{n+1} = x + y_n^2$$

```
(%i4) orbits(y^2+x, 0, 50, 200, [x, -2, 0.25, 0.01], [pointsize, 0.9]);
```



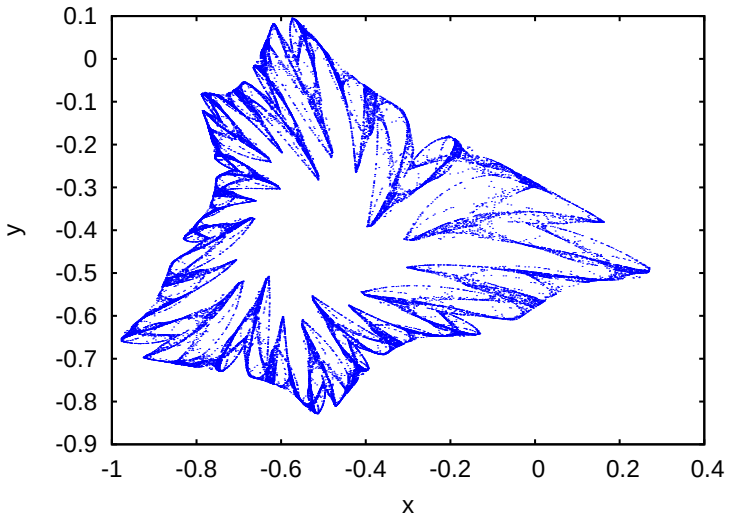
Para ampliar a região à volta da bifurcação na parte de baixo, perto de x = -1.25, use o comando:

```
(%i5) orbits(x+y^2, 0, 100, 400, [x,-1,-1.53,-0.001], [pointsize,0.9],
                [ycenter,-1.2], [yradius,0.4]);
```



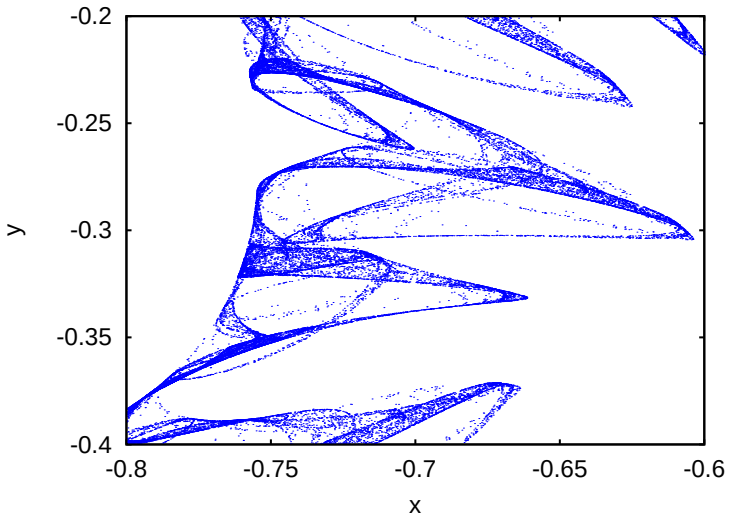
Evolução dum sistema em duas dimensões, que conduz a um fractal:

```
(%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],[-0.5,0],50000,[pointsize,0.7]);
```



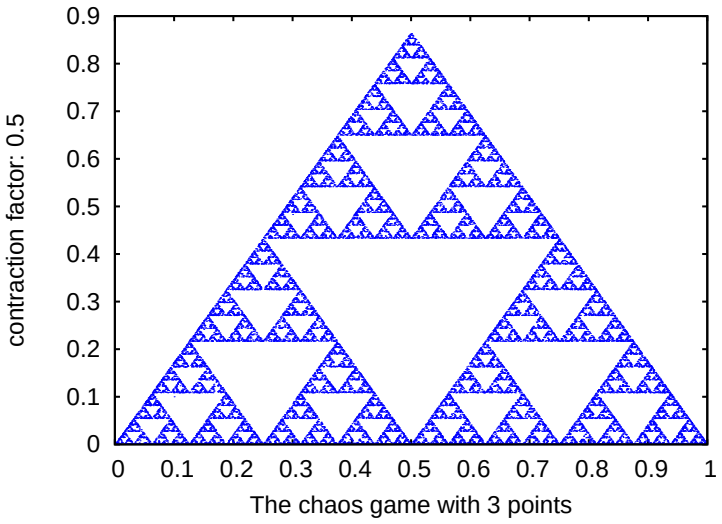
E uma ampliação de uma pequena região no fractal:

```
(%i9) evolution2d([f,g],[-0.5,0],300000,[pointsize,0.7], [xcenter,-0.7],
                  [ycenter,-0.3],[xradius,0.1],[yradius,0.1]);
```



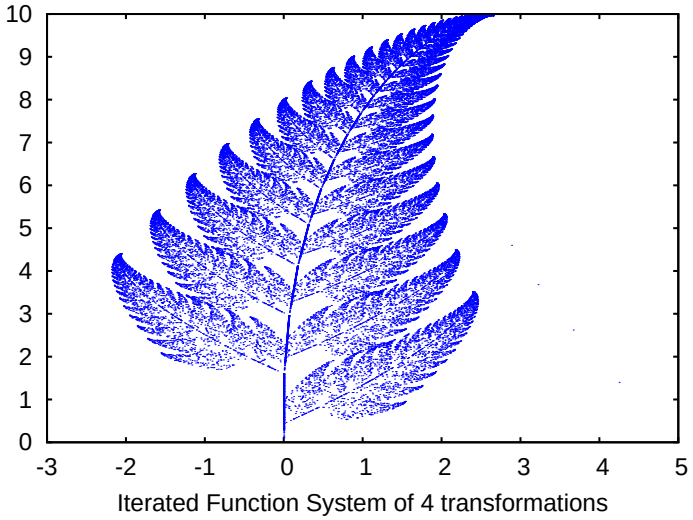
Um gráfico do triângulo de Sierpinsky, obtido com o jogo do caos:

```
(%i9) chaosgame([[0, 0], [1, 0], [0.5, sqrt(3)/2]], [0.1, 0.1], 1/2,
                30000, [pointsize,0.7]);
```



O feto de Barnsley, obtido com o Sistema de Funções Iteradas:

```
(%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,[pointsize,0.9]);
```



Para resolver numericamente a equação diferencial

$$\frac{dx}{dt} = t - x^2$$

Com valor inicial x(t=0) = 1, no intervalo de t desde 0 até 8, e com incrementos de 0.1, usa-se:

```
(%i20) resultados: rk(t-x^2,x,1,[t,0,8,0.1])$
```

os resultados ficarão armazenados na lista resultados.

Para resolver numericamente o sistema:

$$\begin{cases} \frac{dx}{dt} = 4 - x^2 - 4y^2 \\ \frac{dy}{dt} = y^2 - x^2 + 1 \end{cases}$$

para t entre 0 e 4, com valores iniciais -1.25 e 0.75 para (x, y) em 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])$
```

47.3 Visualização usando VTK

A função scene cria imagens a três dimensões e animações, usando o software *Visualization ToolKit* (VTK). Para poder usar essa função é necessário ter Xmaxima e VTK instalados no sistema (incluindo a libraria para utilizar VTK desde TCL, que pode vir num pacote separado em alguns sistemas).

48 eval_string

48.1 Definições para eval_string

`eval_string` (*str*) [Função]

Entrega a sequência de caracteres do Maxima *str* como uma expressão do Maxima e a avalia. *str* é uma sequência de caracteres do Maxima. Essa sequência pode ou não ter um marcador de final (sinal de dólar `$` ou ponto e vírgula `;`). Somente a primeira expressão é entregue e avaliada, se ouver mais de uma.

Reclama se *str* não for uma sequência de caracteres do Maxima.

Exemplos:

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

(%i2) eval_string ("foo: 42; bar: foo^2 + baz");
(%o2)                          42
(%i3) eval_string ("(foo: 42, bar: foo^2 + baz)");
(%o3)                     baz + 1764
```

Para usar essa função escreva primeiro `load("eval_string")`. Veja também `parse_string`.

`parse_string` (*str*) [Função]

Entrega a sequência de caracteres do Maxima *str* como uma expressão do Maxima (sem fazer nenhuma avaliação dessa expressão). *str* é uma sequência de caracteres do Maxima. Essa sequência pode ou não ter um marcador de final (sinal de dólar `$` ou ponto e vírgula `;`). Somente a primeira expressão é entregue e avaliada, se ouver mais de uma.

Reclama se *str* não for uma sequência de caracteres do Maxima.

Exemplos:

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

(%i2) parse_string ("foo: 42; bar: foo^2 + baz");
(%o2)                      foo : 42
(%i3) parse_string ("(foo: 42, bar: foo^2 + baz)");
                                    2
(%o3)           (foo : 42, bar : foo  + baz)
```

Para usar essa função escreva primeiro `load("eval_string")`. Veja também a função `eval_string`.

49 f90

49.1 Definições para f90

`f90` (*expr*) [Função]

O comando f90 é uma actualização do comando `fortran` original do maxima. A principal diferença é na forma como são divididas as linhas muito compridas.

No exemplo seguinte, observe como o comando `fortran` divide linhas no meio de símbolos. O comando `f90` nunca separa uma linha no meio de um símbolo.

```
(%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
```

A função `f90` foi criada como uma forma rápida de resolver um problema. Não será necessariamente um bom exemplo a seguir para traduções de outras linguagens de programação.

Para usar esta função, use primeiro `load("f90")`.

50 ggf

50.1 Definições para ggf

`GGFINFINITY` [Variável de Opção]

Valor por omissão: 3

Essa é uma variável de opção para a função `ggf`.

Quando calculando a fração contínua da função geradora, um quociente parcial tendo um grau (estritamente) maior que *GGFINFINITY* será descartado e o convergente actual será considerado como o valor exato da função geradora; na grande mioria dos casos o grau de todos os quocientes parciais será ou 0 ou 1; se usar um valor muito grande, então poderá fornecer termos suficientes com o objectivo de fazer o cálculo preciso o bastante.

Veja também `ggf`.

`GGFCFMAX` [Variável de opção]

Valor por omissão: 3

Essa é uma variável de opção para a função `ggf`.

Quando calculando a fração contínua da função geradora, se nenhum bom resultado for encontrado (veja o sinalizador *GGFINFINITY*) após se ter calculado uma quantidade de *GGFCFMAX* quocientes parciais, a função geradora será considerada como não sendo uma fração de dois polinómios e a função irá terminar. Coloque livemente um valor muito grande para funções geradoras mais complicadas.

Veja também `ggf`.

`ggf` (*l*) [Função]

Calcula a função geradora (se for uma fração de dois polinómios) de uma sequência, sendo dados seus primeiros termos. *l* é uma lista de números.

A solução é retornada como uma fração de dois polinómios. Se nenhuma solução tiver sido encontrada, é retornado `done`.

Essa função é controlada attravés das variáveis globais *GGFINFINITY* e *GGFCFMAX*. Veja também *GGFINFINITY* e *GGFCFMAX*.

Para usar essa função primeiro escreva `load("ggf")`.

51 impdiff

51.1 Definições para impdiff

`implicit_derivative` (*f*,*indvarlist*,*orderlist*,*depvar*) [Função]

Essa subrotina calcula derivadas implícitas de funções de várias variáveis. *f* é uma função do tipo array, os índices são o grau da derivada na ordem *indvarlist*; *indvarlist* é a lista de variáveis independentes; *orderlist* é a ordem desejada; e *depvar* é a variável dependente.

Para usar essa função escreva primeiro `load("impdiff")`.

52 interpol

52.1 Introdução a interpol

Pacote `interpol` define os métodos Lagrangiano, linear e o de splines cúbicos para interpolação polinomial.

Comentários, correções e sugestões, por favor contacte-me em *'mario AT edu DOT xunta DOT es'*.

52.2 Definições para interpol

`lagrange` (*pontos*) [Função]
`lagrange` (*pontos*, *opção*) [Função]

Calcula a interpolação polinomial através do método Lagrangiano. O argumento *pontos* deve ser um dos seguintes:

- uma matriz de duas colunas, `p:matrix([2,4],[5,6],[9,3])`,
- uma lista de pares, `p: [[2,4],[5,6],[9,3]]`,
- uma lista de números, `p: [4,6,3]`, e nesse caso as abcissas irão ser atribuídas automaticamente aos valores 1, 2, 3, etc.

Nos dois primeiros casos os pares são ordenados em relação à primeira coordenada antes de fazer os cálculos.

Com o argumento *opção* é possível escolher o nome da variável independente, o qual é `'x` por padrão; para definir qualquer outra, z por exemplo, escreva `varname='z`.

Exemplos:

```
(%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) /* Plot the polynomial together with points */
      plot2d([f(x),[discrete,p]],[x,0,10],
           [gnuplot_curve_styles,
                ["with lines","with points pointsize 3"]])$
(%i8) /* Change variable name */
      lagrange(p, varname=w);
                 4        3         2
               73 w    701 w    8957 w    5288 w   186
(%o8)          ----- - ------ + ------- - ------ + ---
                420     210       420      105      5
```

`charfun2` (*x*, `a`, *b*) [Função]

Retorna `true`, i. e., verdadeiro se o número *x* pertence ao intervalo $[a, b)$, e `false`, i. e., falsono caso contrário.

`linearinterpol` (*pontos*) [Função]
`linearinterpol` (*pontos*, *opção*) [Função]

Calcula a interpolação polinomial através do método linear. O argumento *pontos* deve ser um dos seguintes:

- uma matriz de duas colunas, `p:matrix([2,4],[5,6],[9,3])`,
- uma lista de pares, `p: [[2,4],[5,6],[9,3]]`,
- uma lista de números, `p: [4,6,3]`, e nesse caso as abcissas irão ser atribuídas automaticamente aos valores 1, 2, 3, etc.

Nos dois primeiros casos os pares são ordenados em relação à primeira coordenada antes de fazer os cálculos.

Com o argumento *opção* é possível escolher o nome da variável independente, o qual é `'x` por padrão; para definir qualquer outra, z por exemplo, escreva `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)  /* Plot the polynomial together with points */
       plot2d(['(f(x)),[discrete,args(p)]],[x,-5,20],
           [gnuplot_curve_styles,
                 ["with lines","with points pointsize 3"]])$
(%i8)  /* Change variable name */
       linearinterpol(p, varname='s);
       13   3 s
(%o8) (-- - ---) 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` (*pontos*) [Função]
`cspline` (*pontos*, *opção1*, *opção2*, ...) [Função]

Calcula a interpolação polnomial pelo método de splines (polinómios de ordem k que interpolam os dados e têm k-1 derivadas contínuas em todo o intervalo) cúbicos. O argumento *pontos* deve ser um dos seguintes:

- uma matriz de duas colunas, `p:matrix([2,4],[5,6],[9,3])`,
- uma lista de pares, `p: [[2,4],[5,6],[9,3]]`,
- uma lista de números, `p: [4,6,3]`, e nesse caso as abcissas irão ser atribuídas automaticamente aos valores 1, 2, 3, etc.

Nos dois primeiros casos os pares são ordenados em relação à primeira coordenada antes de fazer os cálculos.

Existem três opções para ajustar necessidades específicas:

- `'d1`, o padrão é `'unknown`, é a primeira derivada em x_1; se essa primeira derivada for desconhecida, `'unknown`, a segunda derivada em x_1 é igualada a 0 (o spline cúbico natural); se essa primeira derivada for igual a um número, a segunda derivada é calculada baseando-se nesse número.
- `'dn`, o padrão é `'unknown`, é a primeira derivada em x_n; se essa primeira derivada for desconhecida, `'unknown`, a segunda derivada em x_n é igualada a 0 (o spline cúbico natural); se essa primeira derivada for igual a um número, a segunda derivada é calculada baseando-se nesse número.

- `'nome_var`, o padrão é `'x`, é o nome da variável independente.

Exemplos:

```
(%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) /* Plotting interpolating function */
      plot2d(['(f(x)),[discrete,p]],[x,0,10],
          [gnuplot_curve_styles,
               ["with lines","with points pointsize 3"]])$
(%i7) /* New call, but giving values at the derivatives */
      cspline(p,d1=0,dn=0);
              3          2
        1949 x    11437 x    17027 x   1247
(%o7)  (------- - -------- + ------- + ----) 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 */
      plot2d(['(f(x)),'(g(x)),[discrete,p]],[x,0,10],
           [gnuplot_curve_styles,
              ["with lines","with lines","with points pointsize 3"]])$
```

53 lbfgs

53.1 Introdução a lbfgs

`lbfgs` é uma implementação do algoritmo[1] L-BFGS (Broyden-Fletcher-Goldfarb-Shanno) para resolver problemas de minimização não limitada através de um algoritmo de memória limitada quasi-Newton (BFGS). Esse algoritmo é chamado de método de memória limitada porque uma aproximação de baixo ranque da inverso da matriz Hessiana é armazenado em lugar da inversa da matriz Hessiana completa. O programa foi escrito originàriamente em Fortran [2] por Jorge Nocedal, incorporando algumas funções originalmente escritas por Jorge J. Moré e David J. Thuente, e traduzidas para Lisp automaticamente através do programa `f2cl`. O pacote do Maxima `lbfgs` compreende o código traduzido e adicionalmente uma interface de função que gerencia alguns detallhes.

Referências:

[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

53.2 Definições para lbfgs

`lbfgs` (*FOM*, *X*, *X0*, *epsilon*, *iprint*) [Função]

Encontra uma solução aproximada da minimização não limitada de número de mérito *FOM* sobre a lista de variáveis *X*, começando a partir da estimativa inicial *X0*, tal que $normgradFOM < epsilonmax(1, normX)$.

O algoritmo aplicado é um algoritmo de memória limitada[1] quasi-Newton (BFGS). Esse algoritmo é chamado de método de memória limitada porque uma aproximação de baixo ranque da inverso da matriz Hessiana é armazenado em lugar da inversa da matriz Hessiana completa.

iprint controla as messaens de progresso mostradas através de `lbfgs`.

`iprint[1]`

iprint`[1]` controla a frequência das mensagens de progresso.

`iprint[1] < 0`

Nenhuma mensagem de progresso.

`iprint[1] = 0`

Messagens na primeira iteração e na última iteração.

`iprint[1] > 0`

Mostra uma mensagem a cada *iprint*`[1]` iterações.

`iprint[2]`

iprint`[2]` controla a quantidade de informações fornecidas pelas mensagens de progresso (verbosidade).

`iprint[2] = 0`

Mostra na tela o contador de iterações, o número de avaliações de *FOM*, o valor de *FOM*, a norma do gradiente de *FOM*, e o comprimento do salto.

`iprint[2] = 1`
: O mesmo que *`iprint`*`[2] = 0`, adicionando *X0* e o gradiente de *FOM* avaliado em *X0*.

`iprint[2] = 2`
: O mesmo que *`iprint`*`[2] = 1`, adicionando valores de *X* a cada iteração.

`iprint[2] = 3`
: O mesmo que *`iprint`*`[2] = 2`, adicionando o gradiente de *FOM* a cada iteração.

Veja também `lbfgs_nfeval_max` e `lbfgs_ncorrections`.

Referências:

[1] D. Liu and J. Nocedal. "On the limited memory BFGS method for large scale optimization". *Mathematical Programming B* 45:503–528 (1989)

Exemplo:

```
(%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-
   2    6     2.302653892214013D-02   1.180098521565904D-01   2.100000000000000D+
   3    8     1.496348495303005D-02   9.611201567691633D-02   5.257340567840707D-
   4    9     7.900460841091139D-03   1.325041647391314D-02   1.000000000000000D+
   5   10     7.314495451266917D-03   1.510670810312237D-02   1.000000000000000D+
   6   11     6.750147275936680D-03   1.914964958023047D-02   1.000000000000000D+
   7   12     5.850716021108205D-03   1.028089194579363D-02   1.000000000000000D+
   8   13     5.778664230657791D-03   3.676866074530332D-04   1.000000000000000D+
   9   14     5.777818823650782D-03   3.010740179797255D-04   1.000000000000000D+

 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` [Variável]

Valor por omissão: 100

`lbfgs_ncorrections` [Variável]

Valor por omissão: 25

54 lindstedt

54.1 Definições para lindstedt

`Lindstedt` (*eq*,*pvar*,*torder*,*ic*) [Função]

Esse é um primeiro passo para um código de Lindstedt. Esse código pode resolver problemas com condições iniciais fornecidas, às quais podem ser constantes arbitrárias, (não apenas *%k1* e *%k2*) onde as condições iniciais sobre as equações de perturbação são $z[i] = 0, z'[i] = 0$ para $i > 0$. *ic* é a lista de condições iniciais.

Problemas ocorrem quando condições iniciais não forem dadas, como as constantes nas equações de perturbação são as mesmas que a solução da equação de ordem zero. Também, problemas ocorrem quando as condições iniciais para as equações de perturbação não são $z[i] = 0, z'[i] = 0$ para $i > 0$, tais como a equação de Van der Pol.

Exemplo:

```
(%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
```

Para usar essa função escreva primeiro `load("makeOrders")` e `load("lindstedt")`.

55 linearalgebra

55.1 Introdução a linearalgebra

`linearalgebra` é uma colecção de funções para álgebra linear.

Exemplo:

```
(%i1) load ("linearalgebra");
Warning - you are redefining the Maxima function require_list
Warning - you are redefining the Maxima function matrix_size
Warning - you are redefining the Maxima function rank
(%o1) /usr/local/share/maxima/5.9.2/share/linearalgebra/linearalgebra.mac
(%i2) M : matrix ([1, 2], [1, 2]);
                            [ 1  2 ]
(%o2)                       [      ]
                            [ 1  2 ]
(%i3) nullspace (M);
                                [  1  ]
                                [     ]
(%o3)                      span([   1 ])
                                [ - - ]
                                [   2 ]
(%i4) columnspace (M);
                                 [ 1 ]
(%o4)                       span([   ])
                                 [ 1 ]
(%i5) ptriangularize (M - z*ident(2), z);
                           [ 1   2 - z   ]
(%o5)                      [             ]
                           [           2 ]
                           [ 0  3 z - z  ]
(%i6) M : matrix ([1, 2, 3], [4, 5, 6], [7, 8, 9]) - z*ident(3);
                     [ 1 - z    2      3   ]
                     [                     ]
(%o6)                [   4    5 - z    6   ]
                     [                     ]
                     [   7      8    9 - z ]
(%i7) MM : ptriangularize (M, z);
               [ 4  5 - z            6           ]
               [                                 ]
               [               2                 ]
               [     66        z    102 z   132   ]
               [ 0   --      - -- + ----- + ---   ]
(%o7)          [     49        7     49     49    ]
               [                                 ]
               [               3        2        ]
               [           49 z    245 z    147 z ]
```

```
                [ 0    0    ----- - ------ - ----- ]
                [            264      88      44   ]
(%i8) algebraic : true;
(%o8)                           true
(%i9) tellrat (MM [3, 3]);
                           3       2
(%o9)                    [z  - 15 z  - 18 z]
(%i10) MM : ratsimp (MM);
                [ 4  5 - z           6          ]
                [                               ]
                [               2               ]
(%o10)          [     66      7 z  - 102 z - 132 ]
                [ 0   --    - ------------------ ]
                [     49             49         ]
                [                               ]
                [ 0    0            0           ]
(%i11) nullspace (MM);
                           [        1         ]
                           [                  ]
                           [   2              ]
                           [  z  - 14 z - 16  ]
                           [  --------------  ]
(%o11)               span([        8         ])
                           [                  ]
                           [    2             ]
                           [   z  - 18 z - 12 ]
                           [ - -------------- ]
                           [         12       ]
(%i12) 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  ]
(%o12)                    [                ]
                          [ 9   10  11  12 ]
                          [                ]
                          [ 13  14  15  16 ]
(%i13) columnspace (M);
                              [ 1  ]  [ 2  ]
                              [    ]  [    ]
                              [ 5  ]  [ 6  ]
(%o13)                  span([    ], [    ])
                              [ 9  ]  [ 10 ]
                              [    ]  [    ]
                              [ 13 ]  [ 14 ]
(%i14) apply ('orthogonal_complement, args (nullspace (transpose (M))));
                              [ 0 ]  [  1  ]
                              [   ]  [     ]
```

```
                              [ 1 ]  [  0  ]
(%o14)                  span([   ], [     ])
                              [ 2 ]  [ - 1 ]
                              [   ]  [     ]
                              [ 3 ]  [ - 2 ]
```

55.2 Definições para linearalgebra

`addmatrices` (*f*, *M_1*, ..., *M_n*) [Função]

Usando a função *f* como a função de adição, retorne a adição das matrizes *M_1*, ..., *M_n*. A função *f* deve aceitar qualquer número de argumentos (uma função enária do Maxima).

Exemplos:

```
(%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*) [Função]

Retorna `true` se e somente se *M* for uma matriz e toda entrada de *M* também for uma matriz.

`columnop` (*M*, *i*, *j*, *theta*) [Função]

Se *M* for uma matriz, retorna a matriz que resulta de fazer a operação de coluna `C_i <- C_i - theta * C_j`. Se *M* não tiver uma linha *i* ou *j*, emite uma mensagem de erro.

`columnswap` (*M*, *i*, *j*) [Função]

Se *M* for uma matriz, troca as colunas *i* e *j*. Se *M* não tiver uma coluna *i* ou *j*, emite uma mensagem de erro.

`columnspace` (*M*) [Função]

Se *M* for uma matriz, retorna `span (v_1, ..., v_n)`, onde o conjunto `{v_1, ..., v_n}` é uma base para o espaço coluna de *M*. A diferença entre o maior elemento e o menor elemento do conjunto vazio é `{0}`. Dessa forma, quando o espaço coluna tiver somente um membro, retorna `span ()`.

`copy` (*e*) [Função]

Retorna uma cópia da expressão *e* do Maxima. Embora *e* possa ser qualquer expressão do Maxima, Afunção `copy` é mais útil quando *e* for ou uma lista ou uma matriz; considere: load ("linearalgebra"); m : [1,[2,3]]$ mm : m$ mm[2][1] : x$ m; mm;

```
(%i1) load("linearalgebra")$
(%i2) m : [1,[2,3]]$
(%i3) mm : m$
(%i4) mm[2][1] : x$
(%i5) m;
```

```
(%o5) [1,[x,3]]
(%i6) mm;
(%o6) [1,[x,3]]
```

Vamos tentar a mesma experiência, mas dessa vez tomemos *mm* como sendo uma cópia de *m* m : [1,[2,3]]$ mm : copy(m)$ mm[2][1] : x$ m; mm;

```
(%i7) m : [1,[2,3]]$
(%i8) mm : copy(m)$
(%i9) mm[2][1] : x$
(%i10) m;
(%o10) [1,[2,3]]
(%i11) mm;
(%o11) [1,[x,3]]
```

Dessa vez, a atribuição a *mm* não muda o valor de *m*.

`cholesky` (*M*) [Função]
`cholesky` (*M*, *corpo*) [Função]

Retorna factorização de Cholesky da matriz hermitiana (or autoadjunta) *M*. O valor padrão para o segundo argumento é `generalring`. Para uma descrição dos possíveis valores para *corpo*, veja `lu_factor`.

`ctranspose` (*M*) [Função]

Retorna a matriz transposta conjugada complexa da matriz *M*. A função `ctranspose` usa `matrix_element_transpose` para transpor cada elemento da matriz.

`diag_matrix` (*d_1*, *d_2*,...,*d_n*) [Função]

Retorna uma matriz diagonal matriz com entradas de diagonal *d_1*, *d_2*,...,*d_n*. Quando as entradas de diagonal forem matrizes, as entradas zero da matriz retornada serão todas matrizes de tamanho apropriado; por exemplo:

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

(%i2) diag_matrix(diag_matrix(1,2),diag_matrix(3,4));

                              [ [ 1  0 ]  [ 0  0 ] ]
                              [ [      ]  [      ] ]
                              [ [ 0  2 ]  [ 0  0 ] ]
(%o2)                         [                    ]
                              [ [ 0  0 ]  [ 3  0 ] ]
                              [ [      ]  [      ] ]
                              [ [ 0  0 ]  [ 0  4 ] ]
(%i3) diag_matrix(p,q);

                                     [ p  0 ]
(%o3)                                [      ]
                                     [ 0  q ]
```

`dotproduct` (*u*, *v*) [Função]

Retorna o produto do ponto (produto escalar) dos vectores *u* e *v*. Isso é o mesmo que `conjugate (transpose (u)) . v`. Os argumentos *u* e *v* devem ser vectores coluna.

`eigens_by_jacobi` (*A*) [Função]
`eigens_by_jacobi` (*A*, *tipo_corpo*) [Função]

Calculam os autovalores e autovectores de *A* pelo método de rotações de Jacobi. *A* deve ser uma matriz simétrica (mas essa matriz simétrica precisa não ser nem definida positiva e nem semidefinida positiva). *tipo_corpo* indica o corpo computacional, pode ser ou `floatfield` ou `bigfloatfield`. Se *tipo_corpo* não for especificado, o padrão é `floatfield`.

Os elementos de *A* devem ser números ou expressões que avaliam para números via `float` ou `bfloat` (dependendo do valor de *tipo_corpo*).

Exemplos:

```
(%i1) load ("linearalgebra");
(%o1) /home/robert/tmp/maxima-head/maxima/share/linearalgebra/li\
nearalgebra.mac
(%i2) S : matrix ([1/sqrt(2), 1/sqrt(2)], [- 1/sqrt(2), 1/sqrt(2)]);
                      [     1         1    ]
                      [  -------   ------- ]
                      [  sqrt(2)   sqrt(2) ]
(%o2)                 [                    ]
                      [      1        1    ]
                      [ - -------  ------- ]
                      [   sqrt(2)  sqrt(2) ]
(%i3) L : matrix ([sqrt(3), 0], [0, sqrt(5)]);
                       [ sqrt(3)     0    ]
(%o3)                  [                  ]
                       [    0     sqrt(5) ]
(%i4) M : S . L . transpose (S);
             [ sqrt(5)   sqrt(3)  sqrt(5)   sqrt(3) ]
             [ ------- + -------  ------- - ------- ]
             [    2         2        2         2    ]
(%o4)        [                                      ]
             [ sqrt(5)   sqrt(3)  sqrt(5)   sqrt(3) ]
             [ ------- - -------  ------- + ------- ]
             [    2         2        2         2    ]
(%i5) 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
(%o5) [[1.732050807568877, 2.23606797749979],
                         [  0.70710678118655   0.70710678118655 ]
                         [                                      ]]
                         [ - 0.70710678118655  0.70710678118655 ]
(%i6) float ([[sqrt(3), sqrt(5)], S]);
(%o6) [[1.732050807568877, 2.23606797749979],
                         [  0.70710678118655   0.70710678118655 ]
                         [                                      ]]
```

```
                              [ - 0.70710678118655  0.70710678118655 ]
(%i7) 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
(%o7) [[1.732050807568877b0, 2.23606797749979b0],
                    [  7.071067811865475b-1   7.071067811865475b-1 ]
                    [                                              ]]
                    [ - 7.071067811865475b-1  7.071067811865475b-1 ]
```

`get_lu_factors` (*x*) [Função]

Quando *x* = `lu_factor (A)`, então `get_lu_factors` retorna uma lista da forma `[P, L, U]`, onde P é uma matriz de permutação, L é triangular baixa com a diagonal preenchida com a unidade, e U é triangular alta, e *A = P L U*.

`hankel` (*col*) [Função]
`hankel` (*col*, *lin*) [Função]

Retorna uma matriz de Hankel *H*. A primeira coluna de *H* é *col*; excepto para a primeira entrada, a última linha de *H* é *lin*. O valor padrão para *lin* é o vector nulo com o mesmo comprimento que *col*.

`hessian` (*f*,*vars*) [Função]

Retorna a matriz hessiana de *f* com relação às variáveis na lista *vars*. As entradas *i*,*j* da matriz hessiana são *diff(f vars[i],1,vars[j],1)*.

`hilbert_matrix` (*n*) [Função]

Retorna the *n* by *n* matriz de Hilbert. Quando *n* não for um inteiro positivo, emite uma mensagem de erro.

`identfor` (*M*) [Função]
`identfor` (*M*, *corpo*) [Função]

Retorna uma matriz identidade que tem o mesmo tamanho que a matriz *M*. As entradas de diagonal da matriz identidade são a identidade multiplicativa do corpo *corpo*; o padrão para *corpo* é *generalring*.

O primeiro argumento *M* pode ser uma matriz quadrada ou um não matriz. Quando *M* for uma matriz, cada entrada de *M* pode ser uma matriz quadrada – dessa forma *M* pode ser uma matriz de bloco do Maxima. A matriz pode ser de bloco para qualquer (finita) quantidade de níveis.

Veja também `zerofor`

`invert_by_lu` (*M*, (*rng generalring*)) [Função]

Inverte a matriz *M* através de factorização linear alta (LU). A factorização LU é concluída usando o anel *rng*.

`kronecker_product` (*A*, *B*) [Função]

Retorna o produto de Kronecker das matrizes *A* e *B*.

`listp` (*e*, *p*) [Função]
`listp` (*e*) [Função]

Recebendo um argumento opcional *p*, retorna `true` se *e* for uma lista do Maxima e *p* avalia para `true` para elemento da lista. Quando `listp` não recebe o argumento opcional, retorna `true` se *e* for uma lista do Maxima. em todos os outros casos, retorna `false`.

`locate_matrix_entry` (*M*, *r_1*, *c_1*, *r_2*, *c_2*, *f*, *rel*) [Função]

O primeiro argumento deve ser uma matriz; os argumentos que vão de *r_1* até *c_2* determinam um sub-matriz de *M* que consiste de linhas que vão de *r_1* até *r_2* e colunas que vão de *c_1* até *c_2*.

Encontra uma entrada na sub-matriz *M* que satisfaz alguma propriedade. Existem três casos:

(1) *rel* = `'bool` e *f* um predicado:

Examina a sub-matriz da esquerda para a direita e de cima para baixo, e retorna o índice da primeira entrada que satisfizer o predicado *f*. Se nenhuma entrada da matriz satisfizer o predicado *f*, retorna `false`.

(2) *rel* = `'max` e *f* avaliar para um número real:

Examina a sub-matriz procurando por uma entrada que maximize *f*. Retorna retorna o índice da entrada maximizada.

(3) *rel* = `'min` e *f* avaliar para um número real:

Examina a sub-matriz procurando por uma entrada que minimize *f*. Retorna o índice de uma entrada minimizada.

`lu_backsub` (*M*, *b*) [Função]

Quando *M* = `lu_factor (A, corpo)`, então `lu_backsub (M, b)` resolve o sistema linear *A x* = *b*.

`lu_factor` (*M*, *corpo*) [Função]

Retorna uma lista da forma `[`*LU*`,` *perm*`,` *corpo*`]`, ou da forma `[`*LU*`,` *perm*`,` *cmp*`,` *baixo-cnd* *alto-cnd*`]`, onde

(1) A matriz *LU* contéa factorização de *M* na forma enpacotada. Forma empacotada significa três coisas: Primeiro, as linhas de *LU* são permutadas confirme a lista *perm*. Se, por exemplo, *perm* for a lista list `[3,2,1]`, a primeira linha actual da factorização *LU* será a terceira linha da matriz *LU*. Segundo, o factor triangular baixo de m é a parte triangular baixa de *LU* com as entradas de diagonal todas substituídas pela unidade. Terceiro, o factor triangular alto de *M* é a parte triangular alta de *LU*.

(2) Quando o corpo for ou `floatfield` ou `complexfield`, os números *baixo-cnd* e *alto-cnd* serão associados baixo e alto para o número condicional de norma infinita de *M*. Para todos os corpos (fields), o número condicional de norma infinita não pode ser estimado; para tais corpos, `lu_factor` retorna uma lista com dois itens. Ambos o baixo e o alto associado podem diferir de seus verdadeiros valores de factores arbitráriamente grandes. (Veja também `mat_cond`.)

O argumento *M* deve ser a matriz quadrada.

O argumento opcional *cmp* deve ser um símbolo que determine um anel ou corpo. Os corpos e anéis predefinidos são:

(a) `generalring` – o anel de expressões do Maxima, (b) `floatfield` – o corpo dos números em ponto flutuante do tipo de precisão dupla, (c) `complexfield` – o corpo dos números complexos em ponto flutuante do tipo de precisão dupla, (d) `crering` – o anel das expressões racionais canónicas (CRE) do Maxima, (e) `rationalfield` – o corpo dos números racionais, (f) `runningerror` – rastro de todos os erros de arredondamento de números em ponto flutuante, (g) `noncommutingring` – o anel de expressões do Maxima onde multiplicação for o operador ponto não comutativo.

Quando o corpo for `floatfield`, `complexfield`, ou `runningerror`, o algoritmo usa pivotagem parcial; para todos os outros corpos, linhas são comutadas somente quando necessário para evitar um pivô nulo.

A adição aritmética em ponto flutuante não é associativa, então o significado de 'corpo' difere da definição matemática.

Um membro do corpo `runningerror` é uma lista do Máxima de dois membros da forma `[x,n]`,onde x é um número em onto flutuante e `n` é um inteiro. A diferença relativa entre o valor de 'verdadeiro' de `x` e `x` é aproximadamente associado pelo épsilon da máquina vezes `n`. O erro de execução associado arrasta alguns termos da ordem do quadrado do épsilon da máquina.

Não existe interface de utilizador definida um novo anel. Um utilizador que estiver familiazrizado com o Lisp Comum está apto para definir um novo corpo. Para fazer isso, um utilizador deve definir funções para as operações aritméticas e funções para conversão para a representação de corpo do Máxima e vice-versa. Adicionalmente, para corpos ordenados (onde a pivotagem parcial será usada), um uduário deve definir funções para módulo e para comparar membros do corpo. Após isso tudo que resta é definir uma estrutura de Lisp Comum `mring`. O ficheiro `mring` tem muitos exemplos.

Para calcular a factorização, a primeira tarefa é converter cada entrada de matriz para um elemento do corpo indicado. Quando a cnversão não for possível, a factorização encerra com uma mensagem de erro. Elementos do corpo não precisam ser expressões do Maxima. Elementos do `complexfield`, por exemplo, são números complexos do Lisp Comum. Dessa forma após calcular a factorização, como entradas da matriz devem ser convertidas para expressões do Maxima.

Veja também `get_lu_factors`.

Exemplos:

```
(%i1) load ("linearalgebra");
Warning - you are redefining the Maxima function require_list
Warning - you are redefining the Maxima function matrix_size
Warning - you are redefining the Maxima function rank
(%o1) /usr/local/share/maxima/5.9.2/share/linearalgebra/linearalgebra.mac
(%i2) w[i,j] := random (1.0) + %i * random (1.0);
(%o2)          w     := random(1.) + %i random(1.)
                i, j
(%i3) showtime : true$
Evaluation took 0.00 seconds (0.00 elapsed)
(%i4) M : genmatrix (w, 100, 100)$
Evaluation took 7.40 seconds (8.23 elapsed)
(%i5) lu_factor (M, complexfield)$
```

```
Evaluation took 28.71 seconds (35.00 elapsed)
(%i6) lu_factor (M, generalring)$
Evaluation took 109.24 seconds (152.10 elapsed)
(%i7) showtime : false$

(%i8) M : matrix ([1 - z, 3], [3, 8 - z]);
                         [ 1 - z    3   ]
(%o8)                    [              ]
                         [   3    8 - z ]
(%i9) lu_factor (M, generalring);
                [ 1 - z         3        ]
                [                        ]
(%o9)          [[   3            9       ], [1, 2]]
                [ -----  - z - ----- + 8 ]
                [ 1 - z        1 - z     ]
(%i10) get_lu_factors (%);
                    [   1    0 ]  [ 1 - z         3        ]
          [ 1  0 ]  [          ]  [                        ]
(%o10) [[       ], [   3      ], [                9       ]]
          [ 0  1 ]  [ -----  1 ]  [   0    - z - ----- + 8 ]
                    [ 1 - z    ]  [             1 - z     ]
(%i11) %[1] . %[2] . %[3];
                         [ 1 - z    3   ]
(%o11)                   [              ]
                         [   3    8 - z ]
```

`mat_cond` (*M*, *1*) [Função]
`mat_cond` (*M*, *inf*) [Função]

Retorna o número condiciona da norma de ordem *p* da matriz *m*. Os valores permitidos para *p* são 1 e *inf*. Essa função utiliza a factorização linear alta para inverter a matriz *m*. Dessa forma o tempode execução para `mat_cond` é proporcional ao cubo do tamanho da matriz; `lu_factor` determina as associaçãoes baixa e alta para o número de condição de norma infinita em tempo proporcional ao quadrado do tamanho da matriz.

`mat_norm` (*M*, *1*) [Função]
`mat_norm` (*M*, *inf*) [Função]
`mat_norm` (*M*, *frobenius*) [Função]

Retorna a matriz de norma *p* da matriz *M*. Os valores permitidos para *p* são 1, `inf`, e `frobenius` (a norma da matriz de Frobenius). A matriz *M* pode ser uma matriz não de bloco.

`matrixp` (*e*, *p*) [Função]
`matrixp` (*e*) [Função]

Fornecendo um argumento opcional *p*, `matrixp` retorna `true` se *e* for uma matriz e *p* avaliar para `true` para todo elemento da matriz. Quando a `matrixp` não for fornecido umargumento opcional, retorna `true` se `e` for uma matriz. em todos os outros casos, retorna `false`.

Veja também `blockmatrixp`

`matrix_size` (*M*) [Função]
Retorna uma lista com dois elementos que fornecem o número de linhas e colunas, respectivamente da matriz *M*.

`mat_fullunblocker` (*M*) [Função]
Se *M* for uma matriz de bloco, expande todos os blocos da matriz em todos os níveis. Se *M* for uma matriz, retorna *M*; de outra forma, emite uma mensagem de erro.

`mat_trace` (*M*) [Função]
Retorna o traço da matriz *M*. Se *M* não for uma matriz, retorna uma forma substantiva. Quando *M* for uma matriz de bloco, `mat_trace(M)` retorna o mesmo valor retornado por `mat_trace(mat_unblocker(m))`.

`mat_unblocker` (*M*) [Função]
Se *M* for uma matriz de bloco, `mat_unbloker` desfaz o bloco de *M* um nível. Se *M* for uma matriz, `mat_unblocker (M)` retorna *M*; de outra forma, emite uma mensagem de erro.

Dessa forma se cada entrada de *M* for matriz, `mat_unblocker (M)` retorna uma matriz "desblocada", mas se cada entrada de *M* for uma matriz de bloco, `mat_unblocker (M)` retorna uma matriz de bloco com um nível de bloco a menos.

Se usar matrizes de bloco, muito provavelmente irá querer escolher `matrix_element_mult` para `"."` e `matrix_element_transpose` para `'transpose`. Veja também `mat_fullunblocker`.

Exemplo:

```
(%i1) load ("linearalgebra");
Warning - you are redefining the Maxima function require_list
Warning - you are redefining the Maxima function matrix_size
Warning - you are redefining the Maxima function rank
(%o1) /usr/local/share/maxima/5.9.2/share/linearalgebra/linearalgebra.mac
(%i2) A : matrix ([1, 2], [3, 4]);
                              [ 1  2 ]
(%o2)                         [      ]
                              [ 3  4 ]
(%i3) B : matrix ([7, 8], [9, 10]);
                              [ 7  8  ]
(%o3)                         [       ]
                              [ 9  10 ]
(%i4) matrix ([A, B]);
                        [ [ 1  2 ]  [ 7  8  ] ]
(%o4)                   [ [      ]  [       ] ]
                        [ [ 3  4 ]  [ 9  10 ] ]
(%i5) mat_unblocker (%);
                            [ 1  2  7  8  ]
(%o5)                       [             ]
                            [ 3  4  9  10 ]
```

`nonnegintegerp` (*n*) [Função]
Retorna `true` se e somente se *n* `>= 0` e *n* for um inteiro.

`nullspace` (*M*) [Função]
Se *M* for uma matriz, retorna `span (v_1, ..., v_n)`, onde o conjunto `{v_1, ..., v_n}` é uma base para o espaço nulo de *M*. A diferença entre o maior elemento e o menor elemento do conjunto vazio é `{0}`. Dessa forma, quando o espaço nulo tiver somente um membro, retorna `span ()`.

`nullity` (*M*) [Função]
Se *M* for uma matriz, retorna a dimensão do espaço nulo de *M*.

`orthogonal_complement` (*v_1*, ..., *v_n*) [Função]
Retorna `span (u_1, ..., u_m)`, onde o conjunto `{u_1, ..., u_m}` é uma base para o complemento ortogonal do conjunto `(v_1, ..., v_n)`.

Cada vector no intervalo de *v_1* até *v_n* deve ser um vector coluna.

`polynomialp` (*p*, *L*, *coeffp*, *exponp*) [Função]
`polynomialp` (*p*, *L*, *coeffp*) [Função]
`polynomialp` (*p*, *L*) [Função]
Retorna `true` se *p* for um polinómio nas variáveis da lista *L*, O predicado *coeffp* deve avaliar para `true` para cada coeficiente, e o predicado *exponp* deve avaliar para `true` para todos os expoentes das variáveis na lista *L*. Se quiser usar um valor personalizado para *exponp*, deverá fornecer *coeffp* com um valor mesmo se quiser o valor padrão para *coeffp*.

`polynomialp (p, L, coeffp)` é equivalente a `polynomialp (p, L, coeffp, 'nonnegintegerp)`.

`polynomialp (p, L)` é equivalente a `polynomialp (p, L, 'constantp, 'nonnegintegerp)`.

O polinómio não precisa ser expandido:

```
(%i1) load ("linearalgebra");
Warning - you are redefining the Maxima function require_list
Warning - you are redefining the Maxima function matrix_size
Warning - you are redefining the Maxima function rank
(%o1) /usr/local/share/maxima/5.9.2/share/linearalgebra/linearalgebra.mac
(%i2) polynomialp ((x + 1)*(x + 2), [x]);
(%o2)                         true
(%i3) polynomialp ((x + 1)*(x + 2)^a, [x]);
(%o3)                         false
```

Um exemplo usando um valor personalizado para `coeffp` e para `exponp`:

```
(%i1) load ("linearalgebra");
Warning - you are redefining the Maxima function require_list
Warning - you are redefining the Maxima function matrix_size
Warning - you are redefining the Maxima function rank
(%o1) /usr/local/share/maxima/5.9.2/share/linearalgebra/linearalgebra.mac
(%i2) polynomialp ((x + 1)*(x + 2)^(3/2), [x], numberp, numberp);
(%o2)                         true
```

```
(%i3) polynomialp ((x^(1/2) + 1)*(x + 2)^(3/2), [x], numberp, numberp);
(%o3)                          true
```

Polinómios com duas variáveis:

```
(%i1) load ("linearalgebra");
Warning - you are redefining the Maxima function require_list
Warning - you are redefining the Maxima function matrix_size
Warning - you are redefining the Maxima function rank
(%o1) /usr/local/share/maxima/5.9.2/share/linearalgebra/linearalgebra.mac
(%i2) polynomialp (x^2 + 5*x*y + y^2, [x]);
(%o2)                          false
(%i3) polynomialp (x^2 + 5*x*y + y^2, [x, y]);
(%o3)                          true
```

`polytocompanion` (*p*, *x*) [Função]

Se *p* for um polinómio em *x*, retorna a atriz companheira de *p*. Para um polinómio mônico *p* de grau *n*, temos `p = (-1)^n charpoly (polytocompanion (p, x))`.

Quando *p* não for um polinómio em *x*, emite uma mensagem de erro.

`ptriangularize` (*M*, *v*) [Função]

Se *M* for uma matriz onde cada entrada dessa matriz for um polinómio em *v*, retorna a matriz *M2* tal que

(1) *M2* é triangular alta,

(2) `M2 = E_n ... E_1 M`, onde os elemetnos de *E_1* a *E_n* são matrizes elementares cujas entrada são polinómios em *v*,

(3) `|det (M)| = |det (M2)|`,

Nota: Essa função não verifica se toda entrada é um polinómio em *v*.

`rowop` (*M*, *i*, *j*, *theta*) [Função]

Se *M* for uma matriz, retorna a matriz que resulta de se fazer a operação de linha `R_i <- R_i - theta * R_j`. Se *M* não tiver uma linha *i* ou *j*, emite uma mensagem de erro.

`rank` (*M*) [Função]

Retorna o ranque daquela matriz *M*. O rank é a dimensão do espaço coluna. Exemplo:

```
(%i1) load ("linearalgebra")$
WARNING: DEFUN/DEFMACRO: redefining function $COPY in
         /share/maxima/5.11.0/share/linearalgebra/linalg-utilities.lisp,
         was defined in
         /maxima-5.11.0/src/binary-clisp/comm2.fas
(%i2) rank(matrix([1,2],[2,4]));
(%o2)                                  1
(%i3) rank(matrix([1,b],[c,d]));
Proviso:  {d - b c # 0}
(%o3)                                  2
```

`rowswap` (*M*, *i*, *j*) [Função]

Se *M* for uma matriz, permuta as linha *i* e *j*. Se *M* não tiver uma linha *i* ou *j*, emite uma mensagem de erro.

`toeplitz` (*col*) [Função]
`toeplitz` (*col*, *lin*) [Função]

Retorna uma matriz de Toeplitz T. a primeira coluna de T é *col*; excepto para a primeira entrada, a primeira linha de T é *lin*. O padrão para *lin* é o conjugado complexo de *col*. Exemplo:

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

(%i2)  toeplitz([1,2,3],[x,y,z]);

                                  [ 1  y  z ]
                                  [         ]
(%o2)                             [ 2  1  y ]
                                  [         ]
                                  [ 3  2  1 ]
(%i3)  toeplitz([1,1+%i]);

                                [   1     1 - %I ]
(%o3)                           [                ]
                                [ %I + 1    1    ]
```

`vandermonde_matrix` ([*x_1*, ..., *x_n*]) [Função]

Retorna uma matriz n por n cuja i-ésima linha é `[1, x_i, x_i^2, ... x_i^(n-1)]`.

`zerofor` (*M*) [Função]
`zerofor` (*M*, *fld*) [Função]

Retorna uma matriz zero que tem o mesmo tamanho da matriz M. Toda entrada da matriz zero é a identidade aditiva do anel *fld*; o valor padrão para *fld* é *generalring*.

O primeiro argumento M pode ser uma matriz quadrada ou uma não matriz. Quando M for uma matriz, cada entrada de M pode ser uma matriz quadrada – dessa forma M pode ser uma matriz de bloco do Maxima. A matriz pode ser de bloco para qualquer nível (finito).

Veja também `identfor`

`zeromatrixp` (*M*) [Função]

Se M não for uma matriz de bloco, retorna `true` se `is (equal (e, 0))` for verdadeiro para cada elemento e da matriz M. Se M for uma matriz de bloco, retorna `true` se `zeromatrixp` avaliar para `true` para cada elemento de e.

56 lsquares

/lsquares.texi/1.1/Mon Feb 27 22:09:17 2006//

56.1 Definições para lsquares

`DETCOEF` [Variável global]

Essa variável é usada pelas funções `lsquares` e `plsquares` para armazenar o Coeficiente de Determinação que mede o melhor ajuste. Esse intervalo vai de 0 (nenhuma correlação) a 1 (correlação exacta).

Quando `plsquares` for chamada com uma lista de variáveis independentes, *DETCOEF* é escolhida para uma lista de Coeficientes de Determinação. Veja `plsquares` para detalhes.

Veja também `lsquares`.

`lsquares` (*Mat*,*VarList*,*equação*,*ParamList*) [Função]

`lsquares` (*Mat*,*VarList*,*equação*,*ParamList*,*EsperadosList*) [Função]

Ajuste múltiplo de equações não lineares de uma tabela de dados pelo método dos "mínimos quadrados". *Mat* é uma matriz contendo os dados, *VarList* é uma lista de nomes de variáveis (um para cada coluna de *Mat*), *equação* é a equação a ser ajustada (essa equação deve estar na forma: `depvar=f(indepvari,..., paramj,...)`, `g(depvar)=f(indepvari,..., paramj,...)` ou na forma `g(depvar, paramk,...)=f(indepvari,..., paramj,...)`), *ParamList* é a lista de parâmetros para obter, e *EsperadosList* é uma lista opcional de aproximações iniciais para os parâmetros; quando esse último argumento estiver presente, `mnewton` é usado em lugar de `solve` com o objectivo de pegar os parâmetros.

A equação pode ser completamente não linear com relação às variáveis independentes e à variável dependente. Com o objectivo de usar `solve()`, as equações devem ser lineares ou polinomiais com relação aos parâmetros. Equações como `y=a*b^x+c` podem ser ajustadas para `[a,b,c]` com `solve` se os valores de `x` forem inteiros positivos pequenos e existirem poucos dados (veja o exemplo em lsquares.dem). `mnewton` permite ajustar uma equação não linear com relação aos parâmetros, mas um bom conjunto de aproximações iniciais deve ser fornecido.

Se possível, a equação ajustada é retornada. Se existir mais de uma solução, uma lista de equações é retornada. O Coeficiente de Determinação é mostrado para informar sobre o melhor ajuste, de 0 (nenhuma correlação) a 1 (correlação exacta). Esse valor é também armazenado na variável global *DETCOEF*.

Exemplos usando `solve`:

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

(%i2) lsquares(matrix([1,2,0],[3,5,4],[4,7,9],[5,8,10]),
                [x,y,z], z=a*x*y+b*x+c*y+d, [a,b,c,d]);
      Determination Coefficient = 1.0
                        x y + 23 y - 29 x - 19
(%o2)              z = ----------------------
                                 6
```

```
(%i3) lsquares(matrix([0,0],[1,0],[2,0],[3,8],[4,44]),
               [n,p], p=a4*n^4+a3*n^3+a2*n^2+a1*n+a0,
          [a0,a1,a2,a3,a4]);
      Determination Coefficient = 1.0
                      4       3      2
                   3 n  - 10 n  + 9 n  - 2 n
(%o3)          p = ------------------------
                               6
(%i4) lsquares(matrix([1,7],[2,13],[3,25]),
               [x,y], (y+c)^2=a*x+b, [a,b,c]);
      Determination Coefficient = 1.0
(%o4) [y = 28 - sqrt(657 - 216 x),
                                  y = sqrt(657 - 216 x) + 28]
(%i5) lsquares(matrix([1,7],[2,13],[3,25],[4,49]),
               [x,y], y=a*b^x+c, [a,b,c]);
      Determination Coefficient = 1.0
                                 x
(%o5)                   y = 3 2  + 1
```

Exemplos usando `mnewton`:

```
(%i6) load("lsquares")$

(%i7) lsquares(matrix([1.1,7.1],[2.1,13.1],[3.1,25.1],[4.1,49.1]),
               [x,y], y=a*b^x+c, [a,b,c], [5,5,5]);
                                                  x
(%o7) y = 2.799098974610482 1.999999999999991
                                           + 1.099999999999874
(%i8) lsquares(matrix([1.1,4.1],[4.1,7.1],[9.1,10.1],[16.1,13.1]),
               [x,y], y=a*x^b+c, [a,b,c], [4,1,2]);
                               .4878659755898127
(%o8) y = 3.177315891123101 x
                                           + .7723843491402264
(%i9) lsquares(matrix([0,2,4],[3,3,5],[8,6,6]),
              [m,n,y], y=(A*m+B*n)^(1/3)+C, [A,B,C], [3,3,3]);
                                                        1/3
(%o9) y = (3.999999999999862 n + 4.999999999999359 m)
                                            + 2.00000000000012
```

Para usar essa função escreva primeiro `load("lsquares")`. Veja também `DETCOEF` e `mnewton`.

`plsquares` (*Mat*,*VarList*,*depvars*) [Função]
`plsquares` (*Mat*,*VarList*,*depvars*,*maxexpon*) [Função]
`plsquares` (*Mat*,*VarList*,*depvars*,*maxexpon*,*maxdegree*) [Função]

Ajuste de polinómios de várias variáveis de uma tabela de dados pelo método dos "mínimos quadrados". *Mat* é uma matriz contendo os dados, *VarList* é uma lista de nomes de variáveis (um nome para cada coluna de Mat, mas use `"-"` em lugar de nomes de variáveis para colunas de Mat), *depvars* é o nome de uma variável dependente ou uma lista com um ou mais nomes de variáveis dependentes (cujos

nomes podem estar em *VarList*), *maxexpon* é o expoente máximo opcional para cada variável independente (1 por padrão), e *maxdegree* é o argumento opcional grau máximo do polinómio (*maxexpon* por padrão); note que a soma dos expoentes de cada termo deve ser menor ou igual a *maxdegree*, e se `maxdgree = 0` então nenhum limite é aplicado.

Se *depvars* é o nome de uma variável dependente (fora de uma lista), `plsquares` retorna o polinómio ajustado. Se *depvars* for uma lista de uma ou mais variáveis dependentes, `plsquares` retorna uma lista com o(s) polinómio(s) ajustado(s). Os Coeficientes de Determinação são mostrados com o objectivo de informar sobre o melhor do ajuste, cujo intervalo vai de 0 (nenhuma correlação) a 1 (correlação exacta). Esses valores são também armazenados na variável global *DETCOEF* (uma lista se *depvars* for tambńuma lista).

Um simples exemplo de ajuste linear de várias variáveis:

```
(%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
```

O mesmo exemplo sem restrições de grau:

```
(%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
```

Quantas diagonais possui um polígono de N lados? Que grau polinomial deverá ser usado?

```
(%i4) plsquares(matrix([3,0],[4,2],[5,5],[6,9],[7,14],[8,20]),
                [N,diagonais],diagonais,5);
     Determination Coefficient for diagonais = 1.0
                                  2
                                 N  - 3 N
(%o4)                diagonais = --------
                                    2
(%i5) ev(%, N=9);   /* Testando para um polígono de 9 lados - o eneágono */
(%o5)                  diagonals = 27
```

De quantas formas dispomos para colocar duas raínhas sem que elas estejam ameaçadas num tabuleiro de xadrez n x n ?

```
(%i6) plsquares(matrix([0,0],[1,0],[2,0],[3,8],[4,44]),
                [n,posicoes],[posicoes],4);
     Determination Coefficient for [posicoes] = [1.0]
                          4       3      2
```

```
                          3 n  - 10 n  + 9 n  - 2 n
(%o6)    [posicoes  = -------------------------]
                                  6
(%i7) ev(%[1], n=8); /* Testando para um tabuleiro de (8 x 8) */
(%o7)                posicoes = 1288
```

Um exemplo com seis variáveis dependentes:

```
(%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]
```

Para usar essa função escreva primeiramente `load("lsquares")`.

57 makeOrders

57.1 Definições para makeOrders

`makeOrders` (*indvarlist*,*orderlist*) [Função]

Retorna uma lista de todos os expoentes para um polinómio acima de e incluindo os argumentos.

```
(%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
```

onde `[0, 1]` está associado ao termo b e `[2, 3]` está associado ao termo a^2b^3.

Para usar essa função escreva primeiro `load("makeOrders")`.

58 mnewton

58.1 Definições para mnewton

`newtonepsilon` [Variável de opção]

Valor por omissão: `10.0^(-fpprec/2)`

Precisão para determinar quando a função `mnewton` convergiu em direção à solução.

Veja também `mnewton`.

`newtonmaxiter` [Variável de opção]

Valor por omissão: `50`

Número máximo de iterações que para a função `mnewton` caso essa função não seja convergente ou se convergir muito lentamente.

Veja também `mnewton`.

`mnewton` (*FuncList*,*VarList*,*GuessList*) [Função]

Solução de multiplas funções não lineares usando o método de Newton. *FuncList* é a lista de funções a serem resolvidas, *VarList* é a lista dos nomes de variáveis, e *GuessList* é a lista de aproximações iniciais.

A solução é retornada no mesmo formato retornado pela função `solve()`. Caso a solução não seja encontrada, `[]` é retornado.

Essa função é controlada através das variáveis globais `newtonepsilon` e `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]]
```

Para usar essa função primeiro escreva `load("mnewton")`. Veja também `newtonepsilon` e `newtonmaxiter`.

59 numericalio

59.1 Introdução a numericalio

`numericalio` é uma colecção de funções para ler e escrever ficheiros de dados. O ficheiro é lido completamente para construir um objecto; leituras parciais não são suportadas.

É assumido que cada item a ler ou escrever é atômico: um número inteiro, número em ponto flutuante, grande número em ponto flutuante, sequência de caracteres, ou símbolo, e não um número racional ou um número complexo ou qualquer outro tipo de expressão não atômica. Essas funções podem tentar fazer alguma coisa levemente parecida com expressões não atômicas, mas os resultados não são especificados aqui e são sujeitos a mudanças.

Átomos em ambos os ficheiros de entrada e saída possuem o mesmo formato que em ficheiros de lote do Maxima ou no console interativo. Em particular, sequência de caracteres são contidas dentro de aspas duplas, contrabarra `\` evita qualquer interpretação especial do caractere seguinte, e o ponto de interrogação `?` é reconhecido no início de um símbolo para significar um símbolo do Lisp (em oposição a um símbolo do Maxima). Nenhum caractere de continuação (para continuar linhas quebradas) é reconhecido.

separator_flag diz que caracteres separa elementos. *separator_flag* é um argumento opcional para todas as funções de leitura e escrita.

Para entrada, os valores de *separator_flag* reconhecidos são: `comma` para valores separados por vírgula, `pipe` para valores separados pelo caractere barra vertical `|`, `semicolon` para valores separados por ponto e vírgula `;`, e `space` para valores separados pelos caracteres de espaço e de tabulação. Se o nome do ficheiro a ser lido/escrito termina em `.csv` e *separator_flag* não for especificado, `comma` é assumido. Se o nome do ficheiro termina em alguma outra coisa que não `.csv` e `separator_flag` não for especificado, `space` é assumido.

Para saída, os mesmos quatro sinalizadores são reconhecidos como na entrada, e também `tab`, para valores separados pelo caractere de tabulaçao.

Em entrada, múltiplos espaços e múltiplas tabulações sucessivas contam como um separador simples. Todavia, múltiplas vírgulas, barras verticais, ou ponto-e-vírgulas são significativos. Sucessivas vírgulas, barras verticais, ou ponto-e-vírgulas (com ou sem intercalação de espaços ou tabulações) são considerados como tendo `false` entre os separadores. Por exemplo, `1234,,Foo` é tratado da mesma forma que `1234,false,Foo`. Em saídas, os átomos `false` são escritos como tais; uma lista `[1234, false, Foo]` é escrita `1234,false,Foo`, e não é tentado colapsar a saída para `1234,,Foo`.

59.2 Definições para numericalio

`read_matrix` (*nomeficheiro*) [Função]
`read_matrix` (*nomeficheiro*, *separator_flag*) [Função]

Lê o ficheiro *nomeficheiro* e retorna seu conteúdo completo como uma matriz. Se *separator_flag* não for especificado, o ficheiro é assumido como delimitado por espaços em branco.

`read_matrix` infere o tamanho da matriz dos dados de entrada. Cada linha do ficheiro inicia uma linha da matriz. Se algumas linhas possuirem diferentes comprimentos, `read_matrix` reclama.

`read_lisp_array` (*nomeficheiro*, *A*) [Função]
`read_lisp_array` (*nomeficheiro*, *A*, *separator_flag*) [Função]

`read_lisp_array` exige que o array seja declarado através de `make_array` antes de chamar a função de leitura. (Isso obviamente é necessário para inferir a dimensão do array, que pode ser um problema para arrays com múltiplas dimensões.)

`read_lisp_array` não verifica para ver se o ficheiro de entrada está de acordo com as dimensãoes do array; a entrada é lida como uma lista monótona, então o array é preenchido usando `fillarray`.

`read_maxima_array` (*nomeficheiro*, *A*) [Função]
`read_maxima_array` (*nomeficheiro*, *A*, *separator_flag*) [Função]

`read_maxima_array` requer que o array seja declarado através de `array` antes de chamar a função de leitura. (Isso obviamente é necessário para inferir a dimensão do array, que pode ser uma hassle para arrays com múltiplas dimensões.)

`read_maxima_array` não verifica para ver se o ficheiro de entrada está de acordo com as dimensãoes do array; a entrada é lida como uma lista monótona, então o array é preenchido usando `fillarray`.

`read_hashed_array` (*nomeficheiro*, *A*) [Função]
`read_hashed_array` (*nomeficheiro*, *A*, *separator_flag*) [Função]

`read_hashed_array` trata o primeiro item sobre uma linha como uma chave hash, e associa o restante da linha (como uma lista) com a chava. Por exemplo, a linha `567 12 17 32 55` é equivalente a `A[567]: [12, 17, 32, 55]$`. Linhas não precisam ter o mesmo número de elementos.

`read_nested_list` (*nomeficheiro*) [Função]
`read_nested_list` (*nomeficheiro*, *separator_flag*) [Função]

`read_nested_list` retorna uma lista que tem uma sublista para cada linha de entrada. Linhas não precisam ter o mesmo número de elementos. Linhas vazias *não* são ignoradas: uma linha vazia retorna uma sublista vazia.

`read_list` (*nomeficheiro*) [Função]
`read_list` (*nomeficheiro*, *separator_flag*) [Função]

`read_list` lê todas as entradas em uma lista monótona. `read_list` ignora o caractere de fim de linha.

`write_data` (*X*, *nomeficheiro*) [Função]
`write_data` (*object*, *nomeficheiro*, *separator_flag*) [Função]

`write_data` escreve o objecto *X* no ficheiro *nomeficheiro*.

`write_data` escreve matrizes da forma usual, com uma linha por fileira.

`write_data` escreve arrays declarados do Lisp e do Maxima da forma usual, com um caractere de nova linha no final de todo pedaço. Pedaços dimensionais muito grandes são separados por meio de novas linhas adicionais.

`write_data` escreve arrays desordenados com uma chave seguida por a lista associada sobre cada linha.

`write_data` escreve a lista seguinte com cada sublista em uma linha.

`write_data` escreve uma lista monótona toda em uma linha.

Se `write_data` anexa ao final ou abandona os excessos em seus ficheiros de saída é governado através da variável global `file_output_append`.

60 opsubst

60.1 Definições para opsubst

`opsubst` (*f*,*g*,*e*) [Função]
`opsubst` (*g*=*f*,*e*) [Função]
`opsubst` ([*g1*=*f1*,*g2*=*f2*,..., *gn*=*fn*],*e*) [Função]

A função `opsubst` similar à função `subst`, excepto que `opsubst` somente faz substituições para as operações em uma expressões. Em geral, quando f for um operador em uma expressão e, substitui g por f na expressão e.

Para determinar o operador, `opsubst` escolhe `inflag` para verdadeiro (true). Isso significa que `opsubst` substitui para a forma de operador interna, não para a mostrada, na expressão.

Exemplos:

```
(%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)
```

Internamente, Maxima não usa os operadores de negação unária, divisão, ou de subtração; dessa forma:

```
(%i8) opsubst("+","-",a-b);
(%o8)                      a - b
(%i9) opsubst("f","-",-a);
(%o9)                       - a
(%i10) opsubst("^^","/",a/b);
                             a
(%o10)                       -
                             b
```

A representação interna de -a*b é *(-1,a,b); dessa forma

```
(%i11) opsubst("[","*", -a*b);
(%o11)                   [- 1, a, b]
```

Quando o operador não for um símbolo Maxima, geralmente alguma outra função sinalizará um erro:

```
(%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);
```

Todavia, operadores subscritos são permitidos:

```
(%i13) opsubst(g[5],f, f(x));
(%o13)                      g (x)
                             5
```

Para usar essa função escreva primeiramente `load("opsubst")`.

61 orthopoly

61.1 Introdução a polinómios ortogonais

`orthopoly` é um pacote para avaliação simbólica e numérica de muitos tipos de polinómios ortogonais, incluindo polinómios de Chebyshev, Laguerre, Hermite, Jacobi, Legendre, e ultraesférico (Gegenbauer). Adicionalmentey, `orthopoly` inclui suporte funções esféricas segundo o critério de Bessel, esféricas segundo o critério de Hankel, e funções harmônica esféricas.

Em sua maior parte, `orthopoly` segue as convenções de Abramowitz e Stegun *Handbook of Mathematical Functions*, Chapter 22 (10th printing, December 1972); adicionalmente, usamos Gradshteyn e Ryzhik, *Table of Integrals, Series, and Products* (1980 corrected and enlarged edition), e Eugen Merzbacher *Quantum Mechanics* (2nd edition, 1970).

Barton Willis da University de Nebraska e Kearney (UNK) escreveu o pacote `orthopoly` e sua documetação. O pacote é liberado segundo a licença pública geral GNU (GPL).

61.1.1 Iniciando com orthopoly

`load ("orthopoly")` torna o pacote `orthopoly` disponível para uso.

Para encontrar o polinómio de Legendre de terceira ordem,

```
(%i1) legendre_p (3, x);
                         3              2
              5 (1 - x)    15 (1 - x)
(%o1)       - ---------- + ---------- - 6 (1 - x) + 1
                   2              2
```

Para expressar esse polinómio como uma soma de potências de x, aplique *ratsimp* ou *rat* para o resultado anterior.

```
(%i2) [ratsimp (%), rat (%)];
                           3           3
                        5 x  - 3 x  5 x  - 3 x
(%o2)/R/               [----------, ----------]
                            2           2
```

Alternativamente, faça o segundo argumento para `legendre_p` (sua variável "principal") uma expressão racional canónica (CRE) usando `rat(x)` em lugar de somente `x`.

```
(%i1) legendre_p (3, rat (x));
                               3
                            5 x  - 3 x
(%o1)/R/                    ----------
                               2
```

Para avaliação em ponto flutuante, `orthopoly` usa uma análise de erro durante a execução para estimar uma associação superior para o erro. Por exemplo,

```
(%i1) jacobi_p (150, 2, 3, 0.2);
(%o1) interval(- 0.062017037936715, 1.533267919277521E-11)
```

intervalos possuem a forma `interval (c, r)`, onde c é o centro e r é o raio do intervalo. Uma vez que Maxima não suporta aritmética sobre intervalos, em algumas situações, tais

como em gráficos, vai querer suprimir o erro e sair somente com o centro do intervalo. Para fazer isso, escolha a variável de opção `orthopoly_returns_intervals` para `false`.

```
(%i1) orthopoly_returns_intervals : false;
(%o1)                          false
(%i2) jacobi_p (150, 2, 3, 0.2);
(%o2)                  - 0.062017037936715
```

Veja a secção ver [Avaliação em Ponto Flutuante], Página 643, para maiores informações.

Muitas funções em `orthopoly` possuem uma propriedade `gradef`; dessa forma

```
(%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
```

A função de um único passo no segundo exemplo previne um erro que poderia de outra forma surgir através da avaliação de n para 0.

```
(%i3) ev (%, n = 0);
(%o3)                           0
```

A propriedade `gradef` somente aplica para a variável "principal"; dderivadas com relação a outros argumentos usualmente resultam em uma mensagem de erro; por exemplo

```
(%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);
```

Geralmente, funções em `orthopoly` mapeiam sobre listas e matrizes. Para o mapeamento para avaliação total, as variáveis de opção `doallmxops` e `listarith` devem ambas serem `true` (o valor padrão). Para ilustrar o mapeamento sobre matrizes, considere

```
(%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       ]
```

No segundo exemplo, o elemento `i, j` do valor é `hermite (2, m[i,j])`; isso não é o mesmo que calcular `-2 + 4 m . m`, como visto no próximo exemplo.

```
(%i4) -2 * matrix ([1, 0], [0, 1]) + 4 * m . m;
                      [ 4 x y - 2      0     ]
(%o4)                 [                      ]
                      [     0      4 x y - 2 ]
```

Se avaliar uma função em um ponto fora do seu domínio, geralmente `orthopoly` retorna uma função não avaliada. Por exemplo,

```
(%i1) legendre_p (2/3, x);
(%o1)                           P   (x)
                                 2/3
```

`orthopoly` suporta tradução em TeX; `orthopoly` também faz saídas bidimensionais em um 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
```

61.1.2 Limitations

Quando uma expressão envolve muitos polinómios ortogonais com ordens simbólicas, é possível que a expressão actualmente tenda para zero, e ainda ocorre também que Maxima estar incapacitado de simplificar essa expressão para zero. Se fizer uma divisão por tal quantidade que tende a zero, poderá ficar em apuros. Por exemplo, a seguinte expressão tende para zero para inteiros n maiores que 1, e ainda ocorre também que Maxima está incapacitado de simplificar essa expressão para zero.

```
(%i1) (2*n - 1) * legendre_p (n - 1, x) * x - n * legendre_p (n, x) + (1 - n) * legend
(%o1)  (2 n - 1) P     (x) x - n P (x) + (1 - n) P     (x)
                  n - 1           n               n - 2
```

Para um n específico, podemos reduzir a expressão a zero.

```
(%i2) ev (% ,n = 10, ratsimp);
(%o2)                             0
```

Geralmente, a forma polinomial de um polinómio ortogonal esteja adequada de forma hostil para avaliaçao em ponto flutuante. Aqui está um exemplo.

```
(%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
```

O verdadeiro valor está em torno de 0.184; ess calculo suporta erro de cancelamento por extremo subtrativo.Expandindo o polinómio e então avaliando, fornecendo um melhor resultado.

```
(%i5) p : expand(p)$
(%i6) subst (0.2, x, p);
(%o6) 0.18413609766122982
```

Essa não é uma regra geral; expandindo o polinómio não resulta sempre em expressões que são melhores adaptadas a avaliação numérica. Com grande folga, o melhor caminho para fazer avaliação numérica é fazer um ou mais argumentos da função serem números em ponto flutuante. Em função disso, algoritmos especializados em ponto flutuante são usados para avaliação.

A função `float` do Maxima é até certo ponto indiscriminada; se aplicar `float` a uma expressão envolvendo um polinómio ortogonal com um grau simbólico ou um parâmetro de ordem, esses parâmetos (inteiros) podem ser convertido em números em ponto flutuante; após o que, a expressão não irá avaliar completamente. Considere

```
(%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
```

A expressão em (%o3) não irá avaliar para um número em ponto flutuante; `orthopoly` não reconhece valores em ponto flutuante em lugares onde deve haver valores inteiros. Similarmente, avaliação numérica da função `pochhammer` para ordens que excedam `pochhammer_max_index` pode ser perturbador; considere

```
(%i1) x :  pochhammer (1, 10), pochhammer_max_index : 5;
(%o1)                          (1)
                                  10
```

Aplicando `float` não avalia x para um número em ponto flutuante

```
(%i2) float (x);
(%o2)                        (1.0)
                                  10.0
```

Para avaliar x para um número em ponto flutuante, irá precisar associar `pochhammer_max_index` a 11 ou mais e aplicar `float` a x.

```
(%i3) float (x), pochhammer_max_index : 11;
(%o3)                        3628800.0
```

O valor padrão de `pochhammer_max_index` é 100; modifique esse valor após chama `orthopoly`.

Finalmente, tenha consciência que os livros citados nas referências adotam diferentes definições de polinómios ortogonais; geralmente adotamos as convenções citadas nas convenções de Abramowitz e Stegun.

Antes de suspeitar de um erro no pacote `orthopoly`, verifique alguns casos especiais para determinar se suas definições coincidem com aquelas usadas por `orthopoly`. Definitions muitas vezes diferem por uma normalização; ocasionalmente, autores utilizam versões "modificadas" das funções que fazem a família ortogonal sobre um intervalo diferente do intervalo $(-1, 1)$. Para definir, por exemplo, um polinómio de Legendre que é ortogonal a $(0, 1)$, defina

```
(%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
```

61.1.3 Avaliação em Ponto Flutuante

Muitas funções em `orthopoly` utilizam análise de erro durante a execução para estimar o erro em avaliações em ponto flutuante; as exceções são funções de Bessel esféricas e os polinómios associados de Legendre do segundo tipo. Para avaliações numéricas, as funções de Bessel esféricas chamam funções da colecção de programas `SLATEC`. Nenhum método especializado é usado para avaliação numérica dos polinómios associados de Legendre do segundo tipo.

A análise de erro durante a execução ignora erros que são de segunda ordem ou maior na máquina (também conhecida como perda de algarismos). A análise de erro durante a execução também ignora alguns poucos outros tipos de erro. É possível (embora não provável) que o erro actual exceda o estimado.

Intervalos possuem a forma `interval (c, r)`, onde c é o centro do intervalo e r é seu raio. O centro de um intervalo pode sr um número complexo, e o raio é sempre um número real positivo.

Aqui está um exemplo.

```
(%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) 1.84136091351685630913702249589134936908689044636 68b-1
```

Vamos testar o quanto o erro actual é é menor que o erro estimado

```
(%i4) is (abs (part (y0, 1) - y1) < part (y0, 2));
(%o4)                          true
```

Realmente, por esse exemplo o erro estimado é um maior que o erro verdadeiro.

Maxima não suporta aritmética sobre intervalos.

```
(%i1) legendre_p (7, 0.1) + legendre_p (8, 0.1);
(%o1) interval(0.18032072148437508, 3.1477135311021797E-15)
        + interval(- 0.19949294375000004, 3.3769353084291579E-15)
```

Um utilizador pode definir operadores aritméticos que fazem matemática de intervalos. Para definir adição de intervalos, podemos definir

```
(%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)
```

As rotinas eseciais em ponto flutuante são chamadas quando os argumentos forem complexos. Por exemplo,

```
(%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
```

Adicionalmente, quando os argumentos forem grandes números em ponto flutuante, as rotinas especiais de ponto flutuante são chamadas; todavia, tos grandes números em ponto flutuante são convertidos para números em ponto flutuante de dupla precisão e o resultado final é número em ponto flutuante de precisão dupla.

```
(%i1) ultraspherical (150, 0.5b0, 0.9b0);
(%o1) interval(- 0.043009481257265, 3.3750051301228864E-14)
```

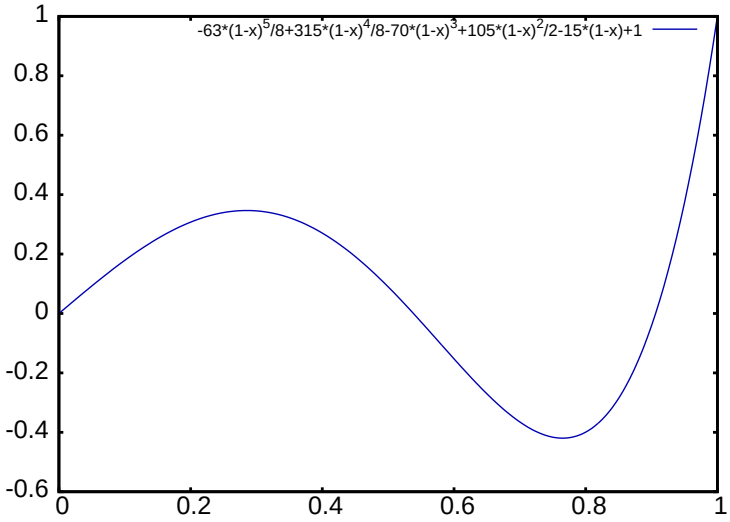
61.1.4 Gráficos e `orthopoly`

Para desenhar gráficos de expressões que envolvem polinómios ortogonais, deverá fazer duas coisas:

1. Escolher a variável de opção `orthopoly_returns_intervals` para `false`,
2. Colocar apóstrofo em qualquer chamada a funções do pacote `orthopoly`.

Se chamadas a funções não receberem apóstrofo, Maxima irá avaliá-las para polinómios antes de montar o gráfico; consequêntemente, as rotinas especializadas em ponto flutuante não serão chamadas. Aqui está um exemplo de como montar o gráfico de uma expressão que envolve um polinómio de Legendre.

```
(%i1) plot2d ('(legendre_p (5, x)), [x, 0, 1]), orthopoly_returns_intervals : false;
(%o1)
```



A expressão *completa* `legendre_p (5, x)` recebe apóstrofo; isso é diferente de apenas colocar apóstrofo no nome da função usando `'legendre_p (5, x)`.

61.1.5 Funções Diversas

O pacote `orthopoly` define o síbolo de Pochhammer e uma função de passo de unidade. `orthopoly` utiliza a função delta de Kronecker e a função de passo de unidade em declarações `gradef`.

Para converter os símbolos Pochhammer em quocientes da funções gama, use `makegamma`.

```
(%i1) makegamma (pochhammer (x, n));
                           gamma(x + n)
(%o1)                      ------------
                             gamma(x)
(%i2) makegamma (pochhammer (1/2, 1/2));
                                 1
(%o2)                            ---------
                                 sqrt(%pi)
```

Derivadas de símbolos de Pochhammer são fornecidas em termos de `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
```

É preciso ser cuidadoso com expressões como (%o1); a diferença das funções `psi` possuem polinómios quando `x` = `-1, -2, .., -n`. Esses polinómios cacelam-se com factores em `pochhammer (x, n)` fazendo da derivada um polinómio de grau `n - 1` quando n for um inteiro positivo.

O símbolo de Pochhammer é definido de ordens negativas até sua representação como um quociente de funções gama. Considere

```
(%i1) q : makegamma (pochhammer (x, n));
```

```
                              gamma(x + n)
(%o1)                         ------------
                                gamma(x)
(%i2) sublis ([x=11/3, n= -6], q);
                                   729
(%o2)                            - ----
                                   2240
```

Alternativamente, podemos tomar ese resultado directamente.

```
(%i1) pochhammer (11/3, -6);
                                   729
(%o1)                            - ----
                                   2240
```

A função passo de unidade é contínua à esquerda; dessa forma

```
(%i1) [unit_step (-1/10), unit_step (0), unit_step (1/10)];
(%o1)                          [0, 0, 1]
```

Se precisar de uma função de degrau unitário que seja ou contínua à esquerda ou contínua à direita do zero, defina a sua própria função usando `signum`; por exemplo,

```
(%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
```

Não redefina a própria `unit_step`; alguns código em `orthopoly` requerem que a função de passo de unidade seja contínua à esquerda.

61.1.6 Algorítmos

Geralmente, `orthopoly` faz avaliações simbólicas pelo uso de uma representação hipergeométrica de polinómios ortogonais. As funções hipegeométricas são avaliadas usando as funções (não documetadas) `hypergeo11` e `hypergeo21`. As excessões são as funções de Bessel metade inteiras e a função de Legendre associada de segundo tipo. As funções de Bessel metade inteiras são avaliadas usando uma representação explícita, e a função de Legendre associada de segundo tipo é avaliada usando recursividade.

Para avaliação em ponto flutuante, nós novamente convertemos muitas fuções em uma forma hipergeométrica; nós avaliamos as funções hipergeométricas usando recursividade para frente. Novamente, as excessões são as funções de Bessel metade inteiras e a função de Legendre associada de segundo tipo. Numericamente, as funções de Bessel meio inteiras são avaliadas usando o código SLATEC.

61.2 Definições para polinómios ortogonais

`assoc_legendre_p` (*n*, *m*, *x*) [Função]

As funções de Legendre associadas de primeiro tipo.

Referência: Abramowitz e Stegun, equações 22.5.37, página 779, 8.6.6 (segunda equação), página 334, e 8.2.5, página 333.

`assoc_legendre_q` (*n*, *m*, *x*) [Função]
A função de Legendre associada de segundo tipo.
Referência: Abramowitz e Stegun, equação 8.5.3 e 8.1.8.

`chebyshev_t` (*n*, *x*) [Função]
A função de Chebyshev de primeiro tipo.
Referência: Abramowitz e Stegun, equação 22.5.47,página 779.

`chebyshev_u` (*n*, *x*) [Função]
A função de Chebyshev do segundo tipo.
Referência: Abramowitz e Stegun, equação 22.5.48,página 779.

`gen_laguerre` (*n*, `a`, *x*) [Função]
O poliômio generalizado de Laguerre.
Referência: Abramowitz e Stegun, equação 22.5.54,página 780.

`hermite` (*n*, *x*) [Função]
O polinómio de Hermite.
Referência: Abramowitz e Stegun, equação 22.5.55,página 780.

`intervalp` (*e*) [Função]
Retorna `true` se a entrada for um intervalo e retorna `false` se não for.

`jacobi_p` (*n*, `a`, *b*, *x*) [Função]
o polinómio de Jacobi.
Os polinómios de Jacobi são actualmente definidos para todo *a* e *b*; todavia, o peso do polinómio de Jacobi `(1 - x)^a (1 + x)^b` não é integrável para `a <= -1` ou *b* `<= -1`.
Referência: Abramowitz e Stegun, equação 22.5.42,página 779.

`laguerre` (*n*, *x*) [Função]
O polinómio de Laguerre.
Referência: Abramowitz e Stegun, equatções 22.5.16 e 22.5.54,página 780.

`legendre_p` (*n*, *x*) [Função]
O polinómio de Legendre de primeiro tipo.
Referência: Abramowitz e Stegun, equações 22.5.50 e 22.5.51,página 779.

`legendre_q` (*n*, *x*) [Função]
O polinómio de Legendre de primeiro tipo.
Referência: Abramowitz e Stegun, equações 8.5.3 e 8.1.8.

`orthopoly_recur` (*f*, *args*) [Função]
Retorna uma relação recursiva para a família de funções ortogonais *f* com argumentos *args*. A recursividade é com relação ao grau do polinómio.

```
(%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
```

O segundo argumento a `orthopoly_recur` deve ser uma lista com o número correcto de argumentos para a função f; se o número de argumentos não for o correcto, Maxima sinaliza com um erro.

```
(%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);
```

Adicionalmente, quando f não for o nome de uma das famílias de polinómios ortogonais, um erro é sinalizado.

```
(%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]

Valor por omissão: `true`

Quando `orthopoly_returns_intervals` for `true`, resultados em ponto flutuante são retornados na forma `interval (c, r)`, onde c é o centro de um intervalo e r é seu raio. O centro pode ser um número complexo; nesse caso, o intervalo é um disco no plano complexo.

`orthopoly_weight` (*f*, *args*) [Função]

Retorna uma lista de três elementos; o primeiro elemento é a fórmula do peso para a família de polinómios ortogonais f com argumentos fornecidos pela lista *args*; os segundos e terceiros elementos fornecem os pontos finais inferior e superior do intervalo de ortogonalidade. Por exemplo,

```
(%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
```

A variável principal de f deve ser um símbolo; Se não for, Maxima sinaliza com um erro.

`pochhammer` (*n*, *x*) [Função]

O símbolo de Pochhammer. Para inteiros não negativos n com `n <= pochhammer_max_index`, a expressão `pochhammer (x, n)` avalia para o produto `x (x + 1) (x + 2) ... (x + n - 1)` when `n > 0` e para 1 quando `n = 0`. Para valores negativos de n, `pochhammer (x, n)` é definido como `(-1)^n / pochhammer (1 - x, -n)`. Dessa forma

```
(%i1) pochhammer (x, 3);
(%o1)                   x (x + 1) (x + 2)
(%i2) pochhammer (x, -3);
                                1
```

```
(%o2)                - ----------------------
                       (1 - x) (2 - x) (3 - x)
```

Para converter um símbolo de Pochhammer em um quociente de funções gama, (veja Abramowitz e Stegun, equação 6.1.22) use `makegamma`; por exemplo

```
(%i1) makegamma (pochhammer (x, n));
                            gamma(x + n)
(%o1)                       ------------
                              gamma(x)
```

Quando n exceder `pochhammer_max_index` ou quando n for simbólico, `pochhammer` retorna uma forma substantiva.

```
(%i1) pochhammer (x, n);
(%o1)                          (x)
                                  n
```

`pochhammer_max_index` [Variável]

Valor por omissão: 100

`pochhammer (n, x)` expande para um produto se e somente se `n <= pochhammer_max_index`.

Exemplos:

```
(%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
```

Referência: Abramowitz e Stegun, equação 6.1.16,página 256.

`spherical_bessel_j (n, x)` [Função]

A Função de Bessel esférica de primeiro tipo.

Referência: Abramowitz e Stegun, equações 10.1.8,página 437 e 10.1.15,página 439.

`spherical_bessel_y (n, x)` [Função]

A Função de Bessel esférica de segundo tipo.

Referência: Abramowitz e Stegun, equações 10.1.9,página 437 e 10.1.15,página 439.

`spherical_hankel1 (n, x)` [Função]

A Função de Hankel esférica de primeiro tipo.

Referência: Abramowitz e Stegun, equação 10.1.36,página 439.

`spherical_hankel2 (n, x)` [Função]

A Função de Hankel esférica de segundo tipo.

Referência: Abramowitz e Stegun, equação 10.1.17,página 439.

`spherical_harmonic (n, m, x, y)` [Função]

A função armônica esférica.

Referência: Merzbacher 9.64.

`unit_step (x)` [Função]

A função de passo de unidade contínua à esquerda; dessa forma `unit_step (x)` tende para `x <= 0` e é igual a 1 para `x > 0`.

Se quiser uma função de degrau unitário que tome o valor 1/2 em zero, use `(1 + signum (x))/2`.

`ultraspherical (n, a, x)` [Função]

A função polinômial ultraesférica (também conhecida como função polinomial de Gegenbauer).

Referência: Abramowitz e Stegun, equação 22.5.46,página 779.

62 plotdf

62.1 Introdução a plotdf

A função `plotdf` cria um gráfico do campo de direcções para uma Equação Diferencial Ordinária (EDO) de primeira ordem, ou para um sistema de duas EDO's autónomas, de primeira ordem.

Por tratar-se de um pacote adicional, para poder usá-lo deverá primeiro carregá-lo com o comando `load("plotdf")`. Também é necessário que Xmaxima esteja instalado, inclusivamente se executar o Maxima desde outra interface diferente.

Para desenhar o campo de direcções de uma única EDO, essa equação deverá escrever-se na forma seguinte:

$$\frac{dy}{dx} = F(x, y)$$

e a função F será dada como argumento para o comando `plotdf`. A variável independente tem que ser sempre x e a variável dependente y. A essas duas variáveis não poderá estar associado nenhum valor numérico.

Para desenhar o campo de direcções de um sistema autónomo de duas EDO's, as duas equações devem ser escritas na forma seguinte

$$\frac{dx}{dt} = G(x, y) \qquad \frac{dy}{dt} = F(x, y)$$

e o argumento para o comando `plotdf` será uma lista com duas expressões para as funções F e G.

Quando se trabalha com uma única equação, `plotdf` assume implicitamente que `x=t` e `G(x,y)=1`, transformando a equação num sistema autónomo com duas equações.

62.2 Definições para plotdf

`plotdf` (*dydx,...opções...*) [Function]
`plotdf` ([*dxdt,dydt*],*...opções...*) [Function]

Desenha um campo de direcções em duas dimensões x e y.

dydx, *dxdt* e *dydt* são expressões que dependem de x e y. Para além dessas duas variáveis, as duas expressões podem depender de um conjunto de parâmetros, com valores numéricos que são dados por meio da opção `parameters` (a sintaxe dessa opção explica-se mais para a frente), ou com um intervalo de possíveis valores definidos com a opção *sliders*.

Várias outras opções podem incluirem-se dentro do comando, ou serem seleccionadas no menú. Clicando num ponto do gráfico faz com que seja desenhada a curva integral que passa por esse ponto; o mesmo pode ser feito dando as coordenadas do ponto com a opção `trajectory_at` dentro do comando plotdf. A direcção de integração pode controlar-se com a opção `direction`, que aceita valores de *forward*, *backward* ou *both*. O número de passos realizados na integração numérica controla-se com a opção `nsteps` e o incremento do tempo em cada passo com a opção `tstep`. Usa-se

o método de Adams Moulton para fazer a integração numérica; também é possível mudar para o método de Runge-Kutta de quarta ordem com ajuste de passos.

Menú da janela do gráfico:

O menú da janela gráfica inclui as seguintes opções: *Zoom*, que permite mudar o comportamento do rato, de maneira que servirá para fazer zoom na região do gráfico clicando com o botão esquerdo. Cada clic alarga a imagem mantendo no centro dela o ponto onde se clicou. Mantendo carregada a tecla `Shift` enquanto se faz clic, faz diminuir o tamanho. Para continuar a desenhar trajectórias com um clic, selecciona-se a opção *Integrate* do menú.

A opção *Config* do menú pode usar-se para mudar a(s) EDO(S) e fazer alguns outros ajustes. Após ter feito alguma alteração, deverá usar a opção *Replot* para activar as novas configurações. Se introduzir duas coordenadas no campo *Trajectory at* do menú de diálogo do *Config*, e a seguir carregar na tecla `Enter`, será acrescentada mais uma curva integral. Se seleccionar a opção *Replot*, só será apresentada a última curva integral seleccionada.

Mantendo o botão direito carregado enquanto se desloca o cursor, poderá arrastar o gráfico na horizontal e na vertical. Outros parâmetros, por exemplo, o número de passos, o valor inicial de t, as coordenadas do centro e o raio, podem ser alterados no sub-menú da opção *Config*.

Com a opção *Save*, pode imprimir-se o gráfico numa impressora Postscript ou gravar uma cópia num ficheiro Postscript. Para optar entre impressão ou gravação em ficheiro, selecciona-se *Print Options* na janela de diálogo de *Config*. Após ter preenchido os campos da janela de diálogo de *Save*, será necessário seleccionar a opção *Save* do primeiro menú para criar o ficheiro ou imprimir o gráfico.

Opções gráficas:

A função `plotdf` admite varias opções, cada uma sendo uma lista de duas ou mais elementos. O primeiro elemento é o nome da opção, e o resto estará formado pelos argumentos para essa opção.

A função `plotdf` reconhece as seguintes opções:

- *tstep* estabelece a amplitude dos incrementos da variável independente t, utilizados para calcular as curvas integrais. Se for dada só uma expressão *dydx*, a variável x será directamente proporcional a t. O valor por omissão é 0.1.
- *nsteps* estabelece o número de passos de comprimento `tstep` que se utilizarão na variável independente para calcular a curva integral. O valor por omissão é 100.
- *direction* estabelece a direcção da variável independente que será seguida para calcular uma curva integral. Os valores possí são: `forward`, para fazer que a variável independente aumente `nsteps` vezes, com incrementos `tstep`; `backward`, para fazer que a variável independente diminua; `both`, para estender a curva integral `nsteps` passos para a frente e `nsteps` passos para atrás. As palavras `right` e `left` podem serem usadas como sinónimos de `forward` e `backward`. O valor por omissão é `both`.
- *tinitial* estabelece o valor inicial da variável t utilizado para calcular curvas integrais. Já que as equações diferenciais são autónomas, esta opção só aparecerá nos gráficos das curvas em função de t. O valor por omissão é 0.

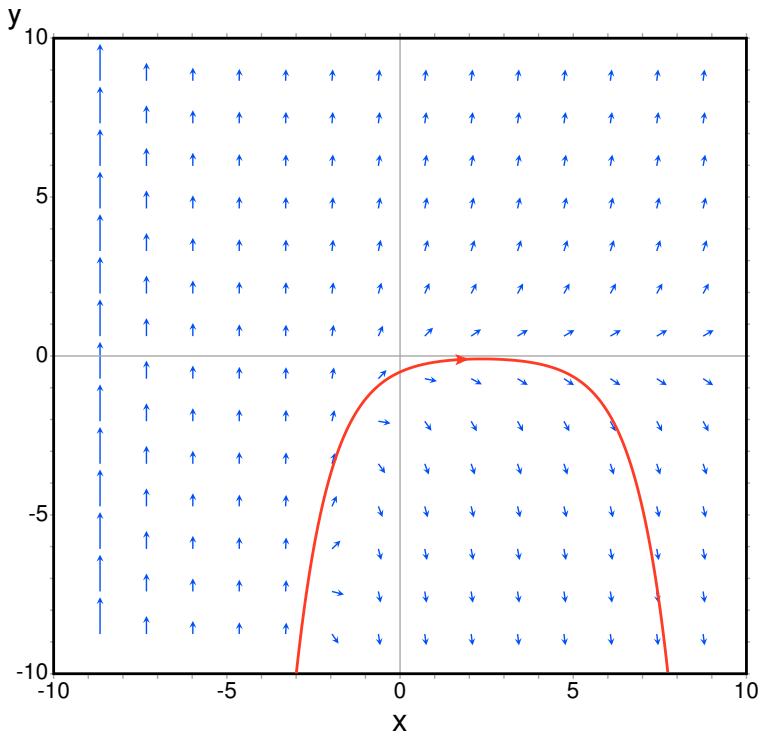
- *versus_t* utiliza-se para criar uma segunda janela gráfica, com o gráfico de uma curva integral, como duas funções x, y, de variável independente t. Se for dado a `versus_t` qualquer valor diferente de 0, mostrar-se-á a segunda janela gráfica, que inclui outro menú, similar ao da janela principal. O valor por omissão é 0.
- *trajectory_at* estabelece as coordenadas *xinitial* e *yinitial* para o ponto inicial da curva integral. Não tem atribuído nenhum valor por omissão.
- *parameters* estabelece uma lista de parâmetros, junto com os seus valores numéricos, que são utilizados na definição da equação diferencial. Os nomes dos parâmetros e os seus valores devem escrever-se em formato de cadeia de caracteres como uma sequência de pares `nome=valor` separados por vírgulas.
- *sliders* estabelece uma lista de parâmetros que poderão ser alterados interactivamente usando barras com sliders, assim como os intervalos de variação dos ditos parâmetros. Os nomes dos parâmetros e os seus intervalos devem escrever-se em formato de cadeia de caracteres como uma sequência de pares `nome=min:max` separados por vírgulas.
- *xfun* estabelece uma cadeia de caracteres com funções de x separadas por ponto e vírgula para ser representadas por cima do campo de direcções. Essas funções serão interpretadas por Tcl, e não por Maxima.
- *xradius* é metade do comprimento do intervalo de valores a representar na direcção x. O valor por omissão é 10.
- *yradius* é metade do comprimento do intervalo de valores a representar na direcção y. O valor por omissão é 10.
- *xcenter* é a coordenada x do ponto situado no centro do gráfico. O valor por omissão é 0.
- *ycenter* é a coordenada y do ponto situado no centro do gráfico. O valor por omissão é 0.
- *width* estabelece a largura da janela gráfica em pixels. O valor por omissão é 500.
- *height* estabelece a altura da janela gráfica em pixels. O valor por omissão é 500.

Exemplos:

NOTA: Em alguns casos, dependendo da interface usada para executar o Maxima, as funções que usam `openmath`, em particular `plotdf`, podem desencadear um bug se terminarem em ponto e vírgula, e não com o símbolo de dólar. Para evitar problemas, usaremos o símbolo de dóla nos exemplos a seguir.

- Para mostrar o campo de direcções da equação diferencial $y' = exp(-x) + y$ e a solução que passa por $(2, -0.1)$:

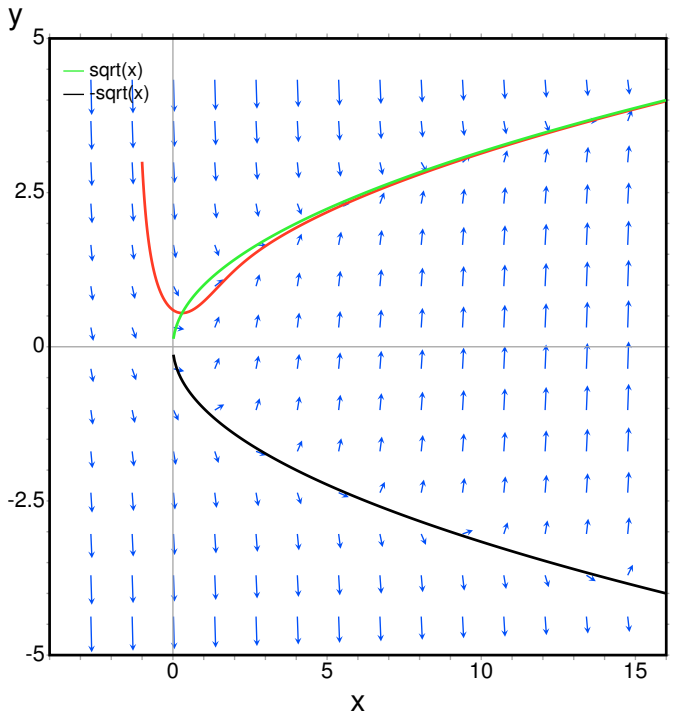
```
(%i1) load("plotdf")$

(%i2) plotdf(exp(-x)+y,[trajectory_at,2,-0.1]);
```



- Para mostrar o campo de direcções da equação $diff(y,x) = x - y^2$ e a solução com condição inicial $y(-1) = 3$, pode utilizar-se o comando:

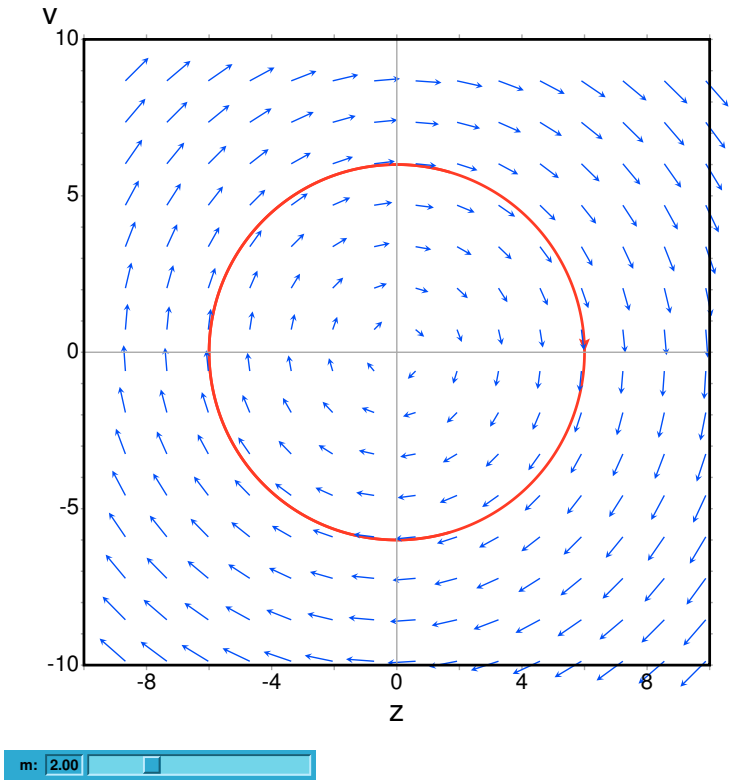
```
(%i3) plotdf(x-y^2,[xfun,"sqrt(x);-sqrt(x)"],
          [trajectory_at,-1,3], [direction,forward],
          [yradius,5],[xcenter,6]);
```

O gráfico também mostra a função $y = sqrt(x)$.



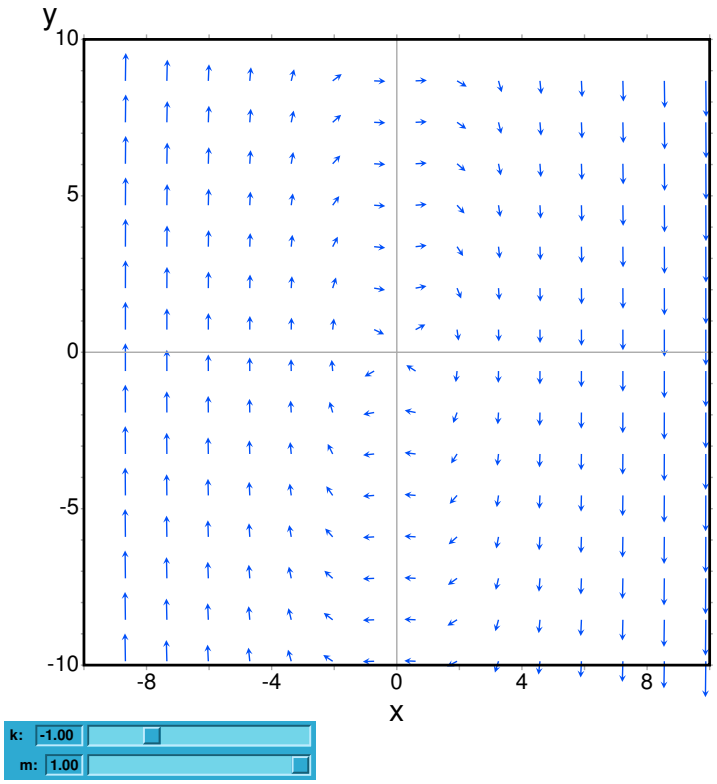
- O exemplo seguinte mostra o campo de direcções de um oscilador harmónico, definido pelas equações $dx/dt = y$ e $dy/dt = -k * x/m$, e a curva integral que passa por $(x, y) = (6, 0)$, com uma barra de slider que permitirá mudar o valor de m interactivamente (k permanecerá fixo em 2):

```
(%i4) plotdf([y,-k*x/m],[parameters,"m=2,k=2"],
             [sliders,"m=1:5"], [trajectory_at,6,0]);
```



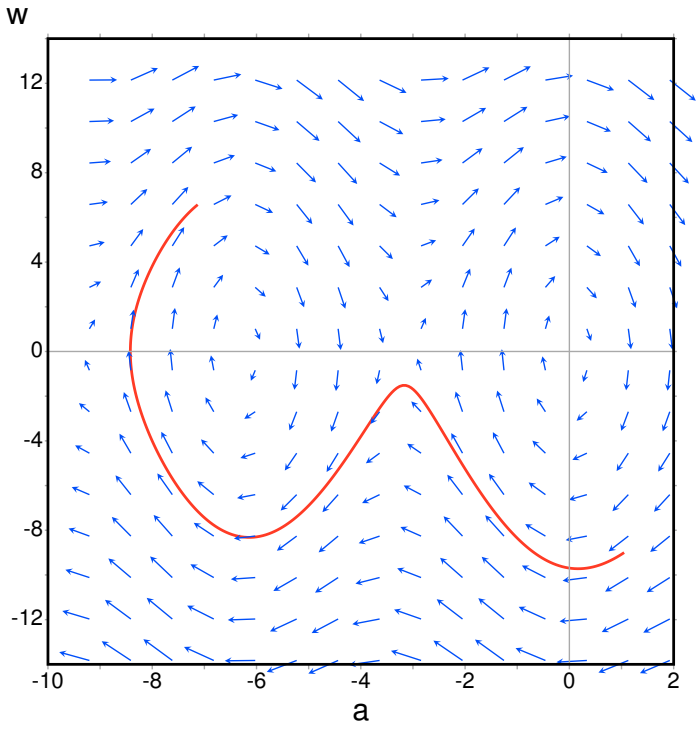
- Para representar o campo de direcções da equação de Duffing, $m * x'' + c * x' + k * x + b * x^3 = 0$, introduz-se a variável $y = x'$ e faz-se:

```
(%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]);
```

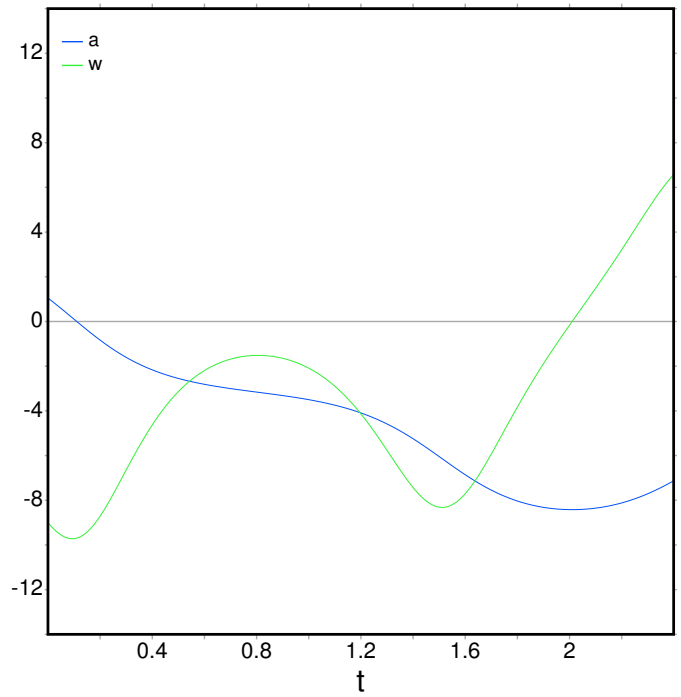


- O campo de direcções dum pêndulo amortecido, incluindo a solução para condições iniciales dadas, com uma barra de slider que pode usar-se para mudar o valor da masa, m, e com o gráfico das duas variáveis de estado em função do tempo:

```
(%i6) plotdf([y,-g*sin(x)/l - b*y/m/l],
         [parameters,"g=9.8,l=0.5,m=0.3,b=0.05"],
         [trajectory_at,1.05,-9],[tstep,0.01],
         [xradius,6],[yradius,14],
         [xcenter,-4],[direction,forward],[nsteps,300],
         [sliders,"m=0.1:1"], [versus_t,1]);
```


w
12
8
4
0
-4
-8
-12
-10
-8
-6
-4
-2
0
2
a


m:
0.297


a
w
12
8
4
0
-4
-8
-12
0.4
0.8
1.2
1.6
2
t

63 romberg

63.1 Definições para romberg

`romberg` (*expr*, *x*, `a`, *b*) [Função]
`romberg` (*F*, `a`, *b*) [Função]

Calcula uma integração numérica pelo método de Romberg.

`romberg(`*expr*`, `*x*`, a, `*b*`)` retorna uma estimativa da integral `integrate(`*expr*`, `*x*`, a, `*b*`)`. *expr* deve ser uma expressão que avalie para um valor em ponto flutuante quando *x* estiver associado a um valor em ponto flutuante.

`romberg(`*F*`, a, `*b*`)` retorna uma estimativa da integral `integrate(`*F*`(x), x, a, `*b*`)` onde `x` representa o não nomeado, isolado argumeno de *F*; o actual argumento não é chamado `x`. *F* deve ser uma função do Maxima ou do Lisp que retorne um valor em ponto flutuante quando o argumento for um número em ponto flutuante. *F* pode nomear uma função traduzida ou compilada do Maxima.

A precisão de `romberg` é governada pelas variáveis globais `rombergabs` e `rombergtol`. `romberg` termina com sucesso quando a diferença absoluta entre duas aproximações sucessivas for menor que `rombergabs`, ou a diferença relativa em aproximações sucessivas for menor que `rombergtol`. Dessa forma quando `rombergabs` for 0.0 (o padrão) somente o erro relativo tem algum efeito sobre `romberg`.

`romberg` divide ao meio o tamanho do passo no máximo `rombergit` vezes antes de interromper; o número máximo de avaliações de função é portanto `2^rombergit`. Se o critério de erro estabelecido por `rombergabs` e por `rombergtol` não for satisfeito, `romberg` mostra uma mensagem de erro. `romberg` sempre faz ao menos `rombergmin` iterações; isso é uma intenção eurísstica de previnir encerramentos espúrios quando o integrando for oscilatório.

`romberg` repetidamente avalia o integrando após associar a variável de integração a um valor específico (e não antes). Essa política de avaliação torna possível aninhar chamadas a `romberg`, para calcular integrais multidimensionais. Todavia, os cálculos de erro não tomam os erros de integrações aninhadas em consideração, então erros podem ser subestimados. Também, métodos imaginados especialmente para problemas multidimensionais podem retornar a mesma precisão com poucas avaliações de função.

`load("romberg")` torna essa função disponível para uso.

Veja também `QUADPACK`, uma colecção de funções de integração numérica.

Exemplos:

Uma integração unidimensonal.

```
(%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 + 1/2
                      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
```

Uma integração bidimensional, implementada com chamadas aninhadas a `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` [Variável de opção]

Valor por omissão: 0.0

A precisão de `romberg` é governada pelas variávies globais `rombergabs` e `rombergtol`. `romberg` termina com sucesso quando a diferença absoluta entre duas aproximações sucessivas for menor que `rombergabs`, ou a diferença relativa em aproximações sucessivas for menor que `rombergtol`. Dessa forma quando `rombergabs` for 0.0 (o padrão) somente o erro relativo tem algum efeito sobre `romberg`.

Veja também `rombergit` e `rombergmin`.

`rombergit` [Variável de opção]

Valor por omissão: 11

`romberg` divide ao meio o tamanho do passo no máximo `rombergit` vezes antes de interromper; o número máximo de avaliações de função é portanto `2^rombergit`. Se o critério de erro estabelecido por `rombergabs` e por `rombergtol` não for satisfeito, `romberg` mostra uma mensagem de erro. `romberg` sempre faz ao menos `rombergmin` iterações; isso é uma intenção eurísstica de previnir encerramentos espúrios quando o integrando for oscilatório.

Veja também `rombergabs` e `rombergtol`.

`rombergmin` [Variável de opção]

Valor por omissão: 0

`romberg` sempre faz ao menos `rombergmin` iterações; isso é uma intenção eurísstica para prevenir terminações espúrias quando o integrando for.

Veja também `rombergit`, `rombergabs`, e `rombergtol`.

`rombergtol` [Variável de opção]

Valor por omissão: 1e-4

A precisão de `romberg` é governada pelas variáveis globais `rombergabs` e `rombergtol`. `romberg` termina com sucesso quando a diferença absoluta entre duas aproximações sucessivas for menor que `rombergabs`, ou a diferença relativa em aproximações sucessivas for menor que `rombergtol`. Dessa forma quando `rombergabs` for 0.0 (o padrão) somente o erro relativo tem algum efeito sobre `romberg`.

Veja também `rombergit` e `rombergmin`.

64 simplex

64.1 Introdução a simplex

`simplex` é um pacote para optimização linear usando o algoritmo simplex.

Exemplo:

```
(%i1) load("simplex")$
(%i2) minimize_sx(x+y, [3*x+2*y>2, x+4*y>3]);
                  9        7       1
(%o2)            [--, [y = --, x = -]]
                  10       10      5
```

64.2 Definições para simplex

`epsilon_sx` [Variável de opção]

Valor por omissão: `10^-8`

Epsilon usando para cálculos numéricos em `linear_program`.

Veja também: `linear_program`.

`linear_program` (*A*, *b*, *c*) [Função]

`linear_program` é uma implementação do algoritmo simplex. `linear_program(A, b, c)` calcula um vetor *x* para o qual `c.x` é o mínimo possível entre vetores para os quais `A.x = b` e `x >= 0`. O argumento *A* é uma matriz e os argumentos *b* e *c* são listas.

`linear_program` retorna uma lista contendo o vetor minimizado *x* e o valor mínimo `c.x`. Se o problema for não associado, é retornado `"Problem not bounded!"` e se o problema for não viável, é retornado `"Problem not feasible!"`.

Para usar essa função primeiramente chame o pacote `simplex` com `load("simplex");`.

Exemplo:

```
(%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
```

Veja também: `minimize_sx`, `scale_sx`, e `epsilon_sx`.

`maximize_sx` (*obj*, *cond*, [*pos*]) [Função]

Maximiza a função linear objetiva *obj* submetida a alguma restrição linear *cond*. Veja `minimize_sx` para uma descrição detalhada de argumentos e valores de retorno.

Veja também: `minimize_sx`.

`minimize_sx` (*obj*, *cond*, [*pos*]) [Função]

Minimiza uma função linear objetiva *obj* submetida a alguma restrição linear *cond*. *cond* é uma lista de equações lineares ou desigualdades. Em desigualdades estritas `>` é substituido por `>=` e `<` por `<=`. O argumento opcional *pos* é uma lista de variáveis de decisão que são assumidas como sendo positivas.

Se o mínimo existir, `minimize_sx` retorna uma lista que contém o menor valor da função objetiva e uma lista de valores de variáveis de decisão para os quais o mínimo é alcançado. Se o problema for não associado, `minimize_sx` retorna "Problem not bounded!" e se o problema for não viável, é retornado "Ploblem not feasible!".

As variáveis de decisão não são assumidas para serem não negativas por padrão. Se todas as variáveis de dicisão forem não negativas, escolha `nonegative_sx` para `true`. Se somente algumas das variáveis de decisão forem positivas, coloque-as então no argumento opcional *pos* (note que isso é mais eficiente que adicionar restrições).

`minimize_sx` utiliza o algoritmo simplex que é implementado na função `linear_program` do Maxima.

Para usar essa função primeiramente chame o pacote `simplex` com `load("simplex");`.

Exemplos:

```
(%i1) minimize_sx(x+y, [3*x+y=0, x+2*y>2]);
                       4       6        2
(%o1)                 [-, [y = -, x = - -]]
                       5       5        5
(%i2) minimize_sx(x+y, [3*x+y>0, x+2*y>2]), nonegative_sx=true;
(%o2)                 [1, [y = 1, x = 0]]
(%i3) minimize_sx(x+y, [3*x+y=0, x+2*y>2]), nonegative_sx=true;
(%o3)                Problem not feasible!
(%i4) minimize_sx(x+y, [3*x+y>0]);
(%o4)                Problem not bounded!
```

Veja também: `maximize_sx`, `nonegative_sx`, `epsilon_sx`.

`nonegative_sx` [Variável de opção]

Valor por omissão: `false`

Se `nonegative_sx` for verdadeiro (true) todas as variáveis de decisão para `minimize_sx` e `maximize_sx` são assumidas para serem positivas.

Veja também: `minimize_sx`.

65 simplification

65.1 Introdução a simplification

O directório `maxima/share/simplification` contém muitos scripts que implementam regras de simplificação e funções, e também algumas funções não relacionadas a simplificação.

65.2 Definições para simplification

65.2.1 Package absimp

O pacote `absimp` contém regras de comparação de sequências decaractere que extendem as regras internas de simplificação para as funções `abs` e `signum`. `absimp` respeita as relações estabelecidas com a função interna `assume` e por meio de declarações tais como `modedeclare (m, even, n, odd)` para inteiros paes ou ímpares.

`absimp` define as funções `unitramp` e `unitstep` em termos de `abs` e `signum`.

`load ("absimp")` torna esse pacote disponível para uso. `demo (absimp)` faz uma demonstração desse pacote.

Exemplos:

```
(%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)
```

65.2.2 Package facexp

O pacote `facexp` contém muitas funções relacionadas a simplificações que fornecem ao utilizador a habilidade de estruturar expressões por meio de expansão controlada. Essa capacidade é especialmente útil quando a expressão contém variáveis que possuem significado físico, porque é muitas vezes verdadeiro que a forma mais econômica de uma tal expressão pode ser obtida por meio de uma expansão completa da expressão com relação a essas variáveis, e então factorizar seus coeficientes. Apesar de ser verdadeiro que esse procedimento é fácil de realizar usando as funções padrão do Maxima, ajustes adicionais podem se desejáveis, e esses toques finais podem ser mais difíceis de aplicar.

A função `facsum` e suas formas relacionadas fornecem um meio conveniente de controlar a estrutura de expressões por esse caminho. Outra função, `collectterms`, pode ser usada para adicionar duas ou mais expressões que já tenham sido simplificadas para essa forma, sem resimplificar a expressão completa novamente. Essa função pode ser útil quando expressões forem muito grandes.

`load ("facexp")` torna dispon;ivel para uso esse pacote. `demo (facexp)` faz uma demonstração desse pacote.

`facsum` (*expr*, *arg_1*, ..., *arg_n*) [Função]

Retorna uma forma de *expr* que depende dos argumentos *arg_1*, ..., *arg_n*. Os argumentos podem ser quaisquer formas adequadas para `ratvars`, ou eles podem ser listas de tais formas. Se os argumentos não forem listas, então a forma retornada é completamente expandida com relação aos argumentos, e os coeficientes dos argumentos foram factorizados. Esses coeficientes são livres dos argumentos, excepto talvez no sentido não racional.

Se quaisquer dos argumentos forem listas, então todas as tais listas são combinadas em uma lista simples, e em lugar de chamar `factor` sobre os coeficientes dos argumentos, `facsum` chama a si mesma sobre esses coeficientes, usando essa nova lista simples que foi construída como o novo argumento listo para essa chamada recursiva. Esse processo pode ser repetido para um quantidade arbitrária de repetições por através do aninhamento dos elementos desejados nas listas.

É possível que alguém possa querer usar `facsum` com relação a subexpressões mais complicadas, tal como `log (x + y)`. Tais argumentos são também permitidos. Sem especificação de variável, por exemplo `facsum (expr)`, o resultado retornado é o mesmo que o que é retornado por meio de `ratsimp (expr)`.

Ocasionalmente o utilizador pode querer obter quaisquer das formas abaixo para expressões que são especificadas somente por meio de seus operadores líderes. Por exemplo, alguém pode querer usar `facsum` com relação a todos os `log`'s. Nessa situação, alguém pode incluir no meio dos argumentos ou o código dos `log`'s eespecíficos que devem ser tratados po esse caminho ou alternativamente a expressão `operator (log)` ou a expressão `'operator (log)`. Se alguém quiser usar `facsum` na expressão *expr* com relação aos operadores *op_1*, ..., *op_n*, pode-se avaliar `facsum (expr, operator (op_1, ..., op_n))`. A forma `operator` pode também aparecer dentro de uma lista de argumentos.

Adicionalmente, a escolha de comutadores `facsum_combine` e `nextlayerfactor` pode afectar o ressultado de `facsum`.

`nextlayerfactor` [Variável global]

Valor por omissão: `false`

Quando `nextlayerfactor` for `true`, chamadas recursivas a `facsum` são aplicdas aos factores da forma factorizada dos coeficientes dos argumentos.

Quando `nextlayerfactor` for `false`, `facsum` é aplicada a cada coeficiente como um todo mesmo se chamadas recursivas a `facsum` acontecerem.

A inclusão do átomo `nextlayerfactor` na lista argumento de `facsum` tem o efieto de `nextlayerfactor: true`, mas para o próximo nível da expressão *somente*. Uma vez que `nextlayerfactor` é sempre associado ou a `true` ou a `false`, `nextlayerfactor` deve ser apresentada com apóstrofo simples mesmo que `nextlayerfactor` apareça na lista de argumento de `facsum`.

`facsum_combine` [Variável global]

Valor por omissão: `true`

`facsum_combine` controla a forma do resultado final retornada por meio de `facsum` quando seu argumento é um quociente de polinómios. Se `facsum_combine` for `false` então a forma será retornada como um somatório completametne expandido como

descrito acima, mas se `true`, então a expressão retornada é uma razão de polinómios, com cada polinómio na forma descrita acima.

A escolha de `true` desse comutador é útil quando se deseja para `facsum` ambos o dumerador e o denominador de uma expressão racional, mas não se deseja que o denominador seja multiplicado de forma completa pelos termos do numerador.

`factorfacsum` (*expr*, *arg_1*, ... *arg_n*) [Função]

Retorna uma forma de *expr* que é obtida por meio de chamada a `facsum` sobre os factores de *expr* com *arg_1*, ... *arg_n* como argumentos. Se qualqeur dos factores de *expr* estiver elevado a um expoente, ambos o factor e o expoente irão ser processados por esse meio.

`collectterms` (*expr*, *arg_1*, ..., *arg_n*) [Função]

Se muitas expressões tiverem sido simplificadas com `facsum`, `factorfacsum`, `factenexpand`, `facexpten` ou com `factorfacexpten`, e elas estão para serem adicionadas umas às outras, pode ser desejável combiná-las usando a função `collecterms`. `collecterms` pode pegar como argumentos todos os argumentos que podem ser fornecidos para essas outras funções associadas com excessão de `nextlayerfactor`, que não tem efeito sobre `collectterms`. A vantagem de `collectterms` está em que `collectterms` retorna uma forma similar a `facsum`, mas uma vez que `collectterms` está adicionando forma que já tenham sido processadas por `facsum`, `collectterms` não precisa repetir aquele esforço. Essa capacidade é especialmente útil quando a expressão a ser somada for muito grande.

65.2.3 Pacote functs

`rempart` (*expr*, *n*) [Função]

Remove a parte *n* da expressão *expr*.

Se *n* é uma lsita da forma `[l, m]` então as partes de *l* até *m* são removidas.

Para usar essa função escreva primeiramente `load("functs")`.

`wronskian` ([*f_1*, ..., *f_n*], *x*) [Função]

Retorna a matriz Wronskiana das funções *f_1*, ..., *f_n* na variável *x*.

f_1, ..., *f_n* pode ser o nome de funções definidas pelo utilizador, ou expressões na variável *x*.

O determinante da matriz Wronskiana é o determinante Wronskiano do conjunto de funções. As funções são linearmente independentes entre si se seu determinante for igual a zero.

Para usar essa função escreva primeiramente `load("functs")`.

`tracematrix` (*M*) [Função]

Retorna o traço (somatório dos elementos da diagonal principal) da matriz *M*.

Para usar essa função escreva primeiramente `load("functs")`.

`rational` (z) [Função]

Multiplica o numerador e o denominador de *z* pelo complexo conjugado do denominador, racionando dessa forma o denominador complexo. Retorna a forma de expressão racional canónica (CRE) se fornecida uma CRE, caso contrário retorna a forma geral.

Para usar essa função escreva primeiramente `load("functs")`.

`nonzeroandfreeof` (*x*, *expr*) [Função]
Retorna `true` se *expr* for diferente de zero e `freeof (x, expr)` retorna `true`. Retorna `false` de outra forma.

Para usar essa função escreva primeiramente `load("functs")`.

`linear` (*expr*, *x*) [Função]
Quando *expr* for uma expressão linear na variável *x*, `linear` retorna `a*x + b` onde *a* é diferente de zero, e *a* e *b* são livres de *x*. De outra forma, `linear` retorna *expr*.

Para usar essa função escreva primeiramente `load("functs")`.

`gcdivide` (*p*, *q*) [Função]
Quando `takegcd` for `true`, `gcdivide` divide os polinómios *p* e *q* por seu maior divisor comum (MDC) e retorna a razão dos resultados.

Quando `takegcd` for `false`, `gcdivide` retorna a razão *p*/*q*.

Para usar essa função escreva primeiramente `load("functs")`.

`arithmetic` (`a`, *d*, *n*) [Função]
Retorna o *n*-ésiomo termo da série aritmética `a`, `a + d`, `a + 2*d`, `...`, `a + (n - 1)*d`.

Para usar essa função escreva primeiramente `load("functs")`.

`geometric` (`a`, *r*, *n*) [Função]
Retorna o *n*-ésimo termo da série geométrica `a`, `a*r`, `a*r^2`, `...`, `a*r^(n - 1)`.

Para usar essa função escreva primeiramente `load("functs")`.

`harmonic` (`a`, *b*, *c*, *n*) [Função]
Retorna o *n*-ésimo termo da série harmônica `a/b`, `a/(b + c)`, `a/(b + 2*c)`, `...`, `a/(b + (n - 1)*c)`.

Para usar essa função escreva primeiramente `load("functs")`.

`arithsum` (`a`, *d*, *n*) [Função]
Retorna a soma dos elementos da série aritmética de 1 a *n*.

Para usar essa função escreva primeiramente `load("functs")`.

`geosum` (`a`, *r*, *n*) [Função]
Retorna a soma dos elementos da série geométrica de 1 a *n*. Se *n* for infinito (`inf`) então a soma será finita se e somente se o valor absoluto de *r* for menor que 1.

Para usar essa função escreva primeiramente `load("functs")`.

`gaussprob` (*x*) [Função]
Retorna a função de probalilidade de Gauss `%e^(-x^2/2) / sqrt(2*%pi)`.

Para usar essa função escreva primeiramente `load("functs")`.

`gd` (*x*) [Função]
Retorna a função de Gudermann `2 * atan(%e^x - %pi/2)`.

Para usar essa função escreva primeiramente `load("functs")`.

`agd (x)` [Função]
Retorna o inverso da função de Gudermann `log (tan (%pi/4 + x/2)))`.
Para usar essa função escreva primeiramente `load("functs")`.

`vers (x)` [Função]
Retorna o *sinus versus* `1 - cos (x)`.
Para usar essa função escreva primeiramente `load("functs")`.

`covers (x)` [Função]
Retorna o *sinus versus* do complemento `1 - sin (x)`.
Para usar essa função escreva primeiramente `load("functs")`.

`exsec (x)` [Função]
Retorna a parte externa da secante `sec (x) - 1`.
Para usar essa função escreva primeiramente `load("functs")`.

`hav (x)` [Função]
Retorna o semi-*sinus versus* `(1 - cos(x))/2`.
Para usar essa função escreva primeiramente `load("functs")`.

`combination (n, r)` [Função]
Retorna o número de combinações de n objectos tomados em grupos de r elementos.
Para usar essa função escreva primeiramente `load("functs")`.

`permutation (n, r)` [Função]
Retorna o número de permutações de r objectos seleccionados de um conjunto de n objectos.
Para usar essa função escreva primeiramente `load("functs")`.

65.2.4 Package ineq

O pacote `ineq` contém regras de simplificação para desigualdades.

Sessão exemplo:

```
(%i1) load("ineq")$
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
(%i2) a>=4;  /* uma desigualdade exemplo */
(%o2)                                a >= 4
(%i3) (b>c)+%; /* adiciona uma segunda e estrita desigualdade */
(%o3)                            b + a > c + 4
(%i4) 7*(x<y); /* multiplica por um número positivo */
(%o4)                              7 x < 7 y
```

```
(%i5) -2*(x>=3*z); /* multiplica por um número negativo */
(%o5)                            - 2 x <= - 6 z
(%i6) (1+a^2)*(1/(1+a^2)<=1); /* Maxima sabe que 1+a^2 > 0 */
                                              2
(%o6)                               1 <= a  + 1
(%i7) assume(x>0)$ x*(2<3); /* assumindo x>0 */
(%o7)                                2 x < 3 x
(%i8) a>=b; /* outa desigualdade */
(%o8)                                 a >= b
(%i9) 3+%; /* adiciona alguma coisa à desigualdade imediatamente acima */
(%o9)                              a + 3 >= b + 3
(%i10) %-3; /* retirando essa alguma coisa */
(%o10)                                a >= b
(%i11) a>=c-b; /* ainda outra desigualdade */
(%o11)                               a >= c - b
(%i12) b+%; /* adiciona b a ambos os lados da desigualdade */
(%o12)                               b + a >= c
(%i13) %-c; /* subtrai c de ambos os lados */
(%o13)                           - c + b + a >= 0
(%i14) -%;  /* multiplica por -1 */
(%o14)                            c - b - a <= 0
(%i15) (z-1)^2>-2*z; /* determinando a verdade de uma assertiva */
                                       2
(%o15)                           (z - 1)  > - 2 z
(%i16) expand(%)+2*z; /* expandindo essa assertiva e adicionado 2*z a ambos os lados *
                                     2
(%o16)                               z  + 1 > 0
(%i17) %,pred;
(%o17)                                 true
```

Seja cuidadoso com o uso dos parêntesis em torno de desigualdades: quando o utilizador digita `(A > B) + (C = 5)` o resltado é `A + C > B + 5`, mas `A > B + C = 5` é um erro de sintaxe, e `(A > B + C) = 5` é alguma coisa completametne diferente.

Faça `disprule (all)` para ver uma lista completa das definições de regras.

O utilizador será questionado se o Maxima for incapaz de decidir o sinal de uma quantidade multiplicando uma desigualdade.

O mais comum recurso estranho é ilustrado por:

```
(%i1) eq: a > b;
(%o1)                                  a > b
(%i2) 2*eq;
(%o2)                                2 (a > b)
(%i3) % - eq;
(%o3)                                  a > b
```

Outro problema é 0 vezes uma desigualdade; o padrão para isso acontecer é 0 ter sido colocado à esquerda sozinho. Contudo, se digitar `X*`*`some_inequality`* e Maxima perguntar sobre o sinal de `X` e responder `zero` (ou `z`), o programa retorna `X*`*`some_inequality`* e não

utiliza a informação que `X` é 0. Pode usar `ev (%, x: 0)` em casos semelhantes a esse; a base de dados irá somente ser usada para propósitos de comparação em decisões, e não para o propósito de avaliação de `X`.

O utilizador pode notar uma resposta lenta quando esse pacote é disponibilizado para uso, como o simplificador é forçado a examinar mais regras do precisaria sem esse pacote, então pode desejar remover essas regras após fazer uso delas. Faça `kill (rules)` para eliminar todas as regras (incluindo qualquer regra que possa ter definido); ou pode ser mais selectivo eliminando somente algumas delas; ou use `remrule` sobre uma regra específica.

Note que se disponibilizar esse pacote para ser usado, após definir suas próprias regras, irá sobrescrever as suas regras que possuirem nomes identicos a nomes contidos nas regras do pacote. As regras no pacote são: `*rule1`, ..., `*rule8`, `+rule1`, ..., `+rule18`, e deve colocar o nome de regra entre aspas duplas ao referir-se a eles, como em `remrule ("+", "+rule1")` para especificamente remover a primeira regra sobre `"+"` ou `disprule ("*rule2")` para mostrar a definição da segunda regra multiplicativa.

65.2.5 Package rducon

`reduce_consts (`*expr*`)` [Função]

Substitui subexpressões constantes de *expr* com construída com átomos constantes, gravando a definição de todas essas constantes construídas na lista de equações `const_eqns`, e retornando a expressão modificada *expr*. Essas partes de *expr* são constantes que retornam `true` quando operadas por meio da função `constantp`. Consequêntemente, antes de usar `reduce_consts`, se pode fazer

```
declare ([objecto que vai receber a propriedade constante], constant)$
```

para escolher a base de dados das quantidades constantes ocorrendo em suas expressões.

Se está a planear gerar saídas em Fortran após esses cálculos simbólicos, uma das primeiras secções de código pode ser o cálculo de todas as constantes. Para gerar esse segmento de código, faça

```
map ('fortran, const_eqns)$
```

Variables como `const_eqns` que afectam `reduce_consts` são:

`const_prefix` (valor padrão: `xx`) é a sequência de caracteres usada para prefixar todos os símbolos gerados por `reduce_consts` para representar subexpressões constantes.

`const_counter` (valor padrão: 1) é o índice inteiro usado para gerar símbolos únicos para representar cada subexpressão constante emcontrada por `reduce_consts`.

`load ("rducon")` torna essa função disponível para uso. `demo (rducon)` faz uma demonstração dessa função.

65.2.6 Pacote scifac

`gcfac (`*expr*`)` [Função]

`gcfac` função de factorização que tenta aplicar a mesma heurística que cientístas aplicam em tentativas de fazer expressões extremamente simples. `gcfac` está limitada a factorizações monomiais. Para um somatório, `gcfac` faz o seguinte:

1. Factores sobre os inteiros.

2. Coloca em evidência o maior expoente de termos ocorrendo como coeficientes, independentemente da complexidade dos termos.
3. Usa (1) e (2) em factorizações de pares de termos adjascentes.
4. Repetidamente e recursivamente aplica essas técnicas até que a expressão não mais mude.

O item (3) não necessáriamente faz uma tarefa óptima factorização par a par devido à dificuldade combinatória natural de encontrar qual de todas dos possíveis rearranjos de pares retorna o mais compacto resultado de factorização de um par.

`load ("scifac")` torna essa função disponível para uso. `demo (scifac)` faz uma demonstração dessa função.

65.2.7 Pacote sqdnst

`sqrtdenest` (*expr*) [Função]

Desaninha `sqrt` de simples, numérico, binômios de raízes irracionais de números racionais , onde for possível. 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
```

Algumas vezes isso ajuda na hora de aplicar `sqrtdenest` mais que uma vez, sobre coisas como `(19601-13860 sqrt(2))^(7/4)`.

`load ("sqdnst")` Torna essa função disponível para uso.

66 solve_rec

/solve_rec.texi/1.7/Tue Jan 16 15:15:10 2007//

66.1 Introdução a solve_rec

`solve_rec` é um pacote para resolver recorrências lineares com coeficientes polinomiais.

Um ficheiro de domostração está disponivel com `demo(solve_rec);`.

Exemplo:

```
(%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)
```

66.2 Definições para solve_rec

`reduce_order` (*rec*, *sol*, *var*) [Função]

Reduz a ordem de recorrência linear *rec* quando uma solução particular *sol* for conhecida. A recorrência reduzida pode ser usada para pegar outras soluções.

Exemplo:

```
(%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
                ====         n
                \       (- 1)
          %k  n  >     -------- + %k  n
            2   /     (n + 1)!     1
                ====
                n = 0
```

`simplify_products` [Variável de opção]

Valor por omissão: `true`

Se `simplify_products` for `true`, `solve_rec` irá tentar simplificar produtos no resultado.

Veja também: `solve_rec`.

`simplify_sum` (*expr*) [Função]

Tenta simplificar todos os somatórios que aparecem na *expr* para uma forma a mais simplificada possível.

`simplify_sum` usa os algoritmos de Gosper e de Zeilberger para simplificar somatórios.

Para usar essa função primeiramente chame o pacote `simplify_sum` com `load("simplify_sum")`.

Exemplo:

```
(%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*]) [Função]

Encontra soluções hipergeométricas para a recorrência linear *eqn* com coeficientes polinomiais na variável *var*. Argumentos opcionais *init* são as condições iniciais.

`solve_rec` pode resolver recorrências lineares com coeficientes constantes, encontrando soluções hipergeométricas para recorrências lineares homogêneas com coeficientes polinomiais, soluções racionais para recorrências lineares com coeficientes polinomiais e pode resolver recorrências do tipo de Ricatti.

Note que o tempo de execução do algoritmo usado para encontrar soluções hipergeométricas aumenta exponencialmente com o grau do coeficiente lider e guia.

Para usar essa função primeiramente chame o pacote `solve_rec` com `load("solve_rec");`.

Exemplo de recorrência linear com coeficientes constantes:

```
(%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
```

Exemplo de recorrência linear com coeficientes polinomiais:

```
(%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
```

Exemplo de recorrência do tipo de Ricatti:

```
(%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
```

Veja também: `solve_rec_rat`, `simplify_products`, e `product_use_gamma`.

`solve_rec_rat` (*eqn*, *var*, [*init*]) [Função]

Encontra soluções racionais para recorrências lineares. Veja solve_rec para uma descrição dos argumentos.

Para usar essa função primeirametne chame o pacote `solve_rec` com `load("solve_rec");`.

Exemplo:

```
(%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)
```

Veja também: `solve_rec`.

`product_use_gamma` [Variável de opção]

Valor por omissão: `true`

Quando simplificando produtos, `solve_rec` introduz a função gama dentro da expressão se `product_use_gamma` for `true`.

Veja também: `simplify_products`, `solve_rec`.

`summand_to_rec` (*summand*, *k*, *n*) [Função]
`summand_to_rec` (*summand*, [*k*, *lo*, *hi*], *n*) [Função]

Retorna a recorrência satisfeita pelo somatório

```
 sup
====
\
 >     x
/
====
k = inf
```

onde x é hipergeométrico em *k* e *n*. SE *inf* e *sup* forem omitidos, são assumidos como sendo `inf` = `-inf` e `sup` = `inf`.

Para usar essa função primeiro chame o pacote `simplify_sum` com `load("simplify_sum")`.

Exemplo:

```
(%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
```

67 stats

67.1 Introdução a stats

O pacote `stats` contém um conjunto de procedimentos de inferência clássica estatística e procedimentos de teste.

Todas essas funções retornam um objecto do Maxima chamado `inference_result` que contém os resultados necessários para inferências de manipulação e tomada de decisões.

A variável global `stats_numer` controla se resultados são mostrados em ponto flutuante ou simbólico e no formato racional; seu valor padrão é `true` e os resultados são retornados no formato de ponto flutuante.

O pacote `descriptive` contém alguns utilitários para manipular estruturas de dados (listas e matrizes); por exemplo, para extrair subamostras. O pacote `descriptive` também contém alguns exemplos sobre como usar o pacote `numericalio` para ler dados a partir de ficheiro no formato texto plano. Veja `descriptive` e `numericalio` para maiores detalhes.

O pacote `stats` precisa dos pacotes `descriptive`, `distrib` e `inference_result`.

Para comentários, erros ou sugestões, por favor contate o autor em

'mario AT edu DOT xunta DOT es'.

67.2 Definições para inference_result

`inference_result` (*título*, *valores*, *números*) [Função]

Constrói um objecto `inference_result` do tipo retornado pelas funções stats. O argumento *título* é uma sequência de caracteres do Maxima co o nome do procedimento; *valores* é uma lissta com elementos da forma `símbolo = valor` e *números* é uma lista com números inteiros positivos no intervalo de um para `length(`*valores*`)`, indicando que valores serão mostrados por padrão.

Exemplo:

Este é um exemplo que mostras os resultados associados a um retángulo. O título deste bojeto é a sequência de caraceteres `"Retângulo"`, o qual armazena cinco resultados, a saber, `'base`, `'altura`, `'diagonal`, `'área` y `'perímetro`, porém só mostra o primeiro, segundo, quinto e quarto resultado. O resultado `'diagonal` também é armazenado neste objecto, no entanto não é mostrado por padrão; para se ter acesso a este valor, faz-se uso da função `take_inference`.

```
(%i1) load("inference_result")$
(%i2) b: 3$ h: 2$
(%i3) inference_result("Retângulo",
                       ['base=b,
                        'altura=h,
                        'diagonal=sqrt(b^2+h^2),
                        'área=b*h,
                        'perímetro=2*(b+h)],
                       [1,2,5,4] );
|   Retângulo
```

```
                              |
                              |    base = 3
                              |
(%o3)                         |   altura = 2
                              |
                              | perímetro = 10
                              |
                              |    area = 6
(%i4) take_inference('diagonal,%);
(%o4)                          sqrt(13)
```

Veja também `take_inference`.

`inferencep` (*obj*) [Função]

Retorna `true` ou `false`, dependendo se *obj* é um objecto `inference_result` ou não.

`items_inference` (*obj*) [Função]

Retorna uma lista com os nomes dos itens em *obj*, que devem ser um objecto `inference_result`.

Exemplo:

O objecto `inference_result` armazena dois valores, a saber `'pi` e `'e`, mas somente o segundo é mostrado. A função `items_inference` retorna os nomes de todos os itens, não importa se eles são ou não mostrados.

```
(%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*) [Função]
`take_inference` (*nome*, *obj*) [Função]
`take_inference` (*lista*, *obj*) [Função]

Retorna o *n*-ésimo valor armazenado em *obj* se *n* for um inteiro positivo, ou o item chamado *nome* se esse for o nome de um item. Se o primeiro argumento for uma lista de números e/ou símbolos, a função `take_inference` retorna uma lista com os resultados correspondentes.

Exemplo:

Fornece um objecto `inference_result`, a função `take_inference` é chamada com o objectivo de extrair alguma informação armazenada nesse objecto.

```
(%i1) load("inference_result")$
(%i2) b: 3$ h: 2$
(%i3) sol: inference_result("Retângulo",
                            ['base=b,
                             'altura=h,
                             'diagonal=sqrt(b^2+h^2),
```

```
                                    'area=b*h,
                                    'perímetro=2*(b+h)],
                                   [1,2,5,4] );
                               |   Retângulo
                               |
                               |    base = 3
                               |
(%o3)                          |   altura = 2
                               |
                               | perímetro = 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]
```

Veja também `inference_result` e `take_inference`.

67.3 Definições para stats

`stats_numer` [Variável de opção]

Valor por omissão: `true`

Se `stats_numer` for `true`, funções de inferência estatística retornam seus resultados em números com ponto flutuante. Se `stats_numer` for `false`, resultados são fornecidos em formato simbólico e racional.

`test_mean` (*x*) [Função]
`test_mean` (*x*, *opção_1*, *opção_2*, ...) [Função]

Esse é o teste-t de média. O argumento x é uma lista ou uma matriz coluna contendo uma amostra unidimensional. `test_mean` tamb;em executa um teste assintótico baseado no *Teorema do Limite Central* se a opção `'asymptotic` for `true`.

Opções:

- `'mean`, o valor padrão é `0`, é o valor da média a ser verificado.
- `'alternative`, o valor padrão é `'twosided`, é a hipótese alternativa; valores válidos são: `'twosided`, `'greater` e `'less`.
- `'dev`, o valor padrão é `'unknown`, corresponde ao valor do desvio padrão quando esse valor de desvio padrão for conhecido; valores válidos são: `'unknown` ou uma expressão positiva.
- `'conflevel`, o valor padrão é `95/100`, nível de confidência para o intervalo de confidência; deve ser uma expressão que toma um valor em (0,1).

- `'asymptotic`, o valor padrão é `false`, indica se `test_mean` exeecuta um teste-t exato ou um teste assintótico baseando-se no *Teorema do Limite Central*; valores válidos são `true` e `false`.

A saída da função `test_mean` é um objecto `inference_result` do Maxima mostrando os seguintes resultados:

1. `'mean_estimate`: a média da amostra.
2. `'conf_level`: nível de confidência seleccionado pelo utilizador.
3. `'conf_interval`: intervalo de confidência para a média da população.
4. `'method`: procedimento de inferência.
5. `'hypotheses`: hipótese do nulo e hipótese alternativa a ser testada.
6. `'statistic`: valor da amostra estatística a ser usado para testar a hipótese do nulo.
7. `'distribution`: distribuição da amostra estatística, juntamente com seus parâmetro(s).
8. `'p_value`: valores de p do teste.

Exemplos:

Executa um teste-t exato com variância desconhecida. A hipótese do nulo é H_0 : $mean = 50$ contra a alternativa unilatera $H_1 : mean < 50$; conforme os resultados, o valor de p é muito grande, não existem evidências paa rejeitar 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
```

Nesta ocasião Maxima executa um testte assintótico, baseado no *Teorema do Limite Central*. A hipótese do nulo é H_0 : $equal(mean, 50)$ contra a alternativa de duas vias $H_1 : notequal(mean, 50)$; conforme os resultados, o valor de p é muito pequeno,

H_0 pode ser rejeitado em favor da alternativa H_1. Note que, como indicado pela componente `Method`, esse procedimento pode ser aplicado a grandes amostras.

```
(%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*) [Função]
`test_means_difference` (*x1*, *x2*, *opção_1*, *opção_2*, ...) [Função]

Esse é o teste-t de diferença de médias entre duas amostras. Os argumentos *x1* e *x2* são listas ou matrizes colunas contendo duas amostras independentes. No caso de diferentes variâncias desconhecidas (veja opções `'dev1`, `'dev2` e `'varequal` abaixo), os graus de liberdade são calculados por meio da aproximação de Welch. `test_means_difference` também executa um teste assintótico baseado no *Teorema do Limite Central* se a opção `'asymptotic` for escolhida para `true`.

Opções:

-
- `'alternative`, o valor padrão é `'twosided`, é a hipótese alternativa; valores válidos são: `'twosided`, `'greater` e `'less`.
- `'dev1`, o valor padrão é `'unknown`, é o valor do desvio padrão da amostra *x1* quando esse desvio for conhecido; valores válidos são: `'unknown` ou uma expressão positiva.
- `'dev2`, o valor padrão é `'unknown`, é o valor do desvio padrão da amostra *x2* quando esse desvio for conhecido; valores válidos são: `'unknown` ou uma expressão positiva.
- `'varequal`, o valor padrão é `false`, se variâncias podem serem consideradas como iguais ou não; essa opção tem efeito somente quando `'dev1` e/ou `'dev2` forem `'unknown`.

- `'conflevel`, o valor padrão é `95/100`, nível de confidência para o intervalo de confidência; deve ser uma expressão que toma valores em (0,1).

  Nota de Tradução: (0,1) representa intervalo aberto.

- `'asymptotic`, o valor padrão é `false`, indica se `test_means_difference` executa um teste-t exato ou um teste assíntótico baseando-se no *Teorema do Limite Central*; valores válidos são `true` e `false`.

A saída da função `test_means_difference` é um objecto `inference_result` do Maxima mostrando os seguintes resultados:

1. `'diff_estimate`: a diferença de médias estimadas.
2. `'conf_level`: nível de confidência seleccionado pelo utilizador.
3. `'conf_interval`: intervalo de confidência para a diferença de médias.
4. `'method`: procedimento de inferência.
5. `'hypotheses`: a hipótese do nulo e a hipótese alternativa a serem testadas.
6. `'statistic`: valor da amostra estatística usado para testar a hipótese do nulo.
7. `'distribution`: distribuição da amostra estatística, juntamente com seu(s) parâmetro(s).
8. `'p_value`: valor de p do teste.

Exemplos:

A igualdade de médias é testada com duas pequenas amostras x e y, contra a alternativa $H_1 : m_1 > m_2$, sendo m_1 e m_2 as médias das populações; variâncias são desconhecidas e supostamente admitidas para serem diferentes.

```
(%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
```

O mesmo teste que antes, mas agora as variâncias são admitidas serem supostamente iguais.

```
(%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*) [Função]
`test_variance` (*x*, *opção_1*, *opção_2*, ...) [Função]

Esse é o teste da variância *chi^2*. O argumento *x* é uma lista ou uma matriz coluna contendo uma amostra unidimensional tomada entre a população normal.

Opções:

- `'mean`, o valor padrão é `'unknown`, é a média da população, quando for conhecida.
- `'alternative`, o valor padrão é `'twosided`, é a hipótese alternativa; valores válidos são: `'twosided`, `'greater` e `'less`.
- `'variance`, o valor padrão é `1`, isso é o valor (positivo) da variância a ser testado.
- `'conflevel`, o valor padrão é `95/100`, nível de confidência para o intervalo de confidência; deve ser uma expressão que toma valores em (0,1).

A saída da função `test_variance` está no objecto `inference_result` do Maxima mostrando os seguintes resultados:

1. `'var_estimate`: a variância da amostra.
2. `'conf_level`: nível de confidência seleccionado pelo utilizador.
3. `'conf_interval`: intervalo de confidência para a variância da população.
4. `'method`: procedimento de inferência.
5. `'hypotheses`: a hipótese do nulo e a hipótese alternativa a serem testadas.
6. `'statistic`: valor da amostra estatística usado para testar a hipótese do nulo.

7. `'distribution`: distribuição da amostra estatística, juntamente com seu parâmetro.
8. `'p_value`: o valor de p do teste.

Exemplos:

Isso é testado se a variância de uma população com média desconhhecida for igual ou maior que 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*) [Função]
`test_variance_ratio` (*x1*, *x2*, *opção_1*, *opção_2*, ...) [Função]

Isso é o teste F da razão de variância para duas populações normais. Os argumentos *x1* e *x2* são listas ou matrizes colunas contendo duas amostras independentes.

Opções:

- `'alternative`, o valor padrão é `'twosided`, é a hipótese alternativa; valores válidos são: `'twosided`, `'greater` e `'less`.
- `'mean1`, o valor padrão é `'unknown`, quando for conhecida, isso é a média da população da qual *x1* foi tomada.
- `'mean2`, o valor padrão é `'unknown`, quando for conhecida, isso é a média da população da qual *x2* foi tomada.
- `'conflevel`, o valor padrão é `95/100`, nível de confidência para o intervalo de confidência da razão; deve ser uma expressão que tome valores em (0,1).

A saída da função `test_variance_ratio` é um objecto `inference_result` do Maxima mostrando os seguintes resultados:

1. `'ratio_estimate`: a razão de variância da amostra.
2. `'conf_level`: nível de confidência seleccionado pelo utilizador.

3. `'conf_interval`: intervalo de confidência para a razão de variância.
4. `'method`: procedimento de inferência.
5. `'hypotheses`: a hipótese do nulo e a hipótese alternativa a serem testadas.
6. `'statistic`: valor da amostra estatística usado para testar a hipótese do nulo.
7. `'distribution`: distribuição da amostra estatística, juntamente com seus parâmetros.
8. `'p_value`: o valor de p do teste.

Exemplos:

a igualdade das variâncias de duas populações normais é verificado contra a alternativa que a primeira é maior que a segunda.

```
(%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*) [Função]

`test_sign` (*x*, *opção_1*, *opção_2*, ...) [Função]

Esse é o teste de sinal não paramétrico para a mediana de uma população contínua. O argumento *x* é uma lista ou uma matriz coluna contendo uma amostra unidimensional.

Opções:

- `'alternative`, o valor padrão é `'twosided`, é a hipótese alternativa; valores válidos são: `'twosided`, `'greater` e `'less`.
- `'median`, o valor padrão é `0`, é o valor da mediana a ser verificado.

A saída da função `test_sign` é um objecto `inference_result` do Maxima mostrando os seguintes resultados:

1. `'med_estimate`: a mediana da amostra.
2. `'method`: procedimento de inferência.

3. `'hypotheses`: a hipótese do nulo e a hipótese alternativa a serem testadas.
4. `'statistic`: valor da amostra estatística usada para testar a hipótese do nulo.
5. `'distribution`: distribuição da amostra estatística, juntamente com seu(s) parâmetro(s).
6. `'p_value`: o valor de p do teste.

Exemplos:

Verifica se a população da qual a amostra foi tomada tem mediana 6, contra a alternativa $H_1 : median > 6$.

```
(%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`) [Função]
`test_signed_rank` (*x*, *opção_1*, *opção_2*, ...) [Função]

Esse é o teste de ranque sinalizado de Wilcoxon para fazer inferências sobre a mediana de uma população contínua. O argumento x é uma lista ou uma matriz coluna contendo uma amostra unidimensional. Executa uma aproximação normal se o tamanho da amostra for maior que 20, ou se existirem zeros ou houverem empates.

Veja também `pdf_rank_test` e `cdf_rank_test`.

Opções:

- `'median`, o valor padrão é `0`, é o valor da mediana a ser verificado.
- `'alternative`, o valor padrão é `'twosided`, é a hipótese alternativa; valores válidos são: `'twosided`, `'greater` e `'less`.

A saída da função `test_signed_rank` é um objecto `inference_result` do Maxima com os seguintes resultados:

1. `'med_estimate`: a mediana da amostra.
2. `'method`: procedimento de inferência.
3. `'hypotheses`: a hipótese do nulo e a hipótese alternativa a serem testadas.
4. `'statistic`: valor da amostra estatística usado para testar a hipótese do nulo.
5. `'distribution`: distribuição da amostra estatística, juntamente com seu(s) parâmetro(s).

6. `'p_value`: o valor de p do teste.

Exemplos:

Verifica a hipótese do nulo $H_0 : median = 15$ contra a alternativa $H_1 : median > 15$. Esse é um teste exato, ua vez que não exite empates.

```
(%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
```

Verifica a hipótese do nulo H_0 : $equal(median, 2.5)$ contra a alternativa H_1 : $notequal(median, 2.5)$. Esse é um teste aproximado, uma vez que ocorrem empates.

```
(%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*) [Função]

`test_rank_sum` (*x1*, *x2*, *opção_1*) [Função]

Esse é o teste de Wilcoxon-Mann-Whitney para comparação das medianas de duas populações contínuas. Os primeiros dois argumentos *x1* e *x2* são listas ou matrizes colunas com os dados de duas amostras independentes. Executa aproximação normal se quaisquer dos tamanhos de amostra for maior que 10, ou se houverem empates.

Opção:

- `'alternative`, o valor padrão é `'twosided`, é a hipótese alternativa; valores válidos são: `'twosided`, `'greater` e `'less`.

A saída da função `test_rank_sum` é um objecto `inference_result` do Maxima com os seguintes resultados:

1. `'method`: procedimento de inferência.
2. `'hypotheses`: a hipótese do nulo e a hipótese alternativa a serem testadas.
3. `'statistic`: valor da amostra estatística usada para testar a hipótese do nulo.
4. `'distribution`: distribuição da amostra estatística, juntamente com seus parâmetros.
5. `'p_value`: o valor de p do teste.

Exemplos:

Verifica se populações possuem medianas similares. Tamanhos de amotra são pequenos e é feito um teste exato.

```
(%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
```

Agora, com grandes amostras e empates, o procedimento faz aproximação norma. A hipótese alternativa é $H_1 : median1 < median2$.

```
(%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*) [Função]

Teste de Shapiro-Wilk para normalidade. O argumento *x* é uma lista de números, e o tamanho da amostra deve ser maior que 2 e menor ou igua a 5000, de outra forma, a função `test_normality` sinaliza com um erro.

Referência:

[1] Algorithm AS R94, Applied Statistics (1995), vol.44, no.4, 547-551

A saída da função `test_normality` é um objecto `inference_result` do Maxima com os seguintes resultados:

1. `'statistic`: valor do W estatístico.
2. `'p_value`: valor de p sob a hipótese de normalidade.

Exemplos:

Verifica a normalidade de uma população, baseada em uma amostra de tamanho 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*) [Função]

`simple_linear_regression` (*x opção_1*) [Função]

Regressão linear simples, $y_i = a + bx_i + e_i$, onde os e_i são $N(0, sigma)$ variáveis aleatórias independentes. O argumento *x* deve ser uma matriz de duas colunas ou uma lista de pares.

Opções:

- `'conflevel`, o valor padrão é `95/100`, nível de confidência para o intervalo de confidência; isso deve ser uma expressão que tome valores em (0,1).
- `'regressor`, o valor padrão é `'x`, nome da variável independente.

A saída da função `simple_linear_regression` é um objecto `inference_result` do Maxima com os seguintes resultados:

1. `'model`: a equação ajustada. Útil para fazer novas previsões. Veja exemplos abaixo.
2. `'means`: média de duas variáveis pseudo-aleatórias.
3. `'variances`: variâncias de ambas as variáveis.
4. `'correlation`: coeficiente de correlação.
5. `'adc`: coeficiente de determinação ajustado.
6. `'a_estimation`: estimador do parâmetro *a*.
7. `'a_conf_int`: intervalo de confidência do parâmetro *a*.

8. `'b_estimation`: estimador do parâmetro b.
9. `'b_conf_int`: intervalo de confidência do parâmetro b.
10. `'hypotheses`: a hipótese do nulo e a hipótese alternativa sobre o parâmetro b.
11. `'statistic`: valor da amostra estatística usado para testar a hipótese do nulo.
12. `'distribution`: distribuição da amostra estatística, juntamente com seu parâmetro.
13. `'p_value`: o valor de p do teste sobre b.
14. `'v_estimation`: estimador de variância imparcial, ou variância residual.
15. `'v_conf_int`: intervalo de confidência da variância.
16. `'cond_mean_conf_int`: intervalo de confidência paa a média condicionada. Veja exemplos abaixo.
17. `'new_pred_conf_int`: intervalo de confidência para uma nova previsão. Veja exemplos abaixo.
18. `'residuals`: lista de pares (previsão, resíduo), ordenados em relação às previsões. Útil para achar o melhor da análise de ajuste. Veja exemplos abaixo.

Somente os itens 1, 4, 14, 9, 10, 11, 12, e 13 acima, nessa ordem, são mostrados por padrão. Os restantes escondem-se até que o utilizador faça uso de funções `items_inference` e `take_inference`.

Exemplo:

Ajustando um modelo linear para uma amostras de duas variáveis. A entrada `%i4` monta p gráfico da amostra junto com a linha de regressão; a entrada `%i5` calcula `y` dado `x=113`; a média e o intervalo de confidência para uma nova previsão quando `x=113` são também calculados.

```
(%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]
```

67.4 Definições para distribuições especiais

`pdf_signed_rank` (*x*, *n*) [Função]

Função densidade de probabilidade da distribuição exacta da estatística do rank sinalizado. O argumento *x* é um número real e *n* um inteiro positivo.

Veja também `test_signed_rank`.

`cdf_signed_rank` (*x*, *n*) [Função]

Função de densidade cumulativa da distribuição exacta da estatística do rank sinalizado. O argumento *x* é um número real e *n* um inteiro positivo.

Veja também `test_signed_rank`.

`pdf_rank_sum` (*x*, *n*, *m*) [Função]

Função densidade de probabilidade da distribuição exacta da estatística do somatório do rank. O argumento *x* é um número real e *n* e *m* são ambos inteiros positivos.

Veja também `test_rank_sum`.

`cdf_rank_sum` (*x*, *n*, *m*) [Função]

Função de densidade cumulativa da distribuição exacta da estatística do somatório do rank. O argumento *x* é um número real e *n* e *m* são ambos inteiro positivos.

Veja também `test_rank_sum`.

68 stirling

68.1 Definições para stirling

`stirling (z,n)` [Função]

Substitui `gamma(x)` pela fórmula de Stirling $O(1/x^(2n-1))$. Quando n for um inteiro estritamente negativo, sinaliza um erro.

Referência: Abramowitz & Stegun, " Handbook of mathematical functions", 6.1.40.

Exemplos:

```
(%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
```

A função `stirling` conhece a diferença entre a variável *gamma* e a função `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
```

Para usar essa função escreva primeiro `load("stirling")`.

69 stringproc

69.1 Introdução a manipulação de sequências de caracteres

O ficheiro `stringproc.lisp` amplia a compatibilidade do Maxima de trabalhar com sequências de caracteres e adiciona algumas funções úteis a entrada e saída de dados.

Para esclarecimentos e erros por favor mande um e-mail para van.nek at arcor.de .

Para disponibilizar `stringproc.lisp` para uso digite `load("stringproc");`.

No Maxima uma sequência de caracteres é facilmente contruída digitando "texto" (qualquer texto desejado entre aspas duplas). Note que sequências de caracteres do Maxima não são sequências de caracteres do Lisp e vice-versa. Testes podem ser concluídos com `stringp` e `lstringp`. Se por alguma razão tiver um valor, que é uma sequência de caracteres do Lisp, talvez quando estiver usando a função `sconcat` do Maxima, poderá converter via `sunlisp`.

```
(%i1) load("stringproc")$
(%i2) m: "text";
(%o2)                          text
(%i3) [stringp(m),lstringp(m)];
(%o3)                      [true, false]
(%i4) l: sconcat("text");
(%o4)                          text
(%i5) [stringp(l),lstringp(l)];
(%o5)                      [false, true]
(%i6) stringp( sunlisp(l) );
(%o6)                          true
```

Todas as funções em `stringproc.lisp`, que retornarem sequências de caracteres, retornam sequências de caracteres do Maxima.

Caracteres são introduzidos como sequências de caracteres do Maxima de comprimento 1. Com certeza, esses caracteres não são caracteres do Lisp. Testes podem ser realizados com `charp` (`lcharp` e conversões do Lisp para o Maxima com `cunlisp`).

```
(%i1) load("stringproc")$
(%i2) c: "e";
(%o2)                            e
(%i3) [charp(c),lcharp(c)];
(%o3)                      [true, false]
(%i4) supcase(c);
(%o4)                            E
(%i5) charp(%);
(%o5)                          true
```

Novamente, todas as funções em `stringproc.lisp`, que retornam caracteres, retornam caracteres do Maxima. devido a esse facto, que os caracteres introduzidos são sequências de caracteres de comprimento 1, pode usar muitas das funções de sequência de caracteres também para caracteres. Como visto, `supcase` é um exemplo.

É importante saber, que o primeiro caractere em uma sequência de caracteres do Maxima éstá na posição 1. Isso é designado devido ao facto de o primeiro elemento em uma lista do

Maxima está na posição 1 também. Veja definições de `charat` e de `charlist` para obter exemplos.

Em aplicações fnções de sequência de caractere são muitas vezes usadas quando estamos trabalhando com ficheiros. Poderá encontrará algumas funções úteis de fluxo e de impressão em `stringproc.lisp`. O seguinte exemplo mostra algumas das funções aqui introduzidas no trabalho.

Exemplo:

`openw` retorna um fluxo de saída para um ficheiro, `printf` então permite escrita formatada para esse ficheiro. Veja `printf` para detalhes.

```
(%i1) load("stringproc")$
(%i2) s: openw("E:/file.txt");
(%o2)                     #<output stream E:/file.txt>
(%i3) for n:0 thru 10 do printf( s, "~d ", fib(n) );
(%o3)                                  done
(%i4) printf( s, "~%~d ~f ~a ~a ~f ~e ~a~%",
              42,1.234,sqrt(2),%pi,1.0e-2,1.0e-2,1.0b-2 );
(%o4)                                  false
(%i5) close(s);
(%o5)                                  true
```

Após fechar o fluxo pode abrí-lo novamente, dessa vez com direção de entrada. `readline` retorna a linha completa como uma sequência de caracteres. O pacote `stringproc` agora oferece muitas funções para manipulação de sequências de caracteres. A troca de indicações/fichas pode ser realizada por `split` ou por `tokens`.

```
(%i6) s: openr("E:/file.txt");
(%o6)                      #<input stream E:/file.txt>
(%i7) readline(s);
(%o7)                      0 1 1 2 3 5 8 13 21 34 55
(%i8) line: readline(s);
(%o8)                42 1.234 sqrt(2) %pi 0.01 1.0E-2 1.0b-2
(%i9) list: tokens(line);
(%o9)           [42, 1.234, sqrt(2), %pi, 0.01, 1.0E-2, 1.0b-2]
(%i10) map( parsetoken, list );
(%o10)           [42, 1.234, false, false, 0.01, 0.01, false]
```

`parsetoken` somente analiza números inteiros e em ponto flutuante. A análise de símbolos ou grandes números em ponto flutuante precisa de `parse_string`, que pode ser disponibilizada para uso através de `eval_string.lisp`.

```
(%i11) load("eval_string")$
(%i12) map( parse_string, list );
(%o12)           [42, 1.234, sqrt(2), %pi, 0.01, 0.01, 1.0b-2]
(%i13) float(%);
(%o13) [42.0, 1.234, 1.414213562373095, 3.141592653589793, 0.01, 0.01, 0.01]
(%i14) readline(s);
(%o14)                                 false
(%i15) close(s)$
```

`readline` retorna `false` quado o fim de ficheiro acontecer.

69.2 Definições para entrada e saída

Exemplo:

```
(%i1) load("stringproc")$
(%i2) s: openw("E:/file.txt");
(%o2)                     #<output stream E:/file.txt>
(%i3) control:
"~2tAn atom: ~20t~a~%~2tand a list: ~20t~{~r ~}~%~2tand an integer: ~20t~d~%"$
(%i4) printf( s,control, 'true,[1,2,3],42 )$
(%o4)                                  false
(%i5) close(s);
(%o5)                                  true
(%i6) s: openr("E:/file.txt");
(%o6)                     #<input stream E:/file.txt>
(%i7) while stringp( tmp:readline(s) ) do print(tmp)$
  An atom:          true
  and a list:       one two three
  and an integer:   42
(%i8) close(s)$
```

`close` (*fluxo*) [Função]

Fecha *fluxo* e retorna `true` se *fluxo* tiver sido aberto anteriormente.

`flength` (*fluxo*) [Função]

Retorna o número de elementos em *fluxo*.

`fposition` (*fluxo*) [Função]
`fposition` (*fluxo*, *pos*) [Função]

Retorna a posição corrente em *fluxo*, se *pos* não está sendo usada. Se *pos* estiver sendo usada, `fposition` escolhe a posição em *fluxo*. *pos* tem que ser um número positivo, o primeiro elemento em *fluxo* está na posição 1.

`freshline` () [Função]
`freshline` (*fluxo*) [Função]

escreve uma nova linha (em *fluxo*), se a posição actual não for um início de linha. Veja também `newline`.

`newline` () [Função]
`newline` (*fluxo*) [Função]

Escreve uma nova linha (para *fluxo*). Veja `sprint` para um exemplo de uso de `newline()`. Note que existem alguns casos, onde `newline()`não trabalha como esperado.

`opena` (*ficheiro*) [Função]

Retorna um fluxo de saída para *ficheiro*. Se um ficheiro já existente tiver sido aberto, `opena` anexa os elementos ao final do ficheiro.

`openr` (*ficheiro*) [Função]

Retorna um fluxo para *ficheiro*. Se *ficheiro* não existir, ele será criado.

`openw` (*ficheiro*) [Função]

Retorna um fluxo de saída para *ficheiro*. Se *ficheiro* não existir, será criado. Se um ficheiro já existente for aberto, `openw` modifica destrutivametne o *ficheiro*.

`printf` (*dest*, *seq_caracte*) [Função]
`printf` (*dest*, *seq_caracte*, *expr_1*, ..., *expr_n*) [Função]

Torna a função FORMAT do Lisp Comum disponível no Maxima. (Retirado de gcl.info: "format produces formatted output by outputting the caracteres of control-string string and observing that a tilde introduces a directive. The caractere 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.")

A seguinte descrição e oa exemplos podem fornecer uma idéia de uso de `printf`. Veja um referência de Lisp para maiores informações.

```
~%       nova linha
~&       novíssima line
~t       tabulação
~$       monetário
~d       inteiro decimal
~b       inteiro binário
~o       inteiro octal
~x       inteiro hexadecimal
~br      inteiro de base b
~r       soletra um inteiro
~p       plural
~f       ponto flutuante
~e       notação científica
~g       ~f ou ~e, dependendo  da magnitude
~a       como mostrado pela função print do Maxima
~s       sequências de caracteres entre "aspas duplas"
~~       ~
~<       justificação de texto, ~> terminador de justificação de texto
~(       conversão de caixa alta/baixa, ~) terminador de conversão de caixa
~[       selecção, ~] terminador de selecção
~{       iteração, ~} terminador de iteração
```

Por favor note que não existe especificador de formato para grandes números em ponto flutuante. Todavia grandes números em ponto flutuante podem simplesmente serem mostrados por meio da directiva `~a`. `~s` mostra as sequências de caracteres entre "aspas duplas"; pode evitar isso usando `~a`. Note que a directiva de selecção `~[` é indexada em zero. Também note que existem algumas directivas, que não trabalham no Maxima. Por exemplo, `~:[` falha.

```
(%i1) load("stringproc")$
(%i2) printf( false, "~a ~a ~4f ~a ~@r",
              "String",sym,bound,sqrt(12),144), bound = 1.234;
(%o2)                 String sym 1.23 2*sqrt(3) CXLIV
(%i3) printf( false,"~{~a ~}",["one",2,"THREE"] );
```

```
(%o3)                          one 2 THREE
(%i4) 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
(%i5) control: "~:(~r~) bird~p ~[is~;are~] singing."$
(%i6) printf( false,control, n,n,if n=1 then 0 else 1 ), n=2;
(%o6)                     Two birds are singing.
```

Se *dest* for um fluxo ou `true`, então `printf` retorna `false`. De outra forma, `printf` retorna uma sequência de caracteres contendo a saída.

`readline` (*fluxo*) [Função]

Retorna uma sequência de caracteres contendo os caracteres a partir da posição corrente em *fluxo* até o fim de linha ou *false* se o fim de linha do ficheiro for encontrado.

`sprint` (*expr_1*, ..., *expr_n*) [Função]

Avalia e mostra seus argumentos um após o outro 'sobre uma linha' iniciando na posição mais à esquerda. Os números são mostrados com o '-' à direita do número, e isso desconsidera o comprimento da linha. `newline()`, que pode ser chamada a partir de `stringproc.lisp` pode ser útil, se desejar colocar uma parada de linha intermédia.

```
(%i1) for n:0 thru 22 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 10946 17711
(%i2) load("stringproc")$
(%i3) 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
```

69.3 Definições para caracteres

`alphacharp` (*caractere*) [Função]

Retorna `true` se *caractere* for um caractere alfabético.

`alphanumericp` (*caractere*) [Função]

Retorna `true` se *caractere* for um caractere alfabético ou um dígito.

`ascii` (*int*) [Função]

Retorna o caractere correspondente ao código numérico ASCII *int*. (-1 < int < 256)

```
(%i1) load("stringproc")$
(%i2) 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` (*caractere_1*, *caractere_2*) [Função]

Retorna `true` se *caractere_1* e *caractere_2* forem os mesmos.

`cequalignore` (*caractere_1*, *caractere_2*) [Função]
como `cequal` mas ignora a caixa alta/baixa.

`cgreaterp` (*caractere_1*, *caractere_2*) [Função]
Retorna `true` se o código numérico ASCII do *caractere_1* for maior que o código numérico ASCII do *caractere_2*.

`cgreaterpignore` (*caractere_1*, *caractere_2*) [Função]
Como `cgreaterp` mas ignora a caixa alta/baixa.

`charp` (*obj*) [Função]
Retorna `true` se *obj* for um caractere do Maxima. Veja na seção "Introdução a manipulação de sequências de caracteres" para ter um exemplo.

`cint` (*caractere*) [Função]
Retorna o código numéico ASCII de *caractere*.

`clessp` (*caractere_1*, *caractere_2*) [Função]
Retorna `true` se o código numérico ASCII de *caractere_1* for menor que o código numérico ASCII de *caractere_2*.

`clesspignore` (*caractere_1*, *caractere_2*) [Função]
Como em `clessp` ignora a caixa alta/baixa.

`constituent` (*caractere*) [Função]
Retorna `true` se *caractere* for caractere gráfico e não o caractere de espaço em branco. Um caractere gráfico é um caractere que se pode ver, adicionado o caractere de espaço em branco. (`constituent` foi definida por Paul Graham, em ANSI Common Lisp, 1996, página 67.)

```
(%i1) load("stringproc")$
(%i2) 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*) [Função]
Converte um caractere do Lisp em um caractere do Maxima. (É possível que não chegue a precisar dessa função.)

`digitcharp` (*caractere*) [Função]
Retorna `true` se *caractere* for um dígito (algarismo de 0 a 9).

`lcharp` (*obj*) [Função]
Retorna `true` se *obj* for um caractere do Lisp. (Pode não precisar dessa função.)

`lowercasep` (*caractere*) [Função]
Retorna `true` se *caractere* for um caractere em caixa baixa.

`newline` [Variable]
O caractere de nova linha.

`space` [Variável]
O caractere de espaço em branco.

`tab` [Variável]
O caractere de tabulação.

`uppercasep` (*caractere*) [Função]
Retorna `true` se *caractere* for um caractere em caixa alta.

69.4 Definições para sequências de caracteres

`sunlisp` (*lisp_string*) [Função]
Converte uma sequência de caracteres do Lisp em uma sequência de caracteres do Maxima. (Em geral, pode não chegar a precisar dessa função.)

`lstringp` (*obj*) [Função]
Retorna `true` se *obj* is uma sequência de caracteres do Lisp. (Em geral, pode não chegar a precisar dessa função.)

`stringp` (*obj*) [Função]
Retorna `true` se *obj* for uma sequência de caracteres do Maxima. Veja a introdução para obter exemplos.

`charat` (*seq_caracte*, *n*) [Função]
Retorna o *n*-ésimo caractere de *seq_caracte*. O primeiro caractere em *seq_caracte* é retornado com $n = 1$.

```
(%i1) load("stringproc")$
(%i2) charat("Lisp",1);
(%o2)                           L
```

`charlist` (*seq_caracte*) [Função]
Retorna a lsita de todos os caracteres em *seq_caracte*.

```
(%i1) load("stringproc")$
(%i2) charlist("Lisp");
(%o2)                     [L, i, s, p]
(%i3) %[1];
(%o3)                           L
```

`parsetoken` (*seq_caracte*) [Função]
`parsetoken` converte a primeira ficha em *seq_caracte* para o correspondente número ou retorna `false` se o número não puder ser determinado. O conjunto de delimitadores para a troca de fichas é `{space, comma, semicolon, tab, newline}`

Nota de tradução: espaço, vírgula, ponto e vírgula, tabulação e nova linha.

```
(%i1) load("stringproc")$
(%i2) 2*parsetoken("1.234 5.678");
(%o2)                          2.468
```

Para analizar, pode também usar a função `parse_string`. Veja a descrição no ficheiro 'share\contrib\eval_string.lisp'.

sconc (*expr_1*, ..., *expr_n*) [Função]

Avalia seus argumentos e concatena-os em uma sequência de caracteres. `sconc` é como `sconcat` mas retorna uma sequência de caracteres do Maxima.

```
(%i1) load("stringproc")$
(%i2) sconc("xx[",3,"]:",expand((x+y)^3));
(%o2)             xx[3]:y^3+3*x*y^2+3*x^2*y+x^3
(%i3) stringp(%);
(%o3)                          true
```

scopy (*seq_caracte*) [Função]

Retorna uma cópia de *seq_caracte* como uma nova sequência de caracteres.

sdowncase (*seq_caracte*) [Função]
sdowncase (*seq_caracte*, *início*) [Função]
sdowncase (*seq_caracte*, *início*, *fim*) [Função]

Como em `supcase`, mas caracteres em caixa alta são convertidos para caracteres em caixa baixa.

sequal (*seq_caracte__1*, *seq_caracte__2*) [Função]

Retorna `true` se *seq_caracte__1* e *seq_caracte__2* tiverem o mesmo comprimento e contiverem os mesmos caracteres.

sequalignore (*seq_caracte__1*, *seq_caracte__2*) [Função]

Como em `sequal` mas igonara a caixa alta/baixa.

sexplode (*seq_caracte*) [Função]

`sexplode` é um apelido para a função `charlist`.

simplode (*lista*) [Função]
simplode (*lista*, *delim*) [Função]

`simplode` takes uma `lista` ou expressões e concatena-as em uma sequência de caracteres. Se nenhum delimitador *delim* for usado, `simplode` funciona como `sconc` e não utiliza delimitador. *delim* pode ser qualquer sequência de caracteres.

```
(%i1) load("stringproc")$
(%i2) simplode(["xx[",3,"]:",expand((x+y)^3)]);
(%o2)             xx[3]:y^3+3*x*y^2+3*x^2*y+x^3
(%i3) simplode( sexplode("stars")," * " );
(%o3)                   s * t * a * r * s
(%i4) simplode( ["One","more","coffee."]," " );
(%o4)                   One more coffee.
```

sinsert (*seq*, *seq_caracte*, *pos*) [Função]

Retorna uma sequência de caracteres que é uma concatenação de `substring (seq_caracte, 1, pos - 1)`, a sequência de caracteres *seq* e `substring (seq_caracte, pos)`. Note que o primeiro caractere está em *seq_caracte* e está na posição 1.

```
(%i1) load("stringproc")$
(%i2) s: "A submarine."$
(%i3) sconc( substring(s,1,3),"yellow ",substring(s,3) );
```

```
(%o3)                  A yellow submarine.
(%i4) sinsert("hollow ",s,3);
(%o4)                  A hollow submarine.
```

sinvertcase (*seq_caracte*) [Função]
sinvertcase (*seq_caracte*, *início*) [Função]
sinvertcase (*seq_caracte*, *início*, *fim*) [Função]

Retorna *seq_caracte* excepto que cada caractere da posição *início* até a posição *fim* está invertido. Se a posição *fim* não for fornecida, todos os caracteres do início ao *fim* de *seq_caracte* são substituídos.

```
(%i1) load("stringproc")$
(%i2) sinvertcase("sInvertCase");
(%o2)                      SiNVERTcASE
```

slength (*seq_caracte*) [Função]

Retorna número de caracteres em *seq_caracte*.

smake (*num*, *caractere*) [Função]

Retorna uma nova sequência de caracteres repetindo *num* vezes *caractere*.

```
(%i1) load("stringproc")$
(%i2) smake(3,"w");
(%o2)                          www
```

smismatch (*seq_caracte__1*, *seq_caracte__2*) [Função]
smismatch (*seq_caracte__1*, *seq_caracte__2*, *test*) [Função]

Retorna a posição do primeiro caractere de *seq_caracte__1* no qual *seq_caracte__1* e *seq_caracte__2* diferem ou `false` em caso contrário. A função padrao de teste para coincidência é `sequal`. Se `smismatch` pode ignorar a caixa alta/baixa, use `sequalignore` como função de teste.

```
(%i1) load("stringproc")$
(%i2) smismatch("seven","seventh");
(%o2)                           6
```

split (*seq_caracte*) [Função]
split (*seq_caracte*, *delim*) [Função]
split (*seq_caracte*, *delim*, *multiple*) [Função]

Retorna a lista de todas as fichas em *seq_caracte*. Cada ficha é uma sequência de caracteres não analisada. `split` usa *delim* como delimitador. Se *delim* não for fornecido, o caractere de espaço é o delimitador padrão. *multiple* é uma variável booleana com `true` como valor padrão. Multiplos delimitadores são lidos como um. Essa função é útil se tabulações são gravadas com caracteres de espaço multiplos. Se *multiple* for escolhido para `false`, cada delimitador é considerado.

```
(%i1) load("stringproc")$
(%i2) split("1.2   2.3   3.4   4.5");
(%o2)                 [1.2, 2.3, 3.4, 4.5]
(%i3) split("first;;third;fourth",";",false);
(%o3)               [first, , third, fourth]
```

sposition (*caractere*, *seq_caracte*) [Função]

Retorna a posição do primeiro caractere em *seq_caracte* que coincide com *caractere*. O primeiro caractere em *seq_caracte* está na posição 1. Para que os caracteres que coincidirem desconsiderem a caixa alta/baixa veja `ssearch`.

sremove (*seq*, *seq_caracte*) [Função]
sremove (*seq*, *seq_caracte*, *test*) [Função]
sremove (*seq*, *seq_caracte*, *test*, *início*) [Função]
sremove (*seq*, *seq_caracte*, *test*, *início*, *fim*) [Função]

Retorna uma sequência de caracteres como *seq_caracte* mas com todas as subsequências de caracteres que coincidirem com *seq*. A função padrão de teste de coincidência é `sequal`. Se `sremove` puder ignorar a caixa alta/baixa enquanto busca por *seq*, use `sequalignore` como teste. Use *início* e *fim* para limitar a busca. Note que o primeiro caractere em *seq_caracte* está na posição 1.

```
(%i1) load("stringproc")$
(%i2) sremove("n't","I don't like coffee.");
(%o2)                    I do like coffee.
(%i3) sremove ("DO ",%,'sequalignore);
(%o3)                     I like coffee.
```

sremovefirst (*seq*, *seq_caracte*) [Função]
sremovefirst (*seq*, *seq_caracte*, *test*) [Função]
sremovefirst (*seq*, *seq_caracte*, *test*, *início*) [Função]
sremovefirst (*seq*, *seq_caracte*, *test*, *início*, *fim*) [Função]

Como em `sremove` excepto que a primeira subsequência de caracteres que coincide com `seq` é removida.

sreverse (*seq_caracte*) [Função]

Retorna uma sequência de caracteres com todos os caracteres de *seq_caracte* em ordem reversa.

ssearch (*seq*, *seq_caracte*) [Função]
ssearch (*seq*, *seq_caracte*, *test*) [Função]
ssearch (*seq*, *seq_caracte*, *test*, *início*) [Função]
ssearch (*seq*, *seq_caracte*, *test*, *início*, *fim*) [Função]

Retorna a posição da primeira subsequência de caracteres de *seq_caracte* que coincide com a sequência de caracteres *seq*. A função padrão de teste de coincidência é `sequal`. Se `ssearch` puder igonorar a caixa alta/baixa, use `sequalignore` como função de teste. Use *início* e *fim* para limitar a busca. Note que o primeiro caracter em *seq_caracte* está na posição 1.

```
(%i1) ssearch("~s","~{~S ~}~%",'sequalignore);
(%o1)                                4
```

ssort (*seq_caracte*) [Função]
ssort (*seq_caracte*, *test*) [Função]

Retorna uma sequência de caracteres que contém todos os caracteres de *seq_caracte* em uma ordem tal que não existam dois caracteres *c* sucessivos e *d* seja tal que `test (c, d)` seja `false` e `test (d, c)` seja `true`. A função padrão de teste para ordenação

é *clessp*. O conjunto de funções de teste é {`clessp`, `clesspignore`, `cgreaterp`, `cgreaterpignore`, `cequal`, `cequalignore`}.

```
(%i1) load("stringproc")$
(%i2) ssort("I don't like Mondays.");
(%o2)                    '.IMaddeiklnnoosty
(%i3) ssort("I don't like Mondays.",'cgreaterpignore);
(%o3)                 ytsoonnMlkIiedda.'
```

`ssubst` (*nova*, *antiga*, *seq_caracte*) [Função]
`ssubst` (*nova*, *antiga*, *seq_caracte*, *test*) [Função]
`ssubst` (*nova*, *antiga*, *seq_caracte*, *test*, *início*) [Função]
`ssubst` (*nova*, *antiga*, *seq_caracte*, *test*, *início*, *fim*) [Função]

Retorna uma sequência de caracteres como *seq_caracte* excepto que todas as subsequências de caracteres que coincidirem com *antiga* são substituídas por *nova*. *antiga* e *nova* não precisam ser de mesmo comprimento. A função padrão de teste para coincidência é para coincidências é `sequal`. Se `ssubst` puder ignorar a cixa alta/baixa enquanto procurando por *antiga*, use `sequalignore` como função de teste. Use *início* e *fim* para limitar a busca. Note que o primeiro caractere em *seq_caracte* está na posição 1.

```
(%i1) load("stringproc")$
(%i2) ssubst("like","hate","I hate Thai food. I hate green tea.");
(%o2)          I like Thai food. I like green tea.
(%i3) ssubst("Indian","thai",%,'sequalignore,8,12);
(%o3)         I like Indian food. I like green tea.
```

`ssubstfirst` (*nova*, *antiga*, *seq_caracte*) [Função]
`ssubstfirst` (*nova*, *antiga*, *seq_caracte*, *test*) [Função]
`ssubstfirst` (*nova*, *antiga*, *seq_caracte*, *test*, *início*) [Função]
`ssubstfirst` (*nova*, *antiga*, *seq_caracte*, *test*, *início*, *fim*) [Função]

Como em `subst` excepto que somente a primeira subsequência de caracteres que coincidir com *antiga* é substituída.

`strim` (*seq*,*seq_caracte*) [Função]

Retorna uma sequência de caracteres como *seq_caracte*, mas com todos os caracteres que aparecerem em *seq* removidos de ambas as extremidades.

```
(%i1) load("stringproc")$
(%i2) "/* comment */"$
(%i3) strim(" /*",%);
(%o3)                          comment
(%i4) slength(%);
(%o4)                             7
```

`striml` (*seq*, *seq_caracte*) [Função]

Como em `strim` excepto que somente a extremidade esquerda de *seq_caracte* é recordada.

`strimr` (*seq*, *seq_caracte*) [Função]

Como em `strim` excepto que somente a extremidade direita de sequência de caracteres é recortada.

substring (*seq_caracte*, *início*) [Função]
substring (*seq_caracte*, *início*, *fim*) [Função]

Retorna a subsequência de caracteres de *seq_caracte* começando na posição *início* e terminando na posição *fim*. O caractere na posição *fim* não é incluído. Se *fim* não for fornecido, a subsequência de caracteres contém o restante da sequência de caracteres. Note que o primeiro caractere em *seq_caracte* está na posição 1.

```
(%i1) load("stringproc")$
(%i2) substring("substring",4);
(%o2)                        string
(%i3) substring(%,4,6);
(%o3)                          in
```

supcase (*seq_caracte*) [Função]
supcase (*seq_caracte*, *início*) [Função]
supcase (*seq_caracte*, *início*, *fim*) [Função]

Retorna *seq_caracte* excepto que caracteres em caixa baixa a partir da posição *início* até a posição *fim* são substituídos pelo correspondente caracteres em cixa alta. Se *fim* não for fornecido, todos os caracteres em caixa baixa de *início* até o fim de *seq_caracte* são substituídos.

```
(%i1) load("stringproc")$
(%i2) supcase("english",1,2);
(%o2)                        English
```

tokens (*seq_caracte*) [Função]
tokens (*seq_caracte*, *test*) [Função]

Retorna uma lista de fichas, que tiverem sido extrídos de *seq_caracte*. As fichas são subsequências de caracteres cujos caracteres satisfazem a uma determinada função de teste. Se o teste não for fornecido, *constituent* é usada como teste padrão. `{constituent, alphacharp, digitcharp, lowercasep, uppercasep, charp, characterp, alphanumericp}` é o conjunto de fnç˜oes de teste. (A versão Lisp de `tokens` é escrita por Paul Graham. ANSI Common Lisp, 1996, page 67.)

```
(%i1) load("stringproc")$
(%i2) tokens("24 October 2005");
(%o2)                   [24, October, 2005]
(%i3) tokens("05-10-24",'digitcharp);
(%o3)                      [05, 10, 24]
(%i4) map(parsetoken,%);
(%o4)                       [5, 10, 24]
```

70 unit

70.1 Introdução a Units

O pacote *unit* torna o utilizador apto a converter entre unidades arbitrárias e trabalhar com dimensões em equações. O funcionamento desse pacote é radicalmente diferente do pacote original units do Maxima - apesar de o original conter uma lista básica de definições, o pacote actual usa um conjunto de regras para permitir ao utilizador escolher, sobre uma base dimensional, qual a resposta fianl de unidade pode ser convertida. Isso irá separar unidades em lugar de misturá-las na tela, permitindo ao utilizador durante a leitura identificar as unidades associadas com uma resposta em particular. Isso permitirá ao utilizador simplificar uma expressão em sua Base fundamental de Unidades, bem como fornecer ajuste fino sobre a simplificação de unidades derivadas. Análise dimensional é possível, e uma variedade de ferramentas está disponível para gerenciar a conversão e também uma variedade de opções de simplificação. Adicionalmente para personalizar conversão automática, *units* também fornede um manual tradicional de opções de conversão.

Nota -quando conversões de unidade forem não exactas Maxima irá fazer aproximações resultando em frações. Esso é uma concequência das técnicas usadas para simplificar unidades. A mensagem de alerta desse tipo de substituição está desabilitada por padrão no caso de inidades (normalmente essas mensagens estão habilitadas) uma vez que essa situação de iemissão de mensagens de alerta ocorre frequêntemente e os alertas confundem a saída. (O estado actual de `ratprint` é restabelecido após uma conversão de unidades, de forma que modificações de utilizador para aquela configuração irão ser preservadas de outra forma.) Se o utilizador precisar dessa informação para `units`, ele pode escolher *unitverbose:on* para reativar a impressão de mensagens de alerta do processo de conversão.

unit está inclído no Maxima no directório share/contrib/unit directory. Isso segue aos pacotes normais do Maxima conforme convenções:

```
(%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.
```

As mensagens WARNING (DE ALERTA) são esperadas n ão uma causa de preocupação - elas indicam que o pacote *unit* está redefinindo funções anteriormente definidas no local

adequado do Maxima. Essa redefinição é necessária com o bojetivo de manusear adequadamente as unidades. O utilizador pode estar consciente que se outras modificações tiverem sido feitas para essas funções por outros pacotes essas novas mudanças irão ser sobrescritas por meio desse processo de disponibilização do pacote `unit`.

O ficheiro *unit.mac* também chama um ficheiro lisp, a saber *unit-functions.lisp*, que contém as funçãoes lisp necessárias ao pacote.

Clifford Yapp é o autor primário. Ele recebeu grande contribuição de Barton Willis da University of Nebraska at Kearney (UNK), Robert Dodier, e da intrépida tribo da lista de mensagens do Maxima.

Existem provavelmente muitos erros. Diga-me quais. `float` e `numer` não fazem o que é esperado.

PORFAZER : funcionalidade de dimensão, manuseio de temperatura, a função `showabbr` e Cia. Ltda. Mostrar exemplos com adição de quantidades contendo unidades.

70.2 Definições para Units

`setunits` (*list*) [Função]

Por padrão, o pacote *unit* não usa qualquer dimensões derivadas, mas irá converter todas as unidades nas sete fundamentais do sistema MKS.

```
(%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
```

Em alguns casos esse é o comportamento desejado. Se o utilizador desejar usar outras unidades, isso é conseguido com o comando `setunits`:

```
(%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
```

A escolha de unidades é completamente flexível. Por exemplo, se quisermos voltar para quiilogramas, metros, e segundos como padrão para essas dimensão nós podemos fazer:

```
(%i11) setunits([kg,m,s]);
(%o11)                                done
(%i12) centigram*inch/minutes^2;
                                    127        kg m
(%o12)                        (-------------) (----)
                               1800000000000     2
                                                 s
```

Unidade derivadas são também manuseáveis por meio desse comando:

```
(%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
```

Note que o pacote *unit* reconhece a combinação não MKS de massa, comprimento, e tempo inverso elevado ao quadrado como uma força, e converte isso para Newtons. É dessa forma que Maxima trabalha geralmente. Se, por exemplo, nós preferirmos dinas em lugar de Newtons, simplesmente fazemos o seguinte:

```
(%i22) setunits(dyn);
(%o22)                                done
```

```
(%i23) kg*m/s^2;
(%o23)                              (100000) (dyn)
(%i24) centigram*inch/minutes^2;
                                      127
(%o24)                              (--------) (dyn)
                                     18000000
```

Para descontinuar simplificando para qualquer unidade de força, usamos o comando `uforget`:

```
(%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
```

Isso pode trabalhar igualmente bem com `uforget(N)` ou `uforget(%force)`.

Veja também `uforget`. Para usar essa função escreva primeiro `load("unit")`.

`uforget` (*list*) [Função]

Por padrão, o pacote *unit* converte todas as unidades para as sete unidaes fundamentais do sitema MKS de unidades. Ess comportamento pode ser mudado com o comando `setunits`. Após o qual, o utilizador pode restabelecer o comportamento padrão para uma dimensão em particular mediante o comando `uforget`:

```
(%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` opera sobre dimensões, não sobre unidades, de forma que qualquer unidade de uma dimensão em particular irá trabalhar. A própia dimensão é também um argumento legal.

Veja também `setunits`. To use this function write first `load("unit")`.

`convert` (*expr*, *list*) [Função]

Quando do restabelecimento dos valores padrão o ambiente global é destruído, existe o comando `convert`, que permite conversões imediatas. `convert` pode aceitar um argumetno simples ou uma lista de unidades a serem usadas na conversão. Quando uma operação de conversão for concluída, o sistema normal de avaliação global é contornado, com o objectivo de evitar que o resultado desejado seja convertido novamente. Como consequência, em cálculos aproximados alertas de "rat" irão ser visíveis se o ambiente global que controla esse comportamento (`ratprint`) for `true`. `convert` também é útil para uma verificação pontual e imediata da precisão de uma conversão global. Outro recurso é que `convert` irá permitir a um utilizador fazer um Base de Conversões Dimensionais mesmo se o ambiente global for escolhido para simplificar par uma Dimensão Derivada.

```
(%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
```

Veja também `setunits` e `uforget`. Para usar essa função primeiramente escreva `load("unit")`.

`usersetunits` [Variável de opção]

Valor por omissão: none

Se um utilizador desejar ter um comportamento padrão de unidade diferente daquele descrito, ele pode fazer uso de *maxima-init.mac* e da variável *usersetunits*. O pacote *unit* irá verificar o ficheiro *maxima-init.mac* na inicialização para ver se a essa variável foi atribuído uma lista. Se isso aconteceu, o pacote *unit* irá usar `setunits` sobre aquela lista e pegar as unidades lá colocadas para serem as padrões. `uforget` irá reverter para o comportamento definido por `usersetunits` sobrescrevendo seus próprios padrões. Por exemplo, Se tivermos um ficheiro *maxima-init.mac* contendo:

```
usersetunits : [N,J];
```

nós poderemos ver o seguinte comportamento:

```
(%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)                    (624150959647704268800) (eV)
(%i11) uforget([dyn,eV]);
(%o11)                             [false, false]
(%i12) kg*m/s^2;
(%o12)                                   N
(%i13) kg*m^2/s^2;
(%o13)                                   J
(%i14) kg*m^3/s^2;
(%o14)                                  J m
(%i15) kg*m*km/s^2;
(%o15)                               (1000) (J)
```

Sem `usersetunits`, as entradas iniciais poderiam ter sido convertidas para o sistema de unidades MKS, e `uforget` poderia ter resultado em um retorno para as regras do MKS. Em vez disso, as preferências do utilizador foram respeitadas em ambos os casos. Note que esse podem ainda serem sobrescritos se for desejado. Para eliminar completamente essa simplificação - i.e. ter as preferências de utilizador escolhidas para os padrões de unidade do Maxima - o comando `dontusedimension` pode ser usado. `uforget` pode restabelecer as preferências de utilizador novamente, mas somente se `usedimension` liberar isso para uso. Alternativamente, `kill(usersetunits)` irá remover completametne todo o conhecimento dessas escolhas de utilizador da sessão actual. Aqui está alguns exemplos de como esssas várias opções trabalham.

```
(%i2) kg*m/s^2;
(%o2)                                    N
(%i3) kg*m^2/s^2;
```

```
(%o3)                                  J
(%i4) setunits([dyn,eV]);
(%o4)                                done
(%i5) kg*m/s^2;
(%o5)                           (100000) (dyn)
(%i6) kg*m^2/s^2;
(%o6)                  (6241509596477042688) (eV)
(%i7) uforget([dyn,eV]);
(%o7)                          [false, false]
(%i8) kg*m/s^2;
(%o8)                                  N
(%i9) kg*m^2/s^2;
(%o9)                                  J
(%i10) dontusedimension(N);
(%o10)                             [%force]
(%i11) dontusedimension(J);
(%o11)                         [%energy, %force]
(%i12) kg*m/s^2;
                                       kg m
(%o12)                                 ----
                                         2
                                        s
(%i13) kg*m^2/s^2;
                                          2
                                       kg m
(%o13)                                 -----
                                         2
                                        s
(%i14) setunits([dyn,eV]);
(%o14)                                 done
(%i15) kg*m/s^2;
                                       kg m
(%o15)                                 ----
                                         2
                                        s
(%i16) kg*m^2/s^2;
                                          2
                                       kg m
(%o16)                                 -----
                                         2
                                        s
(%i17) uforget([dyn,eV]);
(%o17)                         [false, false]
(%i18) kg*m/s^2;
                                       kg m
(%o18)                                 ----
                                         2
```

```
                                      s
(%i19) kg*m^2/s^2;
                                         2
                                      kg m
(%o19)                                -----
                                        2
                                       s
(%i20) usedimension(N);
Done.  To have Maxima simplify to this dimension, use setunits([unit])
to select a unit.
(%o20)                                true
(%i21) usedimension(J);
Done.  To have Maxima simplify to this dimension, use setunits([unit])
to select a unit.
(%o21)                                true
(%i22) kg*m/s^2;
                                      kg m
(%o22)                                ----
                                        2
                                       s
(%i23) kg*m^2/s^2;
                                         2
                                      kg m
(%o23)                                -----
                                        2
                                       s
(%i24) setunits([dyn,eV]);
(%o24)                                done
(%i25) kg*m/s^2;
(%o25)                          (100000) (dyn)
(%i26) kg*m^2/s^2;
(%o26)                 (6241509596477042688) (eV)
(%i27) uforget([dyn,eV]);
(%o27)                           [false, false]
(%i28) kg*m/s^2;
(%o28)                                 N
(%i29) kg*m^2/s^2;
(%o29)                                 J
(%i30) kill(usersetunits);
(%o30)                                done
(%i31) uforget([dyn,eV]);
(%o31)                          [false, false]
(%i32) kg*m/s^2;
                                      kg m
(%o32)                                ----
                                        2
                                       s
```

```
(%i33) kg*m^2/s^2;
                                          2
                                       kg m
(%o33)                                 -----
                                         2
                                        s
```

Desafortunadamente essa ampla variedade de opções é um pouco confus no início, mas uma vez que o utilizador cultiva o uso delas o utilizador perceberá que elas permitem completo controle sobre seu ambiente de trabalho.

`metricexpandall` (*x*) [Função]

Reconstrói listas de unidades globais automaticamente criando todas as unidades métricas desejadas. *x* é um argumento numérico que é usado para especificar quantos prefixos métricos o utilizador deseja que seja definido. Os argumentos são os seguintes, com cada maior número definindo todos os menores números de unidade:

```
            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
```

Normalmente, Maxima não irá definir a expansão completa desses resultados em uma grande número de unidades, mas `metricexpandall` pode ser usada para reconstruir a lista em um estilo mais ou menos completo. A variável relevante no ficheiro *unit.mac* é *%unitexpand*.

`%unitexpand` [Variável]

Valor por omissão: `2`

Ess é o valor fornecido a `metricexpandall` durante a inicialização de *unit*.

71 zeilberger

71.1 Introdução a zeilberger

`zeilberger` é uma implementação do algoritmo de Zeilberger para somatório hipergeométricos definidos, e também para o algoritmo de Gosper para somatórios hipergeométricos indefinidos.

`zeilberger` faz uso do método de optimização "filtering" desenvolvido por Axel Riese.

`zeilberger` foi desenvolvido por Fabrizio Caruso.

`load ("zeilberger")` torna esse pacote disponível para uso.

71.1.1 O problema dos somatórios hipergeométricos indefinidos

`zeilberger` implementa o algoritmo de Gosper para somatório hipergeométrico indefinido. Dado um termo hipergeométrico F_k em k queremos encontrar sua anti-diferença hipergeométrica, isto é, um termo hipergeométrico f_k tal que $F_k = f_(k+1) - f_k$.

71.1.2 O problema dos somatórios hipergeométricos definidos

`zeilberger` implementa o algoritmo de Zeilberger para somatório hipergeométrico definido. Dado um termo hipergeométrico apropriado (em n e k) $F_(n,k)$ e um inteiro positivo d queremos encontrar um d-ésima ordem de recorrência linear com coeficientes polinomiais (em n) para $F_(n,k)$ e uma função racional R em n e k tal que

$a_0 F_(n,k) + ... + a_d F_(n+d), k = Delta_K(R(n,k) F_(n,k))$

onde $Delta_k$ é o k-seguinte operador de diferença, i.e., $Delta_k(t_k) := t_(k+1) - t_k$.

71.1.3 Níveis de detalhe nas informações

Existe também versões de níveis de detalhe fornecidos pelos comandos que são chamados (os níveis) através da adição de um dos seguintes prefixos:

`Summary` Apenas um sumário é mostrado no final

`Verbose` Algumas informações nos passos intermédios

`VeryVerbose`
Muita informação

`Extra` Muito mais informação incluindo informação sobre o sistema linear no algoritmo de Zeilberger

Por exemplo: `GosperVerbose`, `parGosperVeryVerbose`, `ZeilbergerExtra`, `AntiDifferenceSummary`.

71.2 Definições para zeilberger

`AntiDifference (F_k, k)` [Função]
Retorna a anti-diferença hipergeométrica de `F_k`, se essa anti-diferença. De outra forma `AntiDifference` retorna `no_hyp_antidifference`.

`Gosper` (*F_k*, *k*) [Função]

Retorna o certificado racional *R(k)* para *F_k*, isto é, uma função racional tal que

$F_k = R(k+1)F_(k+1) - R(k)F_k$

se essa função racional exitir. De outra forma, `Gosper` retorna `no_hyp_sol`.

`GosperSum` (*F_k*, *k*, *a*, *b*) [Função]

Retorna o somatório de *F_k* de $k = a$ a $k = b$ se *F_k* tiver ma diferença hipergeométrica. De outra forma, `GosperSum` retorna `nongosper_summable`.

Exemplos:

```
(%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)` [Função]

Tenta encontrar uma recorrência de d-ésima ordem para $F\_\{n,k\}$.

O algoritmo retorna uma sequência $[s_1, s_2, ..., s_m]$ de soluções. Cada solução tem a forma

$[R(n,k), [a_0, a_1, ..., a_d]]$

`parGosper` retorna `[]` caso não consiga encontrar uma recorrência.

`Zeilberger (F_{n,k}, k, n)` [Função]

Tenta calcular o somatório hipergeométrico indefinido de $F\_\{n,k\}$.

`Zeilberger` primeiro invoca `Gosper`, e se `Gosper` não conseguir encontrar uma solução, então `Zeilberger` invoca `parGosper`com ordem 1, 2, 3, ..., acima de `MAX_ORD`. Se Zeilberger encontrar uma solução antes de esticar `MAX_ORD`, Zeilberger para e retorna a solução.

O algoritmo retorna uma sequência $[s_1, s_2, ..., s_m]$ de soluções. Cada solução tem a forma

$[R(n,k), [a_0, a_1, ..., a_d]]$

`Zeilberger` retorna `[]` se não conseguir encontrar uma solução.

`Zeilberger` invoca `Gosper` somente se `gosper_in_zeilberger` for `true`.

71.3 Variáveis globais gerais

`MAX_ORD` [Variável global]

Valor por omissão: 5

`MAX_ORD` é a ordem máxima de recorrência tentada por `Zeilberger`.

`simplified_output` [Variável global]

Valor por omissão: `false`

Quando `simplified_output` for `true`, funções no pacote `zeilberger` tentam simplificação adicional da solução.

`linear_solver` [Variável global]

Valor por omissão: `linsolve`

`linear_solver` nomeia o resolvedor que é usado para resolver o sistema de equações no algoritmo de Zeilberger.

`warnings` [Variável global]

Valor por omissão: `true`

Quando `warnings` for `true`, funções no pacote `zeilberger` imprimem mensagens de alerta durante a execução.

`gosper_in_zeilberger` [Variável global]

Valor por omissão: `true`

Quando `gosper_in_zeilberger` for `true`, a função `Zeilberger` chama `Gosper` antes de chamar `parGosper`. De outra forma, `Zeilberger` vai imediatamente para `parGosper`.

`trivial_solutions` [Variável global]

Valor por omissão: `true`

Quando `trivial_solutions` for `true`, `Zeilberger` retorna soluções que possuem certificado igual a zero, ou todos os coeficientes iguais a zero.

71.4 Variáveis relacionadas ao teste modular

`mod_test` [Variável global]

Valor por omissão: `false`

Quando `mod_test` for `true`, `parGosper` executa um teste modular discartando sistemas sem solução.

`modular_linear_solver` [Variável global]

Valor por omissão: `linsolve`

`modular_linear_solver` nomeia o resolvedor linear usado pelo teste modular em `parGosper`.

`ev_point` [Variável global]

Valor por omissão: `big_primes[10]`

`ev_point` é o valor no qual a variável n é avaliada no momento da execução do teste modular em `parGosper`.

`mod_big_prime` [Variável global]

Valor por omissão: `big_primes[1]`

`mod_big_prime` é o módulo usado pelo teste modular em `parGosper`.

`mod_threshold` [Variável global]

Valor por omissão: 4

`mod_threshold` is the maior ordem para a qual o teste modular em `parGosper` é tentado.

Apêndice A Índice de Funções e Variáveis

!

#

%

’

*

+

—

.

/

:

<

=

>

?

[

]

^

_

|

~

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z