Manual do Maxima

Maxima é um sistema de álgebra computacional, implementado em Lisp.

Maxima é derivado do sistema Macsyma, desenvolvido no MIT nos anos de 1968 a 1982 como parte do Projeto MAC. MIT remanejou uma cópia do código fonte do Macsyma para o Departamento de Energia em 1982; aquela versão é agora conhecida como Macsyma DOE. Uma cópia do Macsyma DOE foi mantida pelo Professor William F. Schelter da Universidade do Texas de 1982 até sua morte em 2001. Em 1998, Schelter obteve permissão do Departamento de Energia para liberar o código fonte do Macsyma DOE sob a Licença Pública GNU, e em 2000 ele iniciou o projeto Maxima no SourceForge para manter e desenvolver o Macsyma DOE, agora chamado Maxima.

Notas de tradução:

O código fonte deste documento encontra-se no formato texinfo. Para contribuir com a equipe do Maxima na tarefa de manter a tradução para o português sempre atualizada envie um e-mail para <maxima at math dot utexas dot edu>.

Em caso de dúvida sobre algum trecho deste manual consulte o original inglês.

A seção "Pacotes adicionais" foi anexada ao manual recentemente (fevereiro/2006) e possui partes a serem traduzidas.

Caso sua dúvida persista ou tenha alguma sugestão/aperfeiçoamento/ crítica mande por e-mail para Jorge Barros de Abreu <ficmatin01 at solar dot com dot br>

Mantenedores:

- Vadim V. Zhytnikov (UTF-8)
- Jaime E. Villate (Gráficos)
- Mario Rodriguez
- Jorge Barros de Abreu

Índice Resumido

Sumário

1 Introdução ao Maxima

Inicie o Maxima com o comando "maxima". Maxima mostrará a informação de versão e uma linha de comando. Termine cada comando Maxima com um ponto e vírgula. Termine uma sessão com o comando "quit();". Aqui está um exemplo de sessão:

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

Maxima pode procurar as páginas info. Use o comando *describe* para mostrar todos os comandos e variáveis contendo uma dada seqüência de caracteres, e opcionalmente sua documentação. O ponto de interrogação ? é uma abreviatura para describe:

```
(%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’: 6 5

Info from file /usr/local/info/maxima.info:
 - Function: integerp (<expr>)
     Returns ‘true’ if <expr> is an integer, otherwise ‘false’.


 - Function: askinteger (expr, integer)
 - Function: askinteger (expr)
 - Function: askinteger (expr, even)
```

```
 - Function: askinteger (expr, odd)
     'askinteger (expr, integer)' attempts to determine from the
     'assume' database whether 'expr' is an integer.  'askinteger' will
     ask 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)                          false
```

Para usar um resultado em cálculos posteriores, você pode atribuir esse valor a uma variável ou referir-se a esse mesmo valor através de seu rótulo gerado automaticamente. Adicionalmente, `%` refere-se ao mais recente resultado calculado:

```
(%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 cálculos diferenciais e integrais:

```
(%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 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 resolver sistemas de equações não lineares. Note que se você não quer um resultado impresso, você pode encerrar seu comando com *$* em lugar de encerrar com *;*.

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

                                 3 sqrt(5) - 7        sqrt(5) - 3
                           [y = -------------, x = - -----------]]
                                      2                    2
(%i4) kill(labels);
(%o0)                          done
(%i1) plot2d (sin(x)/x, [x, -20, 20]);
(%o1)
(%i2) plot2d ([atan(x), erf(x), tanh(x)], [x, -5, 5]);
(%o2)
(%i3) plot3d (sin(sqrt(x^2 + y^2))/sqrt(x^2 + y^2), [x, -12, 12], [y, -12, 12]);
(%o3)
```

2 Detecção e Relato de Erros

2.1 Introdução à Detecção e Relato de Erros

Como todos os grandes programas, Maxima contém erros conhecidos e erros desconhecidos. Esse capítulo descreve as facilidades internas para executar o conjunto de testes do Maxima bem como informar novos erros.

2.2 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 "passes," e testes que não produzem a resposta desejada, são marcados como erros conhecidos.

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

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

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

Se o terceiro argumento opcional for dado, um subconjunto de testes é executado. O subconjunto de testes para executar é dado como uma lista de nomes dos testes. O conjunto completo de 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 usuário e variáveis) e então define funções e variáveis apropriadamente para o teste.

`run_testsuite` retorna `done`.

testsuite_files Variável de opção

`testsuite_files` é o conjunto de testes a ser executado por `run_testsuite`. Isso é uma lista de nomes de arquivos contendo os testes a executar. Se alguns dos testes em um arquivo falha de forma conhecida, então em lugar de listar o nome do arquivo, uma lista contendo o nome do arquivo e o número dos testes que falharam é usada.

por exemplo, a linha adinate é uma parte do conjunto de testes padrão:

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

Essa linha especifica a suite de testes que consiste dos arquivos "rtest13s" e "rtest14", mas "rtest14" contém dois testes que falham de forma conhecida: 57 e 63.

bug_report () Função

Imprime os números de versão do Maxima e do Lisp, e chama o link para a página web de informação de erros do projeto Maxima. A informação da versão é a mesma reportada por `build_info`.

Quando um erro é informado, é muito útil copiar a versão do Maxima e do Lisp dentro da informação do erro.

`bug_report` retorna uma seqüência de caracteres vazia `""`.

build_info () Função

Imprime um sumário de parâmetros da compilação do Maxima.

`build_info` retorna uma seqüência de caracteres vazia `""`.

3 Ajuda

3.1 Introdução a Ajuda

A função primária de ajuda on-line é `describe`, que é tipicamente invocada através do ponto de interrogação `?` na linha de comando interativa. `? foo` (com um espaço entre `?` e `foo`) é equivalente a `describe ("foo")`, onde `foo` é o nome ou parte do nome de uma função ou tópico; `describe` então acha todos os ítens documentados que possuem a seqüência de caracteres `foo` em seus títulos. Se existe mais que um tal item, Maxima solicita ao usuário selecionar um item ou ítens para mostrar.

```
(%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 exemplo, ítens 7 e 8 foram selecionados. Todos ou nenhum dos ítens poderia ter sido selecionados através da inserção de `all` ou `none`, que podem ser abreviados para `a` ou `n`, respectivamente.

3.2 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 distingüidos 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 arquivo processado por `batch` ou `demo`, mas não em um arquivo 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`.

3.3 Descartando

Computação simbólica tende a criar um bom volume de arquivos temporários, e o efetivo 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 arquivos 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 ajudar você a determinar quais áreas podem precisar de mais espaço.

3.4 Documentação

O manual on-line de usuário do Maxima pode ser visto em diferentes formas. A partir da linha de comando interativa do Maxima, o manual de usuário é visto em texto plano através do comando `?` (i.e., a função `describe`). O manual de usuário 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.

3.5 Definições para Ajuda

demo (*nomedearquivo*) Função

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

`demo` procura na lista de diretórios `file_search_demo` para achar `nomedearquivo`. Se o arquivo tiver o sufixo `dem`, o sufixo pode ser omitido. Veja também `file_search`.

`demo` avalia seus argumento. `demo` retorna o nome do arquivo 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

Encontra todos os ítens documentados que possuem *string* em seus títulos. Se existe mais de um de tal item, Maxima solicita ao usuário selecionar um item ou ítens para mostrar. Na linha de comando interativa, `? foo` (com um espaço entre `?` e `foo`) é equivalente a `describe ("foo")`.

`describe ("")` retorna uma lista de todos os tópicos documentados no manual online.

`describe` não avalia seu argumento. `describe` sempre retorna `false`.

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 selecionados. Todos ou nenhum dos ítens poderia ter sido selecionado através da inserção de `all` ou `none`, que podem ser abreviado para `a` ou para `n`, respectivamente.

veja ção a Ajuda-snt [Introdução a Ajuda], página ção a Ajuda-pg

example (*tópico*) Função
example () Função

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

O nome do arquivo 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 usuário, 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 Comando

4.1 Introdução a Linha de Comando

’ 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
```

”

Operator

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 Comando

alias (*new_name_1, old_name_1, ..., new_name_n, old_name_n*) Função

provê um nome alternativo para uma função (de usuário ou de sistema), variável, array, etc. Qualquer número de argumentos pode ser usado.

debugmode Variável de opção

Valor padrão: `false`

Quando um erro do Maxima ocorre, Maxima iniciará o depurador se `debugmode` for `true`. O usuário 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 ativado.
- `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 multiplas vezes. Para cada instncia 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 seqüência, 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 canônica (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, exceto 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 instncia 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 afetar 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 intermediárias 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 intermediárias 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 exceto para *a_1*, ..., *a_n*. `kill (allbut (`*infolist*`))` libera todos os ítens exceto 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 intermediárias 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 intermediárias, 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 usuário, 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 intermediárias podem ser gerados por algumas funções. O sinalizador `programmode` controla se `solve` e algumas outras funções geram rótulos de expressões intermediárias em lugar de retornar uma lista de expressões. Algumas outras funções, tais como `ldisplay`, sempre geram rótulos de expressões intermediárias.

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 padrão: `[]`

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

nolabels Variável de opção

Valor padrão: `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 padrão: `false`

Quando `optionset` for `true`, Maxima mostrará uma mensagem sempre que uma opção do Maxima for alterada. Isso é útil se o usuário 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 intermediárias, 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 intermediárias 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 intermediária conta como um.

`playback ([`*m*`, `*n*`])` mostra entradas, saídas e expressões intermediárias 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 usuário pressione `enter`. Esse comportamento é similar a `demo`. `playback (slow)` é útil juntamente com `save` ou `stringout` quando criamos um arquivo 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 afetadas 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 padrão: `_`

`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 si 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 símbolos *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 padrão: `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 usuário 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 Sem Argumentos

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

Veja `operators`.

5.4 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 seqüência 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`.

5.5 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`.

5.6 Definições para Operadores

! Operador

O operador fatorial. Para qualquer número complexo `x` (incluíndo números inteiros, racionais, e reais) exceto 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 fator 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 fatoriais.

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

Veja também `binomial`.

O fatorial 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 fatorial de um número complexo, constante conhecida, ou expressão geral não é simplificado. Ainda assim pode ser possível simplificar o fatorial 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 fatorial de um símbolo não associado não é simplificado.

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

Fatoriais 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 fatorial.

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 fato de as regras de avaliação de expressões predicadas (em particular pelo fato 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 distingüe 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 i
(%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 ig
(%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 fato de as regras de avaliação de expressões predicadas (em particular pelo fato 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)` exceto 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 fato 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 fato de as regras de avaliação de expressões predicadas (em particular pelo fato 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 usuário.

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 fatos 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 fatos 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 atualmente 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 polinômio *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 objeto de estado randômico representa o estado do gerador de números randômicos (aleatórios). O estado compreende 627 palavras de 32 bits.

`make_random_state (`*n*`)` retorna um novo objeto de estado randômico 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 randômico *s*.

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

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

set_random_state (*s*) Função

Copia *s* para o estado do gerador de números randômicos.

`set_random_state` sempre retorna `done`.

random (*x*) Função

Retorna um número pseudorandômico. 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 randômicos.

O gerador de números randômicos do Maxima é uma implementação do algorítmo 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)                 1180449161503683163639O
(%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 você não estiver faminilarizado com a representação binária de números em ponto flutuante, você pode se surpreender que `rationalize (0.1)` não seja igual a 1/10. Esse comportamento não é especial para o 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)                ----------------------
                     2361183241434822606848
(%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
    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 numbe
                                        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 fatos 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 objeto; 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], ordergre
                                                                    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 fatores 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 padrão: `true`

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

sublis (*lista, expr*) Função

Faz multiplas 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 padrão: `true` - controla se os substitutos de `lambda` são aplicados na simplificação após as `sublis` serem usadas ou se você tem que fazer um `ev` para pegar 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 selecionada 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 selecionada 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 selecionada 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 Seção 6.5 [Identificadores], página 55

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 vetor de um dado vetor 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 fatores comuns no numerador e denominador de expressões racionais mas somente se os fatores são explícitos.

Algumas vezes é melhor usar `xthru` antes de `ratsimp` em uma expressão com o objetivo de fazer com que fatores 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 incorretos.
4. O algorítmo usa avaliação em pontos aleatóriamente escolhidos para subexpressões selecionadas cuidadosamente. Isso é sempre negócio um tanto quanto perigoso, embora o algorítmo 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 seqüência 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
```

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

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

6.2 Atribuição

Existem dois operadores de atribuição no Maxima, `":"` e `"::"`. E.g., `a: 3` escolhe a variável `a` para 3. `::` atribui o valor da expressão sobre seu lado direito para o valor da quantidade sobre seu lado esquerdo, que deve avaliar para uma variável atômica ou para uma variável com subscrito.

6.3 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`.

6.4 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)
```

6.5 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.

Um caractere especial pode ser declarado alfabético através da função `declare`. Se isso ocorrer, esse caractere não precisa ser precedido por uma contra barra 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 Seção 3.2 [Lisp e Maxima], página 7 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 Seção 3.2 [Lisp e Maxima], página 7 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
```

6.6 Seqüências de caracteres

Strings (seqüências 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`.

Seqüências 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 seqüência \" 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 seqüência 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) exceto através de encaixar o caractere literal na seqüência de caracteres.

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

Seqüências de caracteres no Maxima são implementadas como símbolos do Lisp, não como seqüencias de caracteres do not Lisp; o que pode mudar em futuras versões do Maxima. Maxima pode mostrar seqüências 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 seqüências de caracteres.

Exemplos:

```
(%i1) s_1 : "Isso é uma seqüência de caracteres  do Maxima.";
(%o1)               Isso é uma seqüência de caracteres  do Maxima.
(%i2) s_2 : "Caracteres \"aspas duplas\" e contrabarras \\ encaixados em uma seqüên
(%o2) Caracteres "aspas duplas" e contrabarra \ encaixados em uma seqüência de cara
(%i3) s_3 : "Caractere de fim de linha encaixado
nessa seqüência de caracteres.";
(%o3) Caractere de fim de linha encaixado
nessa seqüência de caracteres.
(%i4) s_4 : "Ignore o \
caractere de \
fim de linha nessa \
seqüência de caracteres.";
(%o4) Ignore o caractere de fim de linha nessa seqüência de caracteres.
(%i5) ?stringdisp : false;
```

```
(%o5)                          false
(%i6) s_1;
(%o6)                Isso é uma seqüência de caracteres  do Maxima.
(%i7) ?stringdisp : true;
(%o7)                          true
(%i8) s_1;
(%o8)              "Isso é uma seqüência de caracteres  do Maxima."
```

6.7 Desigualdade

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

6.8 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
: fatorial `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 direto. Somente é necessário declarar uma função como um operador; a função operador pode ou não estar definida previamente.

Um exemplo de operadores definidos pelo usuário 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 an infix 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 an infix operator
5 @
 ^
(%i9) "@" (5, 3);
(%o9)                       @(5, 3)
```

6.9 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 efetivamente 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 padrão: "

`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 atual de `boxchar`; o caractere de desenho não é armazenado com a expressão de caixa. Isso quer dizer que se você desenhar uma caixa e em seguida mudar o caractere de desenho a caixa anteriormente desenhada será redesenhada com o caractere 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));
                                 %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(`*objeto*`, `*recurso*`)` retorna `true` se *objeto* 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 intncias 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 *a_i* como um 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 objetivo 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 fatores constantes em evidência no primeiro argumento.

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

Quando *a_i* tiver dois ou mais argumentos, um fator é considerado constante se o segundo argumento for um símbolo e o fator 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 objetos:

`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
```

Declaração `feature`.

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

```
(%o3)                          true
```

Declarações `rassociative` and `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*`)` exceto que essa função habilita ao usuário 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 ex-

terna. 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 fato 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

Seleciona a mesma subexpressão que `part`, mas em lugar de apenas retornar aquela subexpressão como seu valor, isso retorna a expressão completa com a subexpressão selecionada mostrada dentro de uma caixa. A caixa é atualmente 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 padrão: `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 multiplo 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 padrão: `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 padrão: `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 padrão: `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 fatorial 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 diretio 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 diretiro 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) "@"(a, b) := sconcat("(", a, ",", b, ")")$
(%i2) :lisp (get '$+ 'lbp)
100
(%i2) :lisp (get '$+ 'rbp)
100
(%i2) infix ("@", 101, 101)$
(%i3) 1 + a@b + 2;
(%o3)                          (a,b) + 3
(%i4) infix ("@", 99, 99)$
(%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) "@"(a, b) := sconcat("(", a, ",", b, ")")$
(%i2) infix ("@", 100, 99)$
(%i3) foo @ bar @ baz;
(%o3)                       (foo,(bar,baz))
(%i4) infix ("@", 100, 101)$
(%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 afetadas 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 objetivo 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 intermediária (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 intermediários 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 padrão: `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 padrão: `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 padrão: `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 moldura 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 fator (que pode ser uma adição) de *expr* pelos outros fatores de *expr*. Isto é, *expr* é *f_1 f_2 ... f_n* onde ao menos um fator, digamos *f_i*, é uma soma de termos. Cada termo naquela soma é multiplicado por outros fatores no produto. (A saber todos os fatores exceto *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 seqüência de caracteres se o operador principal for uma operador interno ou definido pelo usuário como prefixado, binário ou n-ário infixo, posfixado, matchfix ou nofix. De outra forma `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"
```

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 padrão: `%`

`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 selecionada 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 selecionada 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 fatores 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 fatores, 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 padrão: `false`

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

pickapart (*expr, n*) Função

Atribui rótulos de expressão intermediária 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 intermediárias 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 selecionada 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*.

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

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

See also `nouns` and `evflag`.

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

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 testando.

Antigamente, era possível para um usuário inspecionar `askexp` entrando em uma parada do 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. `askinteger` pergunta ao usuário pela linha de comando se isso não puder ser feito de outra forma, e tenta instalar a informação na base de dados do `assume` se for possível. `askinteger (expr)` é equivalente a `askinteger (expr, integer)`.

`askinteger (expr, even)` e `askinteger (expr, odd)` da mesma forma 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` pergunta ao usuário pelas questões necessárias para completar a sua dedução. As respostas do usuário são guardadas na base de dados pelo tempo que durar a computação corrente. O valor de retorno de `asksign` é um entre `pos`, `neg`, ou `zero`.

demoivre (*expr*) Função
demoivre Variável de opção

A função `demoivre (expr)` converte uma expressão sem escolher a variável global `demoivre`.

Quando a variável `demoivre` for `true`, exponenciais complexas são convertidas em expressões equivalentes em termos de funções circulares: `exp (a + b*%i)` simplifica para `%e^a * (cos(b) + %i*sin(b))` se `b` for livre de `%i`. `a` e `b` não são 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 padrão: `real`

Quando `domain` for escolhida para `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*. Produtos de somas e somas exponenciadas são multiplicadas para fora, numeradores de expressões racionais que são adições são quebradas em suas respectivas parcelas, e multiplicação (comutativa e não comutativa) é distribuída sobre a adição em todos os níveis de *expr*.

Para polinômios se pode usar freqüêntemente `ratexpand` que possui um algorítmo 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 com o objetivo de expandir partes mas não tudo em uma expressão.

`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 for desejado ter `(x+1)^n` expandido onde `n` é maior que `expop` então executando `expand ((x+1)^n)` trabalhará somente se `maxposex` não for menor que `n`.

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

O arquivo '`simplification/facexp.mac`' contém muitas funções relacionadas (em particular `facsum`, `factorfacsum` e `collectterms`, que são chamadas automaticamente) e variáveis (`nextlayerfactor` e `facsum_combine`) que fornecem ao usuário com a habilidade para estruturar expressões por expansão controlada. Descrições breves de função estão disponível em '`simplification/facexp.usg`'. Um arquivo demonstrativo está disponível fazendo `demo("facexp")`.

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

Expande a expressão `expr` com relação às variáveis *x_1*, ..., *x_n*. Todos os produtos envolvendo as variáveis aparecem explicitamente. A forma retornada será livre de produtos de somas de expressões que não estão livres das variáveis. *x_1*, ..., *x_n* podem ser variáveis, operadores, ou expressões.

Por padrão, denominadores não são expandidos, mas isso pode ser controlado através do comutador `expandwrt_denom`.

Essa função, `expandwrt`, não é automaticamente chamada a partir de '`simplification/stopex.mac`'.

expandwrt_denom Variável de opção

Valor padrão: `false`

`expandwrt_denom` controla o tratamento de expressões racionais por `expandwrt`. Se `true`, então ambos o numerador e o denominador da expressão serão expandidos conforme os argumentos de `expandwrt`, mas se `expandwrt_denom` for `false`, então somente o numerador será expandido por aquele caminho.

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

é similar a `expandwrt`, mas trata expressões que são produtos um tanto quanto diferentemente. `expandwrt_factored` expande somente sobre esses fatores de `expr` que contiverem as variáveis *x_1*, ..., *x_n*.

Essa função é automaticamente chamada a aprtir de '`simplification/stopex.mac`'.

expon Variável de opção

Valor padrão: 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 funções circulares e hiperbólicas em *expr* para exponenciais, sem escolher a variável global `exponentialize`.

Quando a variável `exponentialize` for `true`, todas as funções circulares e hiperbólicas são convertidas para a forma exponencial. O valor padrão 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 padrão: 0

`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 for desejado ter `(x+1)^n` expandido onde `n` é maior que `expop` então executando `expand ((x+1)^n)` trabalhará somente se `maxposex` não for menor que `n`.

factlim Variável de opção

Valor padrão: -1

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

intosum (*expr*) Função

Move fatores multiplicativos fora de um somatório para dentro. Se o índice for usado na expressão de fora, então a função tentará achar um índice razoável, o mesmo que é feito para `sumcontract`. Isso é essencialmente a idéia inversa da propriedade `outative` de somatórios, mas note que isso não remove essa propriedade, somente pula sua verificação.

Em alguns casos, um `scanmap (multthru,` *expr*`)` pode ser necessário 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. Para funções de uma única variável `f` então declarada, a "expansão" `f(x + y)` retorna `f(x) + f(y)`, a "expansão" `f(a*x)` retorna `a*f(x)` e ocorre onde `a` é uma "constante". Para funções de dois ou mais argumentos, "linearidade" é definida para ser como no caso de `sum` ou `integrate`, i.e., `f (a*x + b, x)` retorna `a*f(x,x) + b*f(1,x)` para `a` e `b` livres de `x`.

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

mainvar Declaração

Você pode declarar variáveis para serem `mainvar` (variável principal). A escala de ordenação para átomos é essencialmente: números `<` constantes (e.g., `%e`, `%pi`) `<` escalares `<` outras variáveis `<` mainvars. E.g., compare `expand ((X+Y)^4)` com `(declare (x, mainvar), expand ((x+y)^4))`. (Nota: Cuidado deve ser tomado se você eleger o uso desse recurso acima. E.g., se você subtrair uma expressão na qual `x` for uma `mainvar` de uma na qual `x` não seja uma `mainvar`, resimplificação e.g. com `ev (expr, simp)` pode ser necessária se for para ocorrer um cancelamento. Também, se você grava uma expressão na qual `x` é uma `mainvar`, você provavelmente pode também gravar `x`.)

maxapplydepth Variável de opção

Valor padrão: 10000

`maxapplydepth` é a máxima definição para a qual `apply1` e `apply2` irão pesquisar.

maxapplyheight Variável de opção

Valor padrão: 10000

`maxapplyheight` é a elevação máxima a qual `applyb1` irá alcançar antes de abandonar.

maxnegex Variável de opção

Valor padrão: 1000

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

maxposex Variável de opção

Valor padrão: 1000

`maxposex` é o maior expoente que irá ser 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` distribue sobre aquele produto. E.g., `f(x*y)` simplifica para `f(x)*f(y)`.

2. Se `f` é uma função de 2 ou mais argumentos, multiplicatividade é definida como multiplicatividade no primeiro argumento para `f`, e.g., `f (g(x) * h(x), x)` simplifica para `f (g(x) ,x) * f (h(x), x)`.

Essa simplificação não ocorre quando `f` é aplicada a expressões da forma `product (x[i], i, m, n)`.

negdistrib Variável de opção

Valor padrão: `true`

Quando `negdistrib` for `true`, -1 distribue sobre uma expressão. E.g., `-(x + y)` transforma-se em `- y - x`. Mudando o valor de `negdistrib` para `false` permitirá que `- (x + y)` seja mostrado como foi escrito. Isso algumas vezes é útil mas seja muito cuidadoso: como o sinalizador `simp`, isso é um sinalizador que você pode não querer escolher para `false` como algo natural ou necessário com excessão de usar localmente no seu Maxima.

negsumdispflag Variável de opção

Valor padrão: `true`

Quando `negsumdispflag` for `true`, `x - y` é mostrado como `x - y` em lugar de como `- y + x`. Escolhendo isso para `false` faz com que a verificação especial em visualização para a diferença das duas expressões não seja concluída. Uma aplicação é que dessa forma `a + %i*b` e `a - %i*b` podem ambos serem mostrados pelo mesmo caminho.

noeval Símbolo especial

`noeval` suprime a fase de avaliação de `ev`. Isso é útil em conjunção com outros comutadores e para fazer com que expressões sejam resimplificadas sem serem reavaliadas.

noun Declaração

`noun` é uma das opções do comando `declare`. Essa opção faz com que um função seja declarada como "noun" (substantivo), significando que ela não deve ser avaliada automaticamente.

noundisp Variável de opção

Valor padrão: `false`

Quando `noundisp` for `true`, substantivos (nouns) são mostrados com um apóstrofo. Esse comutador é sempre `true` quando mostrando definições de função.

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") que ocorrem na expressão que está sendo avaliada para verbos ("verbs"), i.e., 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 avaliados em ponto flutuante. Isso faz com que variáveis em `expr` às quais tenham sido dados valores numéricos a elas sejam substituídas pelos seus valores correspondentes. `numer` também escolhe o sinalizador `float` para `on`.

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

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

As expressões `expr_1`, ..., `expr_n` podem ser quaisquer expressões, 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` (veja logo abaixo), `evenfun`, `oddfun`, `commutative`, `symmetric`, `antisymmetric`, `nary`, `lassociative`, `rassociative`.

opsubst Variável de opção

Valor padrão: `true`

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

outative Declaração

`declare (f, outative)` diz ao simplificador do Maxima que fatores 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, aquele produto será particionado em fatores que são constantes e fatores que não são e os fatores constantes serão puxados para fora. E.g., `f(a*x)` simplificará para `a*f(x)` onde `a` é uma constante. Fatores 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`, i.e., `f (a*g(x), x)` irá simplificar para `a * f(g(x), x)` sendo `a` livre de `x`.

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

posfun Declaração

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

radcan (*expr*) Função

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

Para algumas expressões `radcan` é que consome inteiramente o tempo. Esse é o custo de explorar certos relacionamentos entre os componentes da expressão para simplificações baseadas sobre fatoração e expansões de fração-parcial de expoentes.

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 padrão: `true`

`radexpand` controla algumas simplificações de radicais.

Quando `radexpand` for `all`, faz com que n-ésimas raízes de fatores de um produto que são potências de n sejam puxados para fora do radical. E.g. Se `radexpand` for `all`, `sqrt (16*x^2)` simplifica para `4*x`.

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

- Se `radexpand` for `all` or `assume (x > 0)` tiver sido executado, `sqrt(x^2)` simplifica para `x`.
- Se `radexpand` for `true` e `domain` for `real` (isso é o padrão), `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 `domain` somente interessa quando `radexpand` for `true`.

radsubstflag Variável de opção

Valor padrão: `false`

`radsubstflag`, se `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. E.g., `g(g(a, b), g(c, d))` simplifica para `g(a, g(b, g(c, d)))`.

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

Simplificação Seqüêncial 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 padrão: `false`

Quando `simpsum` for `true`, o resultado de uma `sum` é simplificado. Essa simplificação pode algumas vezes estar apta a 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 todas as parcelas de uma adição que tem maiores e menores associações que diferem por constantes. O resultado é uma expressão contendo um somatório para cada escolha de cada tais somatórios adicionados a todos os termos extras apropriados que tiveram de ser extraídos para a forma dessa adição. `sumcontract` combina todas as somas compatíveis e usa-se os indices de uma as somas se puder, e então tenta formar um índice razoável se não for usar qualquer dos fornecidos.

Isso pode ser necessário fazer um `intosum (`*expr*`)` antes de `sumcontract`.

sumexpand Variável de opção

Valor padrão: `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 padrão: `true`

When `sumsplitfact` for `false`, `minfactorial` é aplicado após um `factcomb`.

symmetric Declaração

`declare (h, symmetric)` diz ao simplificador do Maxima que `h` é uma função simétrica. E.g., `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 Montando Gráficos

8.1 Definições para Montagem de Gráficos

in_netmath Variável

Valor padrão: `false`

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

openplot_curves (*list*, *rest_options*) Função

Pega uma lista de curvas tais como

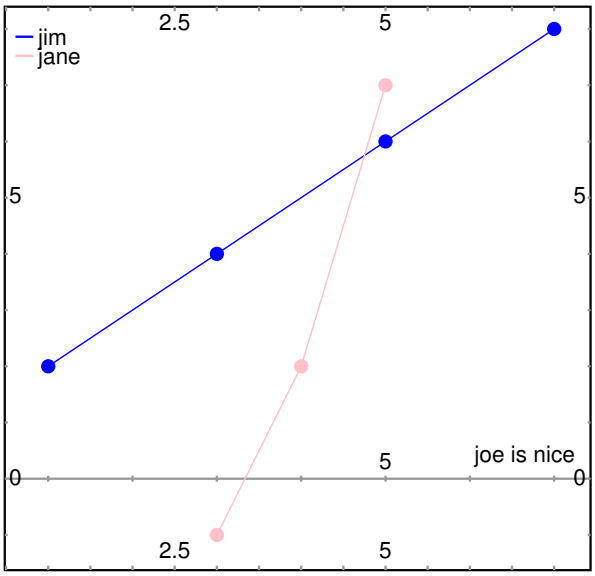
```
[[x1, y1, x2, y2, ...], [u1, v1, u2, v2, ...], ..]
```

ou

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

e monta seus gráficos. Isso é similar a xgraph_curves, mas não usa as rotinas open plot. Argumentos adicionais de símbolos podem ser dados tais como `"{xrange -3 4}"`. O exemplo adiante monta o gráfico de duas curvas, usando pontos grandes, rotulando-se o primeiro `jim` e o segundo rotulando-se `jane`.

```
(%i1) openplot_curves ([["{plotpoints 1} {pointsize 6}
      {label jim} {xaxislabel {joe is nice}}"],
      [1, 2, 3, 4, 5, 6, 7, 8], ["{label jane} {color pink }"],
      [3, -1, 4, 2, 5, 7]]);
```



Algumas outras palavras chave especiais são `xfun`, `color`, `plotpoints`, `linecolors`, `pointsize`, `nolines`, `bargraph`, `labelposition`, `xaxislabel`, e `yaxislabel`.

plot2d (*expr, range, ..., options, ...*) Função
plot2d (*parametric_expr*) Função
plot2d ([*expr_1, ..., expr_n*], *x_range, y_range*) Função
plot2d ([*expr_1, ..., expr_n*], *x_range*) Função
plot2d (*expr, x_range, y_range*) Função
plot2d (*expr, x_range*) Função
plot2d (*expr, x_range*) Função
plot2d ([*name_1, ..., name_n*], *x_range, y_range*) Função
plot2d ([*name_1, ..., name_n*], *x_range*) Função
plot2d (*name, x_range, y_range*) Função
plot2d (*name, x_range*) Função

Mostra a montagem de uma ou mais expressões como uma função de uma variável.

Em todos os casos, *expr* é uma expressão a ser montado o gráfico no eixo vertical como uma função de uma variável. *x_range*, a amplitude do eixo horizontal, é uma lista da forma `[`*variável*`,` *min*`,` *max*`]`, onde *variável* é uma variável que aparece em *expr*. *y_range*, e a amplitude do eixo vertical, é uma lista da forma `[y,` *min*`,` *max*`]`.

`plot2d (`*expr*`,` *x_range*`)` monta o gráfico *expr* como uma função da variável nomeada em *x_range*, sobre a amplitude especificada em *x_range*. Se a amplitude vertical não for alternativamente especificada por `set_plot_option`, essa é escolhida automaticamente. Todas as opções são assumidas terem valores padrão a menos que sejam alternativamente especificadas por `set_plot_option`.

`plot2d (`*expr*`,` *x_range*`,` *y_range*`)` monta o gráfico de *expr* como uma função de uma variável nomeada em *x_range*, sobre a amplitude especificada em *x_range*. O alcance vertical é escolhido para *y_range*. Todas as opções são assumidas terem valores padrão a menos que sejam alternativamente especificadas por `set_plot_option`.

`plot2d ([`*expr_1*`, ..., `*expr_n*`],` *x_range*`)` monta o gráfico *expr_1*, ..., *expr_n* como uma função da variável nomeada em *x_range*, sobre a amplitude especificada em *x_range*. Se a amplitude vertical não for alternativamente especificada por `set_plot_option`, essa é escolhida automaticamente. Todas as opções são assumidas terem valores padrão a menos que sejam alternativamente especificadas por `set_plot_option`.

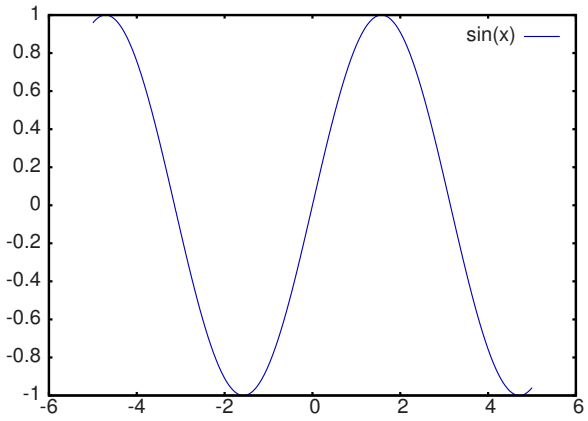
`plot2d ([`*expr_1*`, ..., `*expr_n*`],` *x_range*`,` *y_range*`)` monta o gráfico *expr_1*, ..., *expr_n* como uma função de uma variável nomedada em *x_range*, sobre a amplitude especificada em *x_range*. O alcance vertical é escolhido para *y_range*. Todas as opções são assumidas terem valores padrão a menos que sejam alternativamente especificadas por `set_plot_option`.

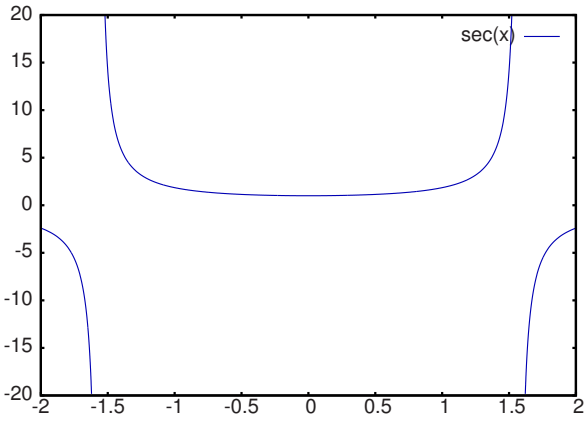
Uma função a ter seu gráfico montado pode ser especificada como o nome de uma função Maxima ou como o nome de uma função Lisp ou como um operador, como uma expressão lambda do Maxima, ou como uma expressão geral do Maxima. Se especificada como um nome ou como expressão lambda, a função deve ser uma função de um argumento.

Exemplos:

Montando um gráfico de uma expressão, e escolhendo alguns parâmetros comumente usados.

```
(%i1) plot2d (sin(x), [x, -5, 5])$
(%i2) plot2d (sec(x), [x, -2, 2], [y, -20, 20], [nticks, 200])$
```

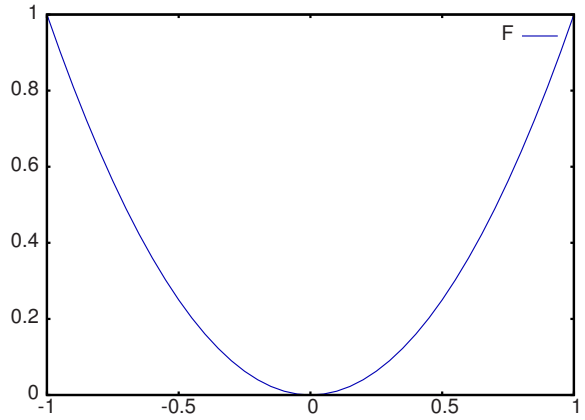


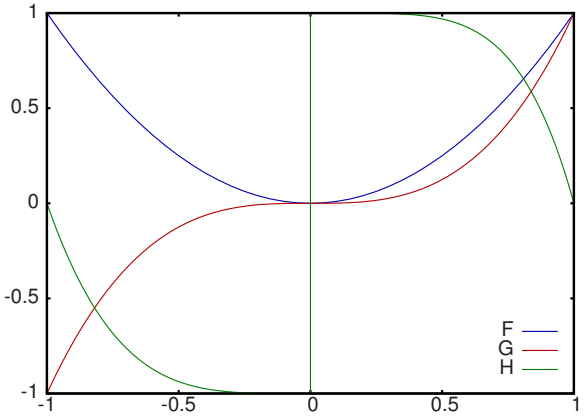


Montando gráfico de funções pelo nome.

```
(%i1) F(x) := x^2 $

(%i2) :lisp (defun |$g| (x) (m* x x x))

$g
(%i2) H(x) := if x < 0 then x^4 - 1 else 1 - x^5 $

(%i3) plot2d (F, [u, -1, 1])$
```

```
(%i4) plot2d ([F, G, H], [u, -1, 1])$
```





Em qualquer lugar onde pode existir uma expressão comum, pode existir uma expressão paramétrica: *parametric_expr* é uma lista da forma `[parametric,` *x_expr*`,` *y_expr*`,` *t_range*`,` *options*`]`. Aqui *x_expr* e *y_expr* são expressões de 1 variável *var* que é o primeiro elemento da amplitude *trange*. A montagem do gráfico mostra o caminho descrito pelo par `[`*x_expr*`,` *y_expr*`]` como *var* varia em *trange*.

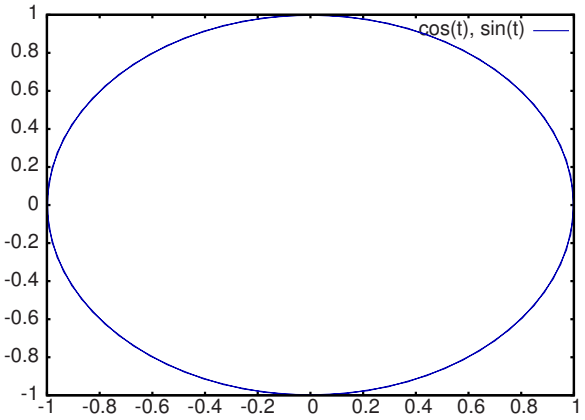
No exemplo seguinte, montaremos o gráfico de um círculo, então faremos a montagem do gráfico com somente poucos pontos usados, desse modo vamos pegar uma estrela, e inicialmente montaremos o gráfico juntamente com uma função comum de X.

Exemplos de gráficos paramétricos:

- Montar o gráfico de um círculo de forma paramétrica.

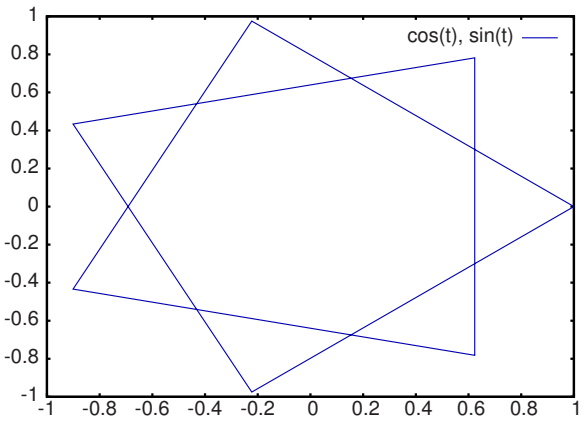
```
(%i1) plot2d ([parametric, cos(t), sin(t), [t, -%pi*2, %pi*2],
```

```
[nticks, 80]])$
```



- Monta o gráfico de uma estrela: liga oito pontos sobre a circunferência de um círculo.

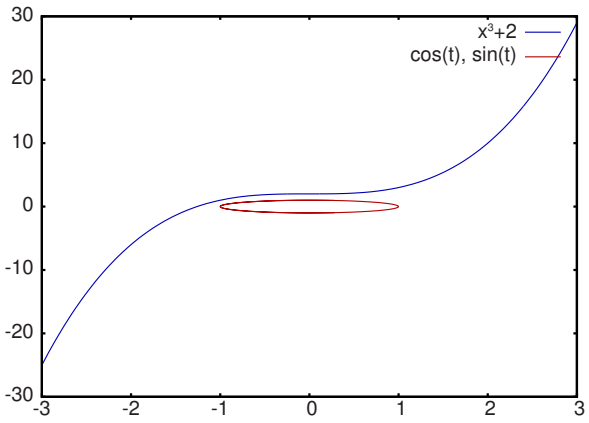
```
(%i2) plot2d ([parametric, cos(t), sin(t), [t, -%pi*2, %pi*2],
        [nticks, 8]])$
```



- Monta o gráfico de um polinômio cúbico da forma comum e de um círculo da forma paramétrica.

```
(%i3) plot2d ([x^3 + 2, [parametric, cos(t), sin(t), [t, -5, 5],
```

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



Expressões discretas podem também serem usadas ou em lugar de expressões comuns ou em lugar de expressões paramétricas: *discrete_expr* é uma lista da forma `[discrete,` *x_list*`,` *y_list*`]` ou da forma `[discrete,` *xy_list*`]`, onde *xy_list* é uma lista de pares `[`*x*`,`*y*`]`.

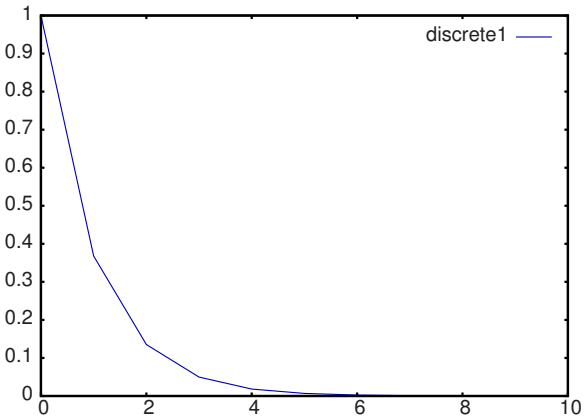
Exemplos de gráficos discretos:

- Cria algumas listas.

```
(%i1) xx:makelist(x,x,0,10)$
(%i2) yy:makelist(exp(-x*1.0),x,0,10)$
(%i3) xy:makelist([x,x*x],x,0,5)$
```

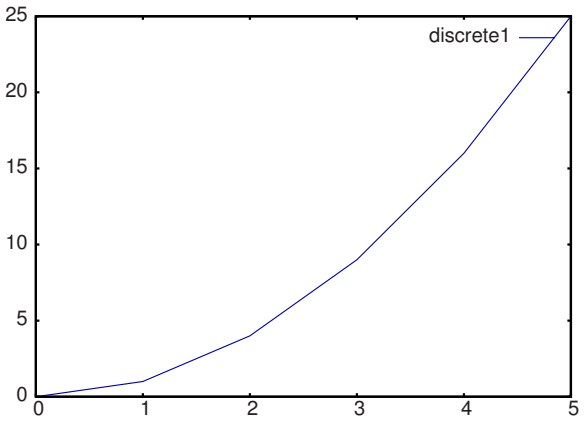
- Monta um gráfico com segmentos de reta.

```
(%i4) plot2d([discrete,xx,yy])$
```



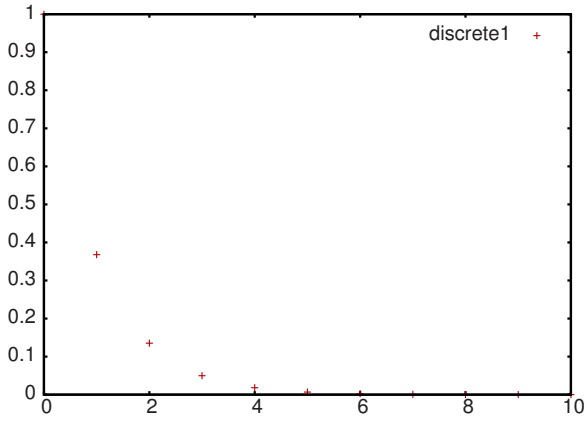
- Monta um gráfico com segmentos de reta, usando uma lista de pares.

```
(%i5) plot2d([discrete,xy])$
```



- Monta um gráfico com pontos.

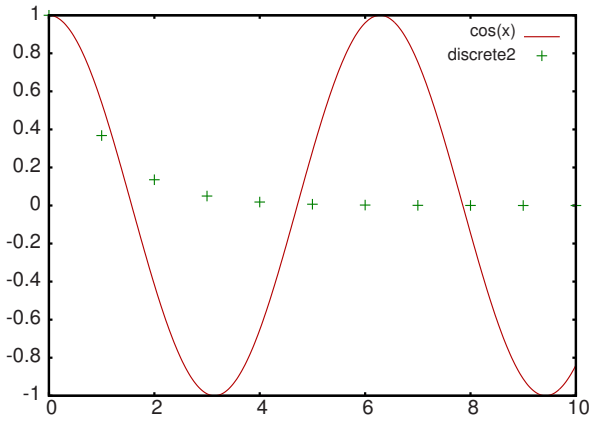
```
(%i6) plot2d([discrete,xx,yy],[gnuplot_curve_styles,
       ["with points"]])$
```



- Monta o gráfico da curva `cos(`x`)` usando linhas e (xx,yy) usando pontos.

```
(%i7) plot2d([cos(x),[discrete,xx,yy]],[x,0,10],
     [gnuplot_curve_styles,
```

```
["with lines","with points pointsize 3"]])$
```



Veja também `plot_options`, que descreve opções de montagem de gráfico e tem mais exemplos.

xgraph_curves (*list*) Função

transforma em gráfico a lista de 'grupos de pontos' dados em lista usando xgraph. Se o programa xgraph não estiver instalado, esse comando irá falhar.

Uma lista de grupos de pontos pode ser da forma

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

ou

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

Um grupo de pontos pode também conter símbolos que fornecem rótulos ou outra informação.

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

transforma em gráfico os três grupos de pontos com três curvas.

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

fizemos com que os grupos de pontos [e os próprios subseqüêntes], não possuam linhas entre os pontos, e para usar pixels largos. Veja a página de manual sobre o xgraph para especificar mais opções.

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

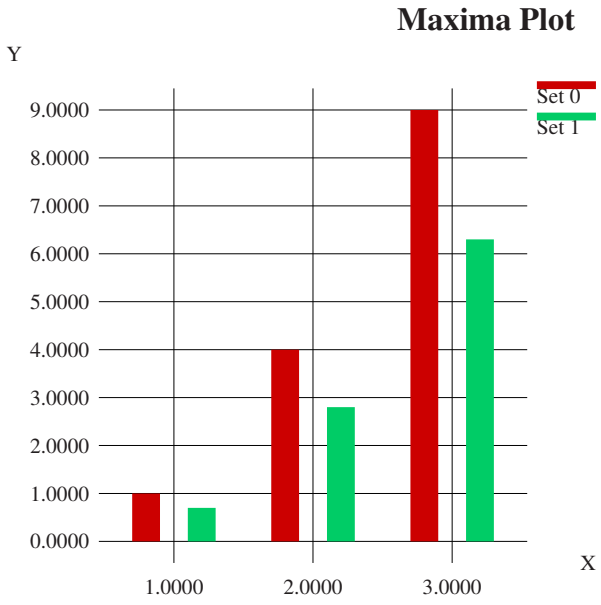
fizemos aí aparecer um "rótulo" de "x^2+y" para esse grupo de pontos em particular. As aspas, ", no início é que dizem ao xgraph isso é um rótulo.

```
pt_set: append ([concat ("TitleText: Dados da Amostra")], [x0, ...])$
```

fizemos o título principal do gráfico ser "Dados da Amostra" ao invés de "Maxima Plot".

Para fazer um gráfico em barras com largura de 0.2 unidades, e para montar o gráfico com duas possibilidades diferentes dos tais gráficos em barras:

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



Um arquivo temporário '`xgraph-out`' é usado.

plot_options Variável de sistema

Elementos dessa lista estabelecem as opções padrão para a montagem do gráfico. Se uma opção está presente em uma chamada a `plot2d` ou `plot3d`, esse valor tem precedência sobre a opção padrão. De outra forma, o valor em `plot_options` é usado. Opções padrão são atribuídas por `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 ítens.

As opções de montagem de gráfico que são reconhecidas por `plot2d` e `plot3d` são as seguintes:

- Opção: `plot_format` determina qual pacote de montagem de gráfico é usado por `plot2d` e `plot3d`.
  - Valor padrão: `gnuplot` Gnuplot é o padrão, e mais avançado, pacote de montagem de gráfico. Esse requer uma instalação externa do gnuplot.
  - Valor: `mgnuplot` Mgnuplot é um invólucro em torno do gnuplot baseado no Tk. Isso está incluído na distribuição do Maxima. Mgnuplot oferece uma GUI rudimentar para o gnuplot, mas tem menos recursos em geral que a interface texto. Mgnuplot requer uma instalação externa do gnuplot e Tcl/Tk.

  - Valor: `openmath` Openmath é um programa-GUI para montagem de gráfico baseado no Tcl/Tk. Isso está incluído na distribuição do Maxima.
  - Valor: `ps` Gera arquivos Postscript simples diretamente do Maxima. Saídas Postscript mais sofisticadas podem ser geradas pelo gnuplot, deixando a opção `plot_format` não especificada (para aceitar o padrão), e posicionando a opção `gnuplot_term` para `ps`.
- Opção: `run_viewer` controla se o visualizador apropriado para o formato da montagem do gráfico pode ou não poderá ser executado.
  - Valor padrão: `true` Executa o programa visualizador.
  - Valor: `false` Não executa o programa visualizador.
- `gnuplot_term` Prepara a saída tipo terminal para gnuplot.
  - Valor padrão: `default` A saída do Gnuplot é mostrada em uma janela gráfica separada.
  - Valor: `dumb` A saída do Gnuplot é mostrada no console do Maxima como uma aproximação "arte ASCII" para gráficos.
  - Valor: `ps` Gnuplot gera comandos na linguagem PostScript de descrição de páginas. Se a opção `gnuplot_out_file` está escolhida para *nomearquivo*, gnuplot escreve os comandos PostScript para *nomearquivo*. De outra forma, os comandos PostScript serão gravados no arquivo `maxplot.ps`.
  - Valor: qualquer outro especificação válida para o gnuplot term Gnuplot pode gerar em muitos outros formatos gráficos tais como png, jpeg, svg etc. Para criar gráficos em todos esses formatos o `gnuplot_term` pode ser escolhido para quaisquer nome (símbolo) de term suportado pelo gnuplot ou mesmo especificação de term do gnuplot com quaisquer opções válidas (seqüência de caracteres). Por exemplo `[gnuplot_term,png]` cria saídas no formato PNG (Portable Network Graphics) enquanto `[gnuplot_term,"png size 1000,1000"]` cria arquivos no formato PNG com tamanho de 1000x1000 pixels. Se a opção `gnuplot_out_file` for escolhida para *nomearquivo*, gnuplot escreve a saída para *nomearquivo*. De outra forma, a saida é gravada no no arquivo `maxplot.`*term*, onde *term* é um nome de terminal do gnuplot.
- Opção: `gnuplot_out_file` Escreve a saída gnuplot para um arquivo.
  - Valor padrão: `false` Nenhum arquivo de saída especificado.
  - Valor: *filename* Exemplo: `[gnuplot_out_file, "myplot.ps"]` Esse exemplo envia uma saída PostScript para o arquivo `myplot.ps` quando usada em conjunto com o terminal PostScript do gnuplot.
- Opção: `x` A amplitude horizontal padrão.

  ```
  [x, - 3, 3]
  ```

  Especifica a amplitude horizontal para [-3, 3].
- Opção: `y` A amplitude vertical padrão.

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

  Especifica a amplitude vertical para [-3, 3].

- Opção: `t` A amplitude padrão para o parâmetro em montagem de gráficos paramétricos.

  ```
  [t, 0, 10]
  ```

  Especifica a amplitude da variável paramétrica para [0, 10].
- Opção: `nticks` Número de pontos iniciais usado pela rotina adaptativa de montagem do gráfico.

  ```
  [nticks, 20]
  ```

  O padrão para `nticks` é 10.
- Opção: `adapt_depth` O número maximo de quebras usada pela rotina adaptativa de montagem do gráfico.

  ```
  [adapt_depth, 5]
  ```

  O padrão para `adapt_depth` é 10.
- Opção: `grid` Escolhe o número de pontos da grade para usar nas direções x e y para montagem de gráficos tridimensionais.

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

  Escolhe a grade para 50 por 50 pontos. A grade padrão é 30 por 30.
- Opção: `transform_xy` Permite que transformações sejam aplicadas à montagem de gráficos tridimensionais.

  ```
  [transform_xy, false]
  ```

  O padrão `transform_xy` é `false`. Se isso não é `false`, pode ser a saída de

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

  A transformação `polar_xy` é previamente definida no Maxima. Isso fornece a mesma transformação que

  ```
  make_transform ([r, th, z], r*cos(th), r*sin(th), z)$
  ```
- Opção: `colour_z` é específica para o formato `ps` de montagem de gráfico.

  ```
  [colour_z, true]
  ```

  O valor padrão para `colour_z` é `false`.
- Opção: `view_direction` Específico para o formato `ps` de montagem de gráfico.

  ```
  [view_direction, 1, 1, 1]
  ```

  O padrão `view_direction` é [1, 1, 1].

Existem muitas opções de montagem de gráficos específicas para gnuplot. Todas essas opções (exceto `gnuplot_pm3d`) são comandos gnuplot em estado natural, especificados como seqüências de caracteres. Consulte a documentação do gnuplot para maiores detalhes.

- Opção: `gnuplot_pm3d` Controla o uso do modo PM3D, que possui recursos avançados em 3D. PM3D está somente disponível no gnuplot em versões após 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 numérica na montagem do gráfico. O valor padrão para `gnuplot_preamble` é uma seqüência de caracteres vazia `""`.

Exemplo:

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

- Opção: `gnuplot_curve_titles` Controla os títulos dados na chave da montagem do gráfico. O valor padrão é `[default]`, que automaticamente escolhe o título de cada curva para a função cujo gráfico está sendo construído. Se não contiver `[default]`, `gnuplot_curve_titles` pode conter uma lista de seqüências de caracteres, cada uma das quais é `"title '`*title_string*`'"`. (Para disabilitar a chave de impressão de 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 seqüências de caracteres controlando a aparência das curvas, i.e., cor, largura, brilho, etc., para serem enviadas para o comando de montagem do gráfico do gnuplot. 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. Veja a documentação do gnuplot de `plot` para maiores informações.

  Exemplo:

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

- Opção: `gnuplot_default_term_command` O comando gnuplot para escolher o tipo de terminal para o terminal padrão. O valor padrão é a seqüência de caracteres vazia `""`, i.e., usa os padrões do gnuplot.

  Exemplo:

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

- Opção: `gnuplot_dumb_term_command` O comando gnuplot para escolher o tipo de terminal para o terminal dumb. O valor padrão é `"set term dumb 79 22"`, que faz a saída texto com 79 caracteres 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. Veja a documentação do gnuplot de `set term postscript` para mais informação.

  Exemplo:

  ```
  [gnuplot_ps_term_command,
  "set term postscript eps enhanced color solid 18"]
  ```

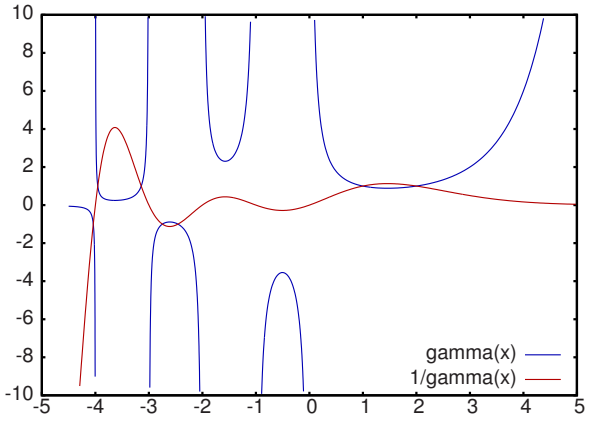
Exemplos:

- Grava um gráfico de `sin(x)` para o arquivo `sin.eps`.

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

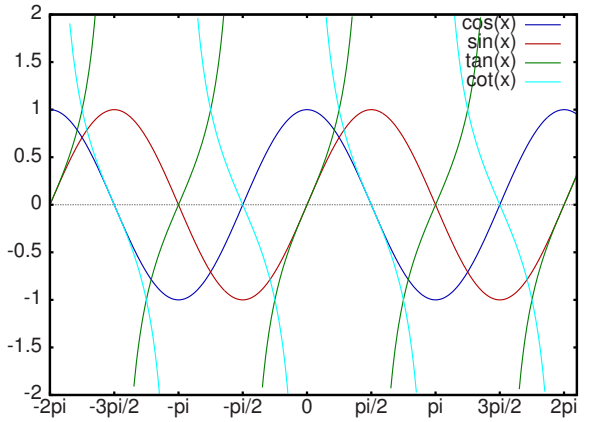
- Usa a opção do y para arrancar singularidades e a opção gnuplot_preamble para colocar a chave na parte inferior do gráfico em lugar de no topo.

```
(%i2) plot2d ([gamma(x), 1/gamma(x)], [x, -4.5, 5], [y, -10, 10],
                        [gnuplot_preamble, "set key bottom"])$
```



- Usa um muito complicado `gnuplot_preamble` para produzir elegantes rótulos para o eixo x. (Note que a seqüência de caracteres `gnuplot_preamble` deve ser fornecida inteiramente sem qualquer quebra de linha.)

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

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



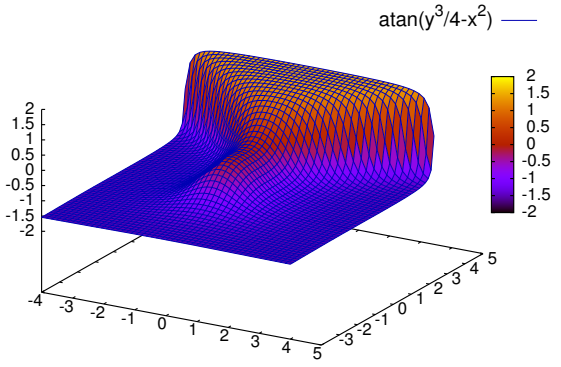
- Usa uma muito complicada `gnuplot_preamble` para produzir elegantes rótulos para o eixo x, e produzir saídas PostScript que pegam vantagens do formato

de texto avançado disponível no gnuplot. (Note que a seqüência de caracteres `gnuplot_preamble` deve ser fornecida inteiramente sem qualquer quebra de linha.)

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

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

- Uma montagem de gráfico tridimensional usando o terminal gnuplot pm3d.

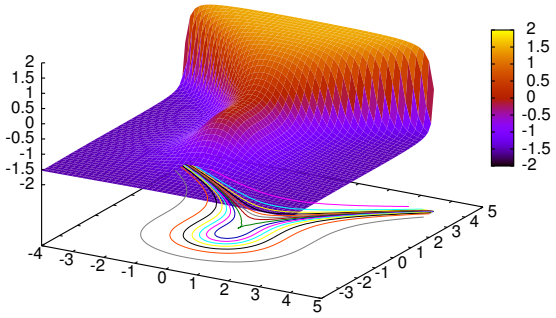
```
(%i7) plot3d (atan (-x^2 + y^3/4), [x, -4, 4], [y, -4, 4],
        [grid, 50, 50], [gnuplot_pm3d, true])$
```



- Uma montagem de gráfico tridimensional sem a malha e com contornos projetados no plano inferior.

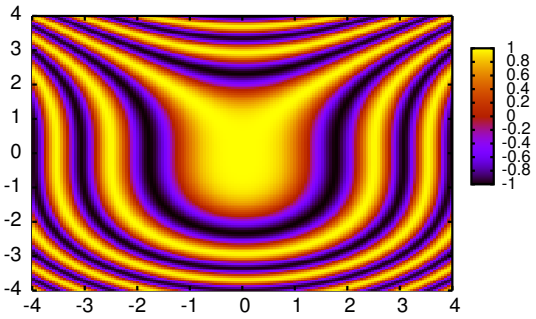
```
(%i8) my_preamble: "set pm3d at s;unset surface;set contour;\
set cntrparam levels 20;unset key"$
(%i9) plot3d(atan(-x^2 + y^3/4), [x, -4, 4], [y, -4, 4],
    [grid, 50, 50], [gnuplot_pm3d, true],
```

```
[gnuplot_preamble, my_preamble])$
```



- Uma montagem de gráfico onde o eixo z é representado apenas por cores. (Note que a seqüência de caracteres `gnuplot_preamble` deve ser fornecida inteiramente sem qualquer quebra de linha.)

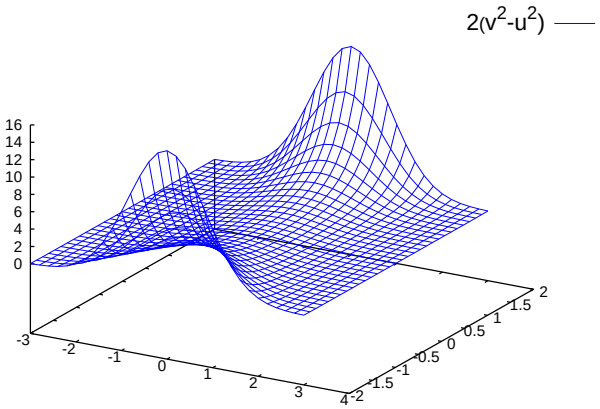
```
(%i10) plot3d (cos (-x^2 + y^3/4), [x, -4, 4], [y, -4, 4],
    [gnuplot_preamble, "set view map; unset surface"],
    [gnuplot_pm3d, true], [grid, 150, 150])$
```



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

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

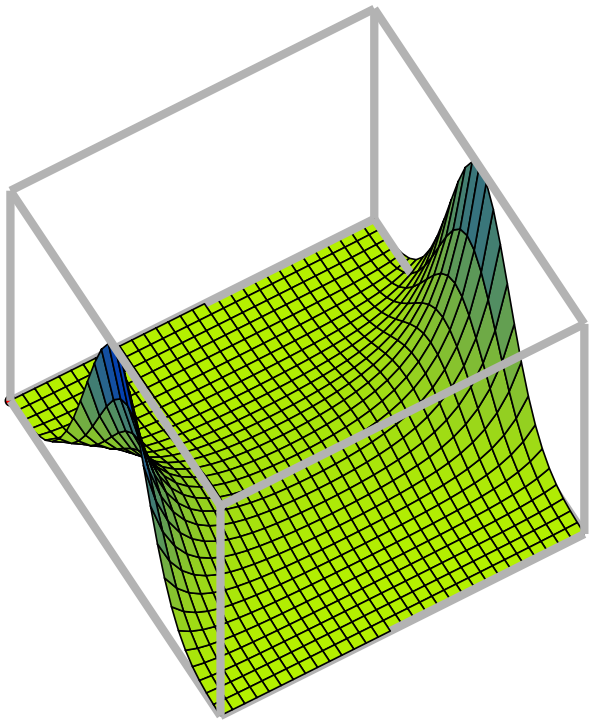
```
(%i1) plot3d (2^(-u^2 + v^2), [u, -3, 3], [v, -7, 7]);
```



monta o gráfico de `z = 2^(-u^2+v^2)` com `u` e `v` variando no intervalo fechado [-3,3] e no intervalo fechado de [-2,2] respectivamente, e com *u* sobre o eixo x, e `v` sobre o eixo y.

O mesmo gráfico pode ser visualizado usando openmath:

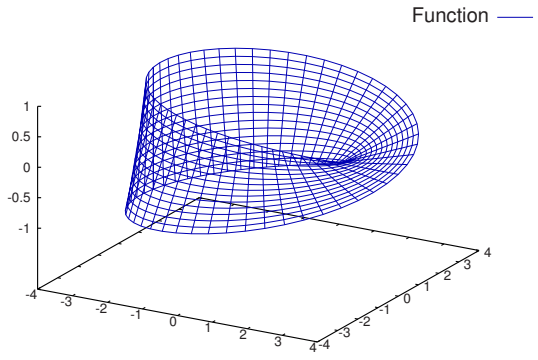
```
(%i2)  plot3d (2^(-u^2 + v^2), [u, -3, 3], [v, -2, 2],
               [plot_format, openmath]);
```



nesse caso o mouse pode ser usado para rotacionar a visualização para olhar na superfície de diferentes ngulos.

Um exemplo do terceiro modelo de argumento é

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

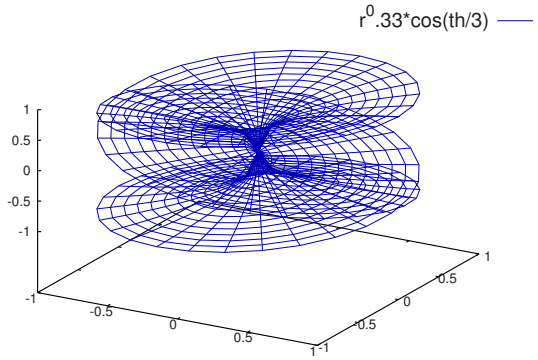


que monta o gráfico da banda de Moebius, parametrizada por três expressões fornecidas como o primeiro argumento para `plot3d`. Um adicional e opcional argumento `['grid, 50, 15]` fornece o número de retâgulos da grade na direção x e na direção y.

Uma função a ter seu gráfico montado pode ser especificada como o nome de uma função Maxima ou como o nome de uma função Lisp ou como um operador, como uma expressão lambda do Maxima, ou como uma expressão geral do Maxima. Se especificada como um nome ou como expressão lambda, a função deve ser uma função de um argumento.

Esse exemplo mostra uma montagem de gráfico da parte real de `z^1/3`.

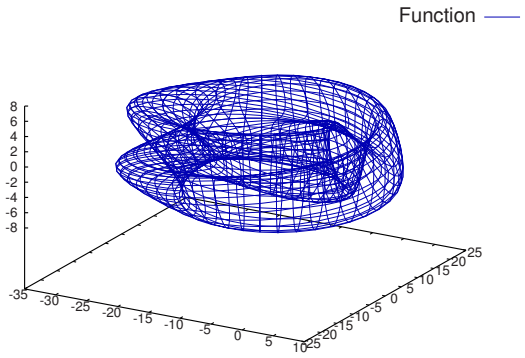
```
(%i4) plot3d (r^.33*cos(th/3), [r, 0, 1], [th, 0, 6*%pi],
      ['grid, 12, 80], ['transform_xy, polar_to_xy],
      ['view_direction, 1, 1, 1.4], ['colour_z, true]);
```



Aqui a opção `view_direction` indica a direção da qual nós pegamos a projeção. Nós atualmente fazemos isso de infinitamente distante, mas paralelo à linha de `view_direction` para a origem. Isso é correntemente usado somente em plot_format `ps`, uma vez que outros visualizadores permitem rotação interativa do objeto.

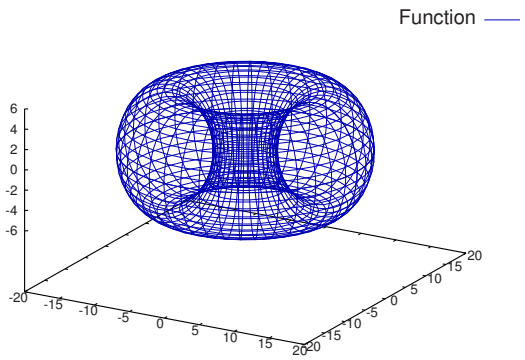
Outro exemplo é uma superfície de Klein:

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

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



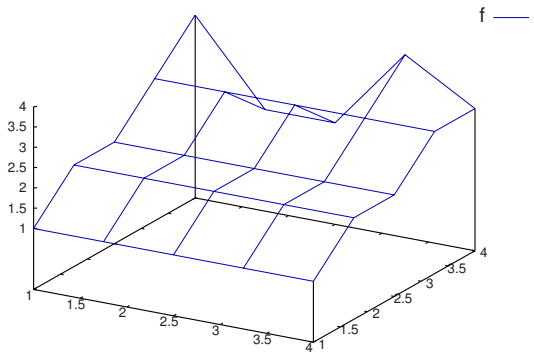
e um toro

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



Algumas vezes isso é necessário para definir uma função para montar o graico da expressão. Todos os argumentos para plot3d são avaliados antes de serem passados para plot3d, e então tentando fazer um expressão que faz apenas o que é preciso pode ser difícil, e é apenas mais fácil fazer uma função.

```
(%i13) M: matrix([1, 2, 3, 4], [1, 2, 3, 2], [1, 2, 3, 4],
       [1, 2, 3, 3])$
(%i14) f(x, y) := float (M [?round(x), ?round(y)])$
(%i15) plot3d (f, [x, 1, 4], [y, 1, 4], ['grid, 4, 4])$
```



Veja `plot_options` para mais exemplos.

make_transform (*vars*, *fx*, *fy*, *fz*) Função

Retornam uma função adequada para a função transformação em plot3d. Use com a opção de montagem de gráfico `transform_xy`.

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

é uma transformação para coordenadas polares.

plot2d_ps (*expr*, *range*) Função

Escreve para pstream uma seqüência de comandos PostScript que montam o graáfico de *expr* sobre *range*.

expr é uma expressão. *range* é uma lista da forma `[`*x*`,` *min*`,` *max*`]` na qual *x* é uma variável que aparece em *expr*.

Veja também `closeps`.

closeps () Função

Essa poderá usualmente ser chamada no final de um seqüência comandos de montagem de gráfico. Isso fecha o fluxo corrente de saída *pstream*, e altera esse para nil. Isso também pode ser chamado ao iniciar uma montagem de gráfico, para garantir que pstream será fechado se estiver aberto. Todos os comandos que escrevem para pstream, abrem isso se necessário. `closeps` é separada de outros comandos de montagem de gráfico, posteriormente podemos querer montar um gráfico com 2 amplitudes ou sobrepor muitas montagens de gráficos, e então devemos manter esse fluxo aberto.

set_plot_option (*opção*) Função

Atribui uma das varáveis globais para impressão. *option* 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`.

`set_plot_option` avalia seu argumento. `set_plot_option` retorna `plot_options` (após modificar um desses elementos).

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

Exemplos:

Modifica a malha (`grid`) e valores de `x`. Quando uma palavra chave em `plot_options` tem um valor atribuído, colocar um apóstrofo evita avaliação.

```
(%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], [view_direction, 1, 1, 1], [colour_z, false],
[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],
[view_direction, 1, 1, 1], [colour_z, false],
[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]]
```

psdraw_curve (*ptlist*) Função

Desenha uma curva conectando os pontos em *ptlist*. O último pode ser da forma `[x0, y0, x1, y1, ...]` ou da forma `[[x0, y0], [x1, y1], ...]`

A função `join` é útil fazendo uma lista dos x's e uma lista dos y's e colocando-os juntos.

psdraw_curve simplesmente invoca a mais primitiva função *pscurve*. Aqui está a definiçã:

```
(defun $psdraw_curve (lis)
```

```
(p "newpath")
($pscurve lis)
(p "stroke"))
```

pscom (*cmd*) Função

cmd é inserida no arquivo PostScript. Exemplo:

```
pscom ("4.5 72 mul 5.5 72 mul translate 14 14 scale");
```

9 Entrada e Saída

9.1 Introdução a Entrada e Saída

9.2 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
```

9.3 Arquivos

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

9.4 Definições para Entrada e Saída de Dados

`__` Variável de sistema

`__` é a expressão de entrada atualmente 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 arquivo processado por batch, __ tem o mesmo significado que na linha de comando interativa. Em um arquivo processado por load, __ está associado à expressão de entrada mais recentemente informada no prompt interativo ou em um arquivo de lote (batch); __ não é associado à expressões de entrada no arquivo que está sendo processado. Em particular, quando load (*nomedearquivo*) for chamado a partir da linha de comando interativa, __ é associado a load (*filename*) enquanto o arquivo 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 atual de entrada =", __), 0);
(%o3) g(x) := (print("Expressão atual 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 atual 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 arquivo processado por batch, _ tem o mesmo significado que na linha de comando interativa. Em um arquivo processado por load load, _ está associado à expressão de entrada mais recentemente avaliada na linha de comando interativa ou em um arquivo de lote; _ não está associada a expressões de entrada no arquivo 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 arquivo processado por `batch`, `%` tem o mesmo significado que na linha de comando interativa. Em um arquivo processado por `load`, `%` é associado à expressão de entrada mais recentemente calculada na linha de comando interativa ou em um arquivo de lote; `%` não está associada a expressões de saída no arquivo 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 padrão: `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 arquivos `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 arquivo processado por `batch`, `%th` possue o mesmo significado que na linha de comando interativa. Em um arquivo processado por `load`, `%th` refere-se a expressões de saída mais recentemente calculadas na linha de comando interativa ou em um arquivo de lote; `%th` não se refere a expressões de saída no arquivo 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 Seção 3.2 [Lisp e Maxima], página 7 para mais sobre esse ponto.

A notação `? word` (um ponto de interrogação seguido de uma palavra e separado desta por um espaço em branco) é equivalente a `describe ("word")`.

absboxchar — Variável de opção

Valor padrão: `!`

`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 padrão: `false`

`file_output_append` governa se funções de saída de arquivo anexam ao final ou truncam seu arquivo de saída. Quando `file_output_append` for `true`, tais funções anexam ao final de seu arquivo de saída. De outra forma, o arquivo de saída é truncado.

`save`, `stringout`, e `with_stdout` respeitam `file_output_append`. Outras funções que escrevem arquivos de saída não respeitam `file_output_append`. Em partivular, montagem de gráficos e traduções de funções sempre truncam seu arquivo 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`, exceto que o arquivo transcrito, se já existe, terá sempre alguma coisa adicionada ao seu final.

`closefile` fecha o arquivo transcrito que foi aberto anteriormente por `appendfile` ou por `writefile`.

batch (*filename*) Função

Lê expressões Maxima do arquivo *filename* e as avalia. `batch` procura pelo arquivo *filename* na lista `file_search_maxima`. Veja `file_search`.

filename compreende uma seqüência de expressões Maxima, cada uma terminada com `;` ou `$`. A varável especial `%` e a função `%th` referem-se a resultados prévios dentro do arquivo. O arquivo pode incluir construções `:lisp`. Espaços, tabulações, e o caracter de nova linha no arquivo serão ignorados. um arquivo 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 arquivo a menos que exista um erro. Se uma entrada de usuário for requisitada (by `asksign` ou `askinteger`, por exemplo) `batch` interrompe para coletar 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 arquivo. O arquivo não pode incluir construções `:lisp`.

`batchload` retorna o caminho de *filename*, como uma seqüência de caracteres. `batchload` avalia seu argumento.

Veja também `batch` e `load`.

closefile () Função

Fecha o arquivo 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` arquivo. Você pode diminuir muitas expressões juntas pelo uso de `collapse ([`*expr_1*`, ..., `*expr_n*`])`. Similarmente, você 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 seqüência 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 seqüência 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 seqüência de caracteres. Ao contrário de `concat`, os argumentos arrumados *não* precisam ser atômicos.

O resultado é uma seqüência 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 objetivo de ter resultados intermediários 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 padrão: `true`

Quando `display2d` é `false`, O console visualizador é unidimensional ao invés de bidimensional.

display_format_internal Variável de opção

Valor padrão: `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 fatores 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 padrão: 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 padrão: `[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 padrão: `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 arquivo *filename* e retorna o caminho para o arquivo (como uma seqüência de caracteres) se ele for achado; de outra forma `file_search` retorna `false`. `file_search (`*filename*`)` procura nos diretórios 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 atual passado existe, antes de tentar coincidir esse nome atual com o modelo "coringa" de busca do arquivo. Veja `file_search_maxima` concernente a modelos de busca de arquivos.

O argumento *filename* pode ser um caminho e nome de arquivo, ou apenas um nome de arquivo, ou, se um diretório de busca de arquivo inclui um modelo de busca de arquivo, apenas a base do nome de arquivo (sem uma extensão). Por exemplo,

```
file_search ("/home/wfs/special/zeta.mac");
file_search ("zeta.mac");
file_search ("zeta");
```

todos acham o mesmo arquivo, assumindo que o arquivo exista e `/home/wfs/special/###.mac` está em `file_search_maxima`.

`file_search` (*filename*, *pathlist*) procura somente nesses diretórios especificados por *pathlist*, que é uma lista de seqüências de caracteres. O argumento *pathlist* substitui os diretórios de busca padrão, então se a lista do caminho é dada, `file_search` procura somente nesses especificados, e não qualquer dos diretórios padrão de busca. Mesmo se existe somente um diretório em *pathlist*, esse deve ainda ser dado como uma lista de um único elemento.

O usuário pode modificar o diretório de busca padrão. Veja `file_search_maxima`.

`file_search` é invocado por `load` com `file_search_maxima` e `file_search_lisp` como diretórios 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 diretórios a serem procurados por `load`, `demo`, e algumas outras funções do Maxima. O valor padrão dessas variáveis nomeia vários diretórios na instalaçã padrão do Maxima.

O usuáro pode modificar essas variáveis, quer substituindo os valores padrão ou colocando no final diretórios 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 diretórios adicionais. Isso pode ser conveniente para colocar assim uma expressão no arquivo `maxima-init.mac` de forma que o caminho de busca de arquivo é atribuído automaticamente quando o Maxima inicia.

Multiplas extensões de arquivo e e multiplos caminhos podem ser especificados por construções "coringa" especiais. A seqüência de caracteres `###` expande a busca para além do nome básico, enquanto uma lista separada por vírgulas e entre chaves `{foo,bar,baz}` expande em multiplas seqüências 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 arquivo. *filename* não precisa referir-se a um arquivo atual; nenhuma tentativa é feita para abrir o arquivo 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 seqüência 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 padrão: 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 padrão: `%i`

`inchar` é o prefixo dos rótulos de expressões fornecidas pelo usuário. Maxima automaticamente constrói um rótulo para cada expressão de entrada por concatenação de `inchar` e `linenum`. A `inchar` pode ser atribuído qualquer seqüência 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 intermediária 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 padrão: `%t`

`linechar` é o refixo de rótulos de expressões intermediárias gerados pelo Maxima. Maxima constrói um rótulo para cada expressão intermediária (se for mostrada) pela concatenação de `linechar` e `linenum`. A `linechar` pode ser atribuído qualquer seqüência de caracteres ou símbolo, não necessáriamente um caractere simples.

Expressões intermediárias podem ou não serem mostradas. See `programmode` e `labels`.

linel Variável de opção

Valor padrão: 79

`linel` é a largura assumida (em caracteres) do console para o propósito de mostrar expressões. A `linel` pode ser atribuído qualquer valor pelo usuário, 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 afetadas por `linel`.

lispdisp Variável de opção

Valor padrão: `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 interrogaçõ na frente.

Exemplos:

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

load (*nomedearquivo*) Função

Avalia expressões em *nomedearquivo*, dessa forma conduzindo variáveis, funções, e outros objetos dentro do Maxima. A associação de qualquer objeto existente é substituída pela associação recuperada de *nomedearquivo*. Para achar o arquivo, `load` chama `file_search` com `file_search_maxima` e `file_search_lisp` como diretórios de busca. Se `load` obtém sucesso, isso retorna o nome do arquivo. De outra forma `load` imprime uma mensagem e erro.

`load` trabalha igualmente bem para códigos Lisp e códigos Maxima. Arquivos 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 arquivos Lisp e `batchload` para carregar arquivos Maxima.

`load` não reconhece construções `:lisp` em arquivos do Maxima, e quando processando *nomedearquivo*, as variáveis globais `_`, `__`, `%`, e `%th` possuem as mesmas associações que possuiam quando `load` foi chamada.

Veja também `loadfile`, `batch`, `batchload`, e `demo`. `loadfile` processa arquivos Lisp; `batch`, `batchload`, e `demo` processam arquivos Maxima.

Veja `file_search` para mais detalhes sobre o mecanismo de busca de arquivos.

`load` avalia seu argumento.

loadfile (*nomedearquivo*) Função

Avalia expressões Lisp em *nomedearquivo*. `loadfile` não invoca `file_search`, então `nomedearquivo` deve obrigatóriamente incluir a extensão do arquivo e tanto quanto o caminho como necessário para achar o arquivo.

`loadfile` pode processar arquivos criados por `save`, `translate_file`, e `compile_file`. O usuário pode achar isso mais conveniente para usar `load` em lugar de `loadfile`.

`loadfile` avalia seu argumento, então `nomedearquivo` deve obrigatóriamente ser uma seqüência de caracteres literal, não uma variável do tipo seqüência de caracteres. O operador apóstrofo-apóstrofo `''` não aceita avaliação.

loadprint Variável de opção

Valor padrão: `true`

`loadprint` diz se deve imprimir uma mensagem quando um arquivo é chamado.

- Quando `loadprint` é `true`, sempre imprime uma mensagem.
- Quando `loadprint` é `'loadfile`, imprime uma mensagem somente se um arquivo é chamado pela função `loadfile`.
- Quando `loadprint` é `'autoload`, imprime uma mensagem somente se um arquivo é automaticamente carregado. Veja `setup_autoload`.

- Quando `loadprint` é `false`, nunca imprime uma mensagem.

obase Variável de opção

Valor padrão: 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 padrão: `%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 seqüência 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 padrão: `false`

Projetistas de pacotes que usam `save` ou `translate` para criar pacotes (arquivos) para outros usarem podem querer escolher `packagefile: true` para prevenir qu informações sejam acrescentadas à lista de informações do Maxima (e.g. `values`, `funções`) exceto onde necessário quando o arquivo é carregado. Nesse caminho, o conteúdo do pacote não pegará no caminho do usuário quando ele adicionar seus próprios dados. Note que isso não resolve o problema de possíveis conflitos de nome. Também note que o sinalizador simplesmente afeta o que é saída para o arquivo pacote. Escolhendo o sinalizador para `true` é também útil para criar arquivos de init do Maxima.

pfeformat Variável de opção

Valor padrão: `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 intermediária.

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 arquivo, veja `printfile`.

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

`rmxchar` é the caractere desenhado lado direito de uma matriz.

Veja também `lmxchar`.

save (*filename*, *name_1*, *name_2*, *name_3*, ...) Função
save (*filename*, *values*, *functions*, *labels*, ...) Função
save (*filename*, [*m*, *n*]) Função
save (*filename*, *name_1*=*expr_1*, ...) Função
save (*filename*, *all*) Função

Armazena os valores correntes de *name_1*, *name_2*, *name_3*, ..., em *filename*. Os argumentos são os nomes das variáveis, funções, ou outros objetos. 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 arquivo de saída. Quando `file_output_append` for `true`, `save` anexa ao final doarquivo de saída. De outra forma, `save` trunca o arquivo de saída. Nesse caso, `save` cria o arquivo 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 usuário.

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*`, `*name_1*`=`*expr_1*`, `*name_2*`=`*expr_2*`, ...)` armazena os valores de *expr_1*, *expr_2*, ..., com nomes *name_1*, *name_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 usuário, 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 usuário; veja `myoptions`.

`save` avalia seus argumentos. *filename* deve obrigatóriamente ser uma seqüência de caracteres, não uma variável tipo seqüência de caracteres. O primeiro e o último rótulos a salvar, se especificado, devem obrigatóriamente serem inteiros. O operador apóstrofo-apóstrofo `''` avalia uma variável tipo seqüência de caracteres para seu valor seqüência 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 padrão: `true`

Quando `savedef` é `true`, a vesão Maxima de uma função de usuário 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 objetos 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 canônica (CRE) na expressão `expr`.

Veja também `ratvars`.

stardisp Variável de opção

Valor padrão: `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 seqüência de caracteres, e dessa forma não pode ser usada em um cálculo.

stringdisp Variável Lisp

Valor padrão: `false`

Quando `?stringdisp` for `true`, seqüências de caracteres serão mostradas contidas em aspas duplas. De outra forma, aspas não são mostradas.

`?stringdisp` é sempre `true` quando mostrando uma definição de função.

`?stringdisp` é uma variável Lisp, então deve ser escrita com um ponto de interrogação `?` na frente.

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 arquivo na mesma forma de expressões que foram digitadas para entrada. O arquivo 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 arquivo de saída. Quando `file_output_append` for `true`, `stringout` anexa ao final do arquivo de saída. De outra forma, `stringout` trunca o arquivo de saída. Nesse caso, `stringout` cria o arquivo de saída se ele não existir ainda.

A forma geral de `stringout` escreve os valores de um ou mais expressões para o arquivo 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 arquivo.

A forma especial `stringout (`*filename*`, functions)` escreve todas as funções definidas pelo usuário (nomeadas pela lista global `functions`) para o arquivo.

A forma especial `stringout (`*filename*`, values)` escreve todas as variáveis atribuídas pelo usuário (nomeadas pela lista global `values`) para o arquivo. 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, momearquivo*) Função
tex (*label, nomearquivo*) 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 direta 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*, *nomearquivo*) anexa ao final uma representação TeX de *expr* no arquivo *nomearquivo*. `tex` não avalia o argumento *nomearquivo*; apóstrofo-apóstrofo `''` força a avaliação so argumento.

`tex` (*rótulo*, *nomearquivo*) anexa ao final uma representação TeX da expressão chamada de *rótulo*, com um rótulo de equação, ao arquivo *nomearquivo*.

`tex` não avalia o argumento *nomearquivo*; 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

Escolhe 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 seqüência de caracteres *s* dentro da saída TeX em lugar de *a*.

`texput (`*a*`, `*s*`, `*operator_type*`)`, onde *operator_type* é `prefix`, `infix`, ou `postfix` 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:

```
(%i1) texput (me,"\\mu_e");
(%o1)                          \mu_e
(%i2) tex (me);
$$\mu_e$$
(%o2)                          false
(%i3) texput (lcm, "\\mathrm{lcm}");
(%o3)                       \mathrm{lcm}
(%i4) tex (lcm (a, b));
$$\mathrm{lcm}\left(a , b\right)$$
(%o4)                          false
(%i5) prefix ("grad");
(%o5)                          grad
(%i6) texput ("grad", " \\nabla ", prefix);
(%o6)                           180
(%i7) tex (grad f);
$$ \nabla f$$
(%o7)                          false
(%i8) infix ("~");
(%o8)                            ~
(%i9) texput ("~", " \\times ", infix);
(%o9)                           180
(%i10) tex (a ~ b);
$$a \times b$$
(%o10)                         false
(%i11) postfix ("@");
(%o11)                           @
(%i12) texput ("@", "!!", postfix);
(%o12)                          160
(%i13) tex (x @);
$$x!!$$
(%o13)                         false
(%i14) matchfix ("<<", ">>");
(%o14)                           <<
(%i15) texput ("<<", [" \\langle ", " \\rangle "], matchfix);
(%o15)              \langle ( \rangle , false)
(%i16) tex (<<a>>);
```

```
$$ \langle a \rangle $$
(%o16)                         false
(%i17) tex (<<a, b>>);
$$ \langle a , b \rangle $$
(%o17)                         false
(%i18) texput ("<<", [" \\langle ", " \\rangle ", " \\, | \\,"], matchfix);
(%o18)            \langle ( \rangle ,  \, | \,)
(%i19) tex (<<a>>);
$$ \langle a \rangle $$
(%o19)                         false
(%i20) tex (<<a, b>>);
$$ \langle a \, | \,b \rangle $$
(%o20)                         false
```

system (*comando*) Função

Executa *comando* como um processo separado. O comando é passado ao shell padraõ 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 freqüência que você 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 objetivo de fazer com que a impressão do gráfico seja concluída em segundo plano (retornando o controle para o Maxima) e remover o arquivo temporário após isso ter sido concluído faça:

```
system("(xgraph -bar -brw .7 -nl < _hist.out;  rm -f _hist.out)&")
```

ttyoff Variável de opção

Valor padrão: `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 afetadas 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 arquivo de saída. Quando `file_output_append` for `true`, `with_stdout` anexa ao final do arquivo de saída. De outra forma, `with_stdout` trunca o arquivo de saída. Nesse caso, `with_stdout` cria o arquivo 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 usuário e Maxima é então gravada nesse arquivo, 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 arquivo 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 arquivo transcrito pode ser substituído, ou o arquivo pode receber um anexo. `appendfile` sempre anexa para o arquivo 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 arquivo 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 padrão: 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 padrão: `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 padrão: `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

Fatorial 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 padrão: `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 padrão: 16

`fpprec` é o número de algarismos significativos para aritmética sobre grandes números em ponto flutuante `fpprec` não afeta cálculos sobre números em ponto flutuante comuns.

Veja também `bfloat` e `fpprintprec`.

fpprintprec Variável de Opção

Valor padrão: 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.

?round (*x*) Função Lisp
?round (*x*, *divisor*) Função Lisp

Arredonda o ponto flutuante *x* para o inteiro mais próximo. O argumento obrigatoriamente deve ser um ponto flutuante comum, não um grandes números em ponto flutuante. A `?` começando o nome indica que isso é uma função Lisp.

```
(%i1) ?round (-2.8);
(%o1)                          - 3
```

?truncate (*x*) Função Lisp
?truncate (*x*, *divisor*) Função Lisp

Trunca o ponto flutuante *x* na direção do 0, para transormar-se em um inteiro. O argumento deve ser um número em ponto flutuante comum, não um grandes números em ponto flutuante. A `?` começando o nome indica que isso é uma função Lisp.

```
(%i1) ?truncate (-2.8);
(%o1)                          - 2
(%i2) ?truncate (2.4);
(%o2)                           2
(%i3) ?truncate (2.8);
(%o3)                           2
```

11 Contextos

11.1 Definições para Contextos

activate (*context_1, ..., context_n*) Função

Ativa os contextos *context_1*, ..., *context_n*. Os fatos nesses contextos estão então disponíveis para fazer deduções e recuperar informação. Os fatos nesses contextos não são listadas através de `facts ()`.

A variável `activecontexts` é a lista de contextos que estão ativos pelo caminho da função `activate`.

activecontexts Variável de sistema

Valor padrão: `[]`

`activecontexts` é a lista de contextos que estão ativos pelo caminho da função `activate`, em oposição a sendo ativo 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* `and` *pred_2*`)` and `not (`*pred_1* `or` *pred_2*`)` não são reconhecidas.

O mecanismo de dedução do Maxima não é muito forte; exitem conseqüências 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 padrão: `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 padrão: `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

logarítmo `log(`x`)`, `asksign` sempre solicita uma entrada ao usuário, 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 padrão: `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 padrão: `initial`

`context` nomeia a coleção de fatos mantida através de `assume` e `forget`. `assume` adiciona fatos à coleção nomeada através de `context`, enquanto `forget` remove fatos.

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 ativado automaticamente.

Veja `contexts` para uma descrição geral do mecanismo de contexto.

contexts Variável de opção

Valor padrão: `[initial, global]`

`contexts` é uma lista dos contextos que existem atualmente, incluindo o contexto ativo atualmente.

O mecanismo de contexto torna possível para um usuário associar e nomear uma porção selecionada de fatos, chamada um contexto. Assim que isso for concluído, o usuário pode ter o Maxima assumindo ou esquecendo grande quantidade de fatos meramente através da ativação ou desativação seu contexto.

Qualquer átomo simbólico pode ser um contexto, e os fatos 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 fatos naquele contexto estão "ativos" (significando que eles são usados em deduções e recuperados) como estão também todos os fatos em qualquer contexto que for um subcontexto do contexto ativo.

Quando um novo Maxima for iniciado, o usuário 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 fatos no contexto especificado.

Se *item* não for o nome de um contexto, `facts (item)` retorna uma lista de fatos conhecidos sobre *item* no contexto atual. Fatos que estão atuvos, mas em um diferente contexto, não são listados.

`facts ()` (i.e., sem argumento) lista o contexto atual.

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 usuário.

`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 atual, o novo contexto atual irá tornar-se o primeiro subcontexto disponível do contexto atual 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 atualmente, ou porque ele é um subcontexto do contexto atual, 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 ativo atualmente.

`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 atual é assumido.

12 Polinômios

12.1 Introdução a Polinômios

Polinômios são armazenados no Maxima ou na forma geral ou na forma de Expressões Racionais Canônicas (CRE). Essa última é uma forma padrão, e é usada internamente por operações tais como `factor`, `ratsimp`, e assim por diante.

Expressões Racionais Canônicas constituem um tipo de representação que é especialmente adequado para polinômios expandidos e funções racionais (também para polinômios parcialmente fatorados 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. Polinômios são representados recursivamente por uma lista consistindo da variável principal seguida por uma série de pares de expressões, uma para cada termo do polinômio. 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 polinômio 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 fato 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 usuário 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 polinômios, onde o numerador e o denominador não possuem fatores comuns, e o denominador for positivo. A forma interna é essencialmente um par de polinômios (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 polinômios. 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 Polinômios

algebraic Variável de opção

Valor Padrão: `false`

`algebraic` deve ser escolhida para `true` com o objetivo de que a simplificação de inteiros algébricos tenha efeito.

berlefact Variável de opção

Valor Padrão: `true`

Quando `berlefact` for `false` então o algorítmo de fatoração de Kronecker será usado. De outra forma o algorítmo 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 fatorar *expr* com o objetivo 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 polinômio *p_1* na variável *x_n* (isso é o conteúdo) e cujo segundo elemento é o polinômio *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 polinômio *p_1* dividido pelo polinômio *p_2*, na variável principal do polinômio, *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) pegando 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 polinômios *p_1*, *p_2*, *p_3*, ... e cujos restantes elementos são os polinômios divididos pelo mdc. Isso sempre usa o algoritmo `ezgcd`.

facexpand Variável de opção

Valor Padrão: `true`

`facexpand` controla se os fatores irredutíveis retornados por `factor` estão na forma expandida (o padrão) ou na forma recursiva (CRE normal).

factcomb (*expr*) Função

Tenta combinar os coeficientes de fatoriais em *expr* com os próprios fatoriais 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

Fatora a expressão *expr*, contendo qualquer número de variáveis ou funções, em fatores irredutíveis sobre os inteiros. `factor (`*expr*`, p)` fatora *expr* sobre o campo dos inteiros com um elemento adjunto cujo menor polinômio é p.

`factor` usa a função `ifactors` para fatorar inteiros.

`factorflag` se `false` suprime a fatoração de fatores inteiros de expressões racionais.

`dontfactor` pode ser escolhida para uma lista de variáveis com relação à qual fatoração não é para ocorrer. (Essa é inicialmente vazia). Fatoração 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 fatores de uma expressão que é um produto de fatores seja guardada por certas funções com o objetivo de aumentar a velocidade de futuras fatorações de expressões contendo alguns dos mesmos fatores.

`berlefact` se `false` então o algoritmo de fatoração de Kronecker será usado de outra forma o algoritmo de Berlekamp, que é o padrão, será usado.

`intfaclim` se `true` maxima irá interromper a fatoração de inteiros se nenhum fator for encontrado após tentar divisões e o método rho de Pollard. Se escolhida para `false` (esse é o caso quando o usuário chama `factor` explicitamente), a fatoração completa do inteiro será tentada. A escolha do usuário 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 fatorando 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 fatoração de fatores 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 fatorado.

factorsum (*expr*) Função

Tenta agrupar parcelas em fatores de *expr* que são adições em grupos de parcelas tais que sua adição é fatorável. `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 polinômios *p_1* e *p_2* usando um algorítmo especial para a multiplicação de polinômios. `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*) pega 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` exceto 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` exceto 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 algorítmo é empregado. Escolhendo `gcd` para `ez`, `subres`, `red`, ou `spmod` seleciona o algorítmo `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 polinômios homogêneos é recomendado que `gcd` igual a `subres` seja usado. Para pegar 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

algorítmo, 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 canônica (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 polinômios de uma variável, ou de outra forma polinômios 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 polinômios de uma única variável com coeficientes sendo funções racionais em outras variáveis.

`gcdex` implementa o algorítmo Euclideano, onde temos a seqüência 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

Fatora 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). Fatorações são normalizadas fazendo *a* e *b* não negativos.

gfactor (*expr*) Função

Fatora o polinômio *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 padrão: true

Se `true`, maxima irá interromper a fatoração de inteiros se nenhum fator for encontrado após tentar divisões e o método rho de Pollard e a fatoração não irá ser completada.

Quando `intfaclim` for `false` (esse é o caso quando o usuário chama `factor` explicitamente), a fatoração completa irá ser tentada. `intfaclim` é escolhida para `false` quando fatores são calculados em `divisors`, `divsum` e `totient`.

Chamadas internas a `factor` respeitam o valor especificado pelo usuário para `intfaclim`. Setting `intfaclim` to `true` may reduce `intfaclim`. Escolhendo `intfaclim` para `true` podemos reduzir o tempo gasto fatorando 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 canônica (CRE).

lratsubst (*L*, *expr*) Função

é análogo a `subst (`*L*`,` *expr*`)` exceto 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 canônica (CRE) quando `modulus` for colocado em seu valor original, então você pode precisar repetir o rat *expr*, e.g., `expr: rat (ratdisrep (expr))`, com o objetivo de pegar resultados corretos.

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 conseqüências 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 polinômio *p* na variável *x* em uma composição funcional de polinômios 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 polinômio *p_1* dividido pelo polinômio *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 canônica (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 fatorada para CRE. Durante operações racionais a expressão é mantida como totalmente fatorada como

possível sem uma chamada ao pacote de fatoração (`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 fatores dentro de cada parte podem não ser relativamente primos.

`ratprint` se `false` suprime a impressão de mensagens informando o usuário 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 canônicas (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 (exceto 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 canônica (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 polinômios ou um polinômio 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 canônica (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 canônica (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 canônica (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 canônica (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 canônicas (CRE) são manipuladas na forma parcialmente fatorada.

Durante operações racionais a expressão é mantida como completamente fatorada 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 fator 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 fatorados 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 canônica (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 canônica (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 usuário 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 polinômios na forma recursiva, isto é, os coeficientes de variável principal são polinômios 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 afetado por algum dos sinalizadores que afetam `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 canônicas (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 canônicas (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 polinômio *p_1* dividido pelo polinômio *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 polinômios *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 fator em comum não constante.

Se *p_1* ou *p_2* puderem ser fatorados, 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` pega 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 fatores de uma expressão que é um produto de fatores sejam gravados por certas funções com o objetivo de aumentar a velocidade em posteriores fatorações de expressões contendo algum desses mesmos fatores.

sqfr (*expr*) Função

é similar a `factor` exceto que os fatores do polinômio são "livres de raízes". Isto é, eles possuem fatores somente de grau um. Esse algoritmo, que é também usado no primeiro estágio de `factor`, utiliza o fato que um polinômio tem em comum com sua n'ésima derivada todos os seus fatores de grau maior que n. Dessa forma pegando o maior divisor comum com o polinômio das derivadas com relação a cada variável no polinômio, todos os fatores 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 polinômios *p_1*, ..., *p_n*. Cada argumento *p_i* é um polinômio concoeficientes inteiros.

`tellrat (x)` efetivamente significa substituir 0 por *x* em funções racionais.

`tellrat ()` retorna uma lista das substituições correntes.

`algebraic` deve ser escolhida para `true` com o objetivo 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 pega kernels (núcleos) e remove propriedades `tellrat`.

Quando fazemos `tellrat` em um polinômio de várias variáveis, e.g., `tellrat (x^2 - y^2)`, pode existir uma ambigüidade como para ou substituir *y*`^2` por *x*`^2` ou vice-versa. Maxima seleciona uma ordenação particular, mas se o usuário 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 canônicas (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
- A base dos logarítmos naturais, e, é representada no Maxima como `%e`.

false Constante
- a contante Booleana, falso. (NIL em Lisp)

inf Constante
- infinito positivo real.

infinity Constante
- infinito complexo.

minf Constante
- menos infinito real.

%pi Constante
- "pi" é representado no Maxima como `%pi`.

true Constante
- a constante Booleana, verdadeiro. (T em Lisp)

14 Logarítmos

14.1 Definições para Logarítmos

%e_to_numlog Variável de opção

Valor padrão: `false`

Quando `true`, sendo `r` algum número racional, e `x` alguma expressão, `%e^(r*log(x))` irá ser simplificado em `x^r` . Note-se que o comando `radcan` também faz essa transformação, e transformações mais complicadas desse tipo também. O comando `logcontract` "contrai" expressões contendo `log`.

li [*s*] (*z*) Função

Representa a função polilogarítmo de ordem *s* e argumento *z*, definida por meio de séries infinitas

```
                    inf
                    ====   k
                    \     z
          Li (z) =   >    --
            s       /      s
                    ====  k
                    k = 1
```

`li [1]` é `- log (1 - z)`. `li [2]` e `li [3]` são as funções dilogarítmo e trilogarítmo, respectivamente.

Quando a ordem for 1, o polilogarítmo 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 polilogarítmo 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 logarítmo natural (base e) de x.

Maxima não possui uma função interna para logarítmo de base 10 ou de outras bases. `log10(x) := log(x) / log(10)` é uma definição útil.

Simplificação e avaliação de logarítmos são governadas por muitos sinalizadores globais:

`logexpand` - faz com que `log(a^b)` torne-se `b*log(a)`. Se `logexpand` for escolhida para `all`, `log(a*b)` irá também simplificar para `log(a)+log(b)`. Se `logexpand` for escolhida para `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 escolhida para `false`, todas essas simplificações irão ser desabilitadas.

`logsimp` - se `false` então nenhuma simplificação de `%e` para um expoente contendo `log`'s é concluída.

`lognumer` - se `true` então argumentos negativos em ponto flutuante para `log` irá sempre ser convertido para seu valor absoluto antes que `log` seja tomado. Se `numer` for também `true`, então argumentos negativos inteiros para `log` irão também ser convertidos para seu valor absoluto.

`lognegint` - se `true` implementa a regra `log(-n) -> log(n)+%i*%pi` para `n` um inteiro positivo.

`%e_to_numlog` - quando `true`, `r` sendo algum número racional, e `x` alguma expressão, `%e^(r*log(x))` irá ser simplificado em `x^r` . Note-se que o comando `radcan` também faz essa transformação, e transformações mais complicadas desse tipo também. O comando `logcontract` "contrai" expressões contendo `log`.

logabs Variável de opção

Valor padrão: `false`

Quando fazendo integração indefinida onde logs são gerados, e.g. `integrate(1/x,x)`, a resposta é dada em termos de `log(abs(...))` se `logabs` for `true`, mas em termos

de `log(...)` se `logabs` for `false`. Para integração definida, a escolha `logabs:true` é usada, porque aqui "avaliação" de integral indefinida nos extremos é muitas vezes necessária.

logarc Variável de opção

logarc (*expr*) Função

Se `true` irá fazer com que as funções circularee inversas e hiperbólicas sejam convertidas em formas logarítimicas. `logarc(`*exp*`)` irá fazer com que essa conversão para uma expressão particular *exp* sem escolher o comutador ou tendo que re-avaliar a expressão com `ev`.

Quando a variável global `logarc` for `true`, funções circulares inversas e funções hiperbólicas são substituídas por suas funções logarítmicas equivalentes. O valor padrão de `logarc` é `false`.

A função `logarc(`*expr*`)` realiza aquela substituição para uma expressão *expr* sem modificar o valor da variável global `logarc`.

logconcoeffp Variável de opção

Valor padrão: `false`

Controla quais coeficientes são contraídos quando usando `logcontract`. Pode ser escolhida para o nome de uma função predicado de um argumento. E.g. se você gosta de gerar raízes quadradas, você pode fazer `logconcoeffp:'logconfun$ logconfun(m):=featurep(m,integer) ou ratnump(m)$` . Então `logcontract(1/2*log(x));` irá fornecer `log(sqrt(x))`.

logcontract (*expr*) Função

Recursivamente examina 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 você faz `declare(n,integer);` então `logcontract(2*a*n*log(x));` fornece `a*log(x^(2*n))`. Os coeficientes que "contraem" dessa maneira são aqueles tais que 2 e `n` que satisfazem `featurep(coeff,integer)`. O usuário pode controlar quais coeficientes são contraídos escolhendo a opção `logconcoeffp` para o nome de uma função predicado de um argumento. E.g. se você gosta de gerara raízes quadradas, você pode fazer `logconcoeffp:'logconfun$ logconfun(m):=featurep(m,integer) ou ratnump(m)$` . então `logcontract(1/2*log(x));` irá fornecer `log(sqrt(x))`.

logexpand Variável de opção

Valor padrão: `true`

Faz com que `log(a^b)` torne-se `b*log(a)`. Se for escolhida para `all`, `log(a*b)` irá também simplificar para `log(a)+log(b)`. Se for escolhida para `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 for escolhida para `false`, todas essas simplificações irão ser desabilitadas.

lognegint Variável de opção

Valor padrão: `false`

Se `true` implementa a regra `log(-n) -> log(n)+%i*%pi` para `n` um inteiro positivo.

lognumer Variável de opção

Valor padrão: `false`

Se `true` então argumentos negativos em ponto flutuante para `log` irão sempre ser convertidos para seus valores absolutos antes que o `log` seja tomado. Se `numer` for também `true`, então argumentos inteiros negativos para `log` irão também ser convertidos para seus valores absolutos.

logsimp Variável de opção

Valor padrão: `true`

Se `false` então nenhuma simplificação de `%e` para um expoente contendo `log`'s é concluída.

plog (x) Função

Representa o principal ramo logarítmos naturais avaliados para complexos 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 usuário 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 coleção 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`, and `%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 usuário 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 cossenos 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 cossenos ocorrendo subseqüêntemente.

`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 padrão: `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 padrão: `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 padrão: `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 padrão: `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 canônica 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` lineariza 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), seção 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

16.2 specint

`hypgeo` é um pacote para manusear transformações de Laplace de funções especiais. `hyp` é um pacote para manusear funções Hipergeométricas generalizadas.

`specint` tenta calcular a integral definida (sobre o intervalo de zero a infinito) de uma expressão contendo funções especiais. Quando o integrando contém um fator `exp (-s t)`, o resultado é uma transformação de Laplace.

A sintaxe é como segue:

`specint (exp (-s*`*t*`) * `*expr*`, `*t*`);`

onde *t* é a variável de integração e *expr* é uma expressão contendo funções especiais.

Se `specint` não puder calcular a integral, o valor de retorno pode conter vários símbolos Lisp, incluindo `other-defint-to-follow-negtest`, `other-lt-exponential-to-follow`, `product-of-y-with-nofract-indices`, etc.; isso é um bug.

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)                      Polinômio de Hermite (Note bem: he, não h. Veja A&S
%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
gammagreek(a,z)                 Função gama incompleta
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
```

`demo ("hypgeo")` mostra muitos exemplos de transformações de Laplace calculadas através de `specint`.

Esse é um trabalho em andamento. Alguns nomes de funções podem mudar.

16.3 Definições para Funções Especiais

airy (x) Função

A função de Airy Ai. Se o argumento x for um número, o valor numérico de `airy (x)` é retornado. de outra forma, uma expressão não avaliada `airy (x)` é retornada.

A equação de Airy `diff (y(x), x, 2) - x y(x) = 0` tem duas soluções linearmente independentes, chamadas `ai` e `bi`. Essa equação é muito popular como uma aproximação para problemas mais complicados em muitos ambientes de física matemática.

`load ("airy")` chama as funções `ai`, `bi`, `dai`, e `dbi`.

O pacote `airy` contém rotinas para calcular `ai` e `bi` e suas derivadas `dai` e `dbi`. O resultado é um número em ponto flutuante se o argumento for um número, e uma expressão não avaliada de outra forma.

Um erro ocorre se o argumento for maior que o esperado causando um estouro nas exponenciais, ou uma perda de precisão no `sin` ou no `cos`. Isso faz o intervalo de validade sobre -2800 a 10^38 para `ai` e `dai`, e de -2800 a 25 para `bi` e `dbi`.

Essas regras de derivação são conhecidas para Maxima:

- `diff (ai(x), x)` retorna `dai(x)`,
- `diff (dai(x), x)` retorna `x ai(x)`,
- `diff (bi(x), x)` retorna `dbi(x)`,
- `diff (dbi(x), x)` retorna `x bi(x)`.

Valores de função são calculados a partir das séries de Taylor convergentes para `abs(`x`) < 3`, e a partir de expansões assintóticas para x `< -3` ou x `> 3` como necessário. Esses resultados somente apresentam discrepâncias numéricas muito pequenas em x `= 3` e x `= -3`. Para detalhes, veja Abramowitz e Stegun, *Handbook of Mathematical Functions*, Sessão 10.4 e Tabela 10.11.

`ev (taylor (ai(x), x, 0, 9), infeval)` retorna uma expansão de Taylor em ponto flutuante da função `ai`. Uma expressão similar pode ser construída para `bi`.

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 desatualizada. 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 padrão: `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, $scaled_bessel_i(v,z) = exp(-abs(z)) * bessel_i(v,z)$. 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 padrão: 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 fatoriais.

Veja também `makegamma`.

makegamma (*expr*) Função

Transforma instâncias de funções binomiais, fatorial, e beta em *expr* para funções gama.

Veja também `makefact`.

numfactor (*expr*) Função

Retorna o fator 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 canônico para adições e potências de termos de seno e cosseno 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 padrão: 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 cosseno 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 alé disso, livre de senos ou cossenos.

`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 usuário 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 `maxpsifracnum`. 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 padrão: 20

`maxpsiposint` é o maior valor positivo para o qual `psi[n](x)` irá tentar calcular um valor exato.

maxpsinegint Variável de opção

Valor padrão: -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 padrão: 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.

maxpsifracdenom Variável de opção

Valor padrão: 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], página 200 e [elliptic_kc], página 201.

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], página 200 and [elliptic_ec], página 201.

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], página 200.

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 *Gamma*. Use `makegamma` para avaliar esse valor.

elliptic_ec (*m*) Função

A integral elíptica completa de sgundo 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 funçõesv*Gamma*. Use `makegamma` para avaliar esse valor.

18 Limites

18.1 Definições para Limites

lhospitallim Variável de Opção

Valor padrão: 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 padrão: `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 dos 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 `atvalues` 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           !
                                          !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 vetor.) 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 padrão: `[]`

`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 indiretas 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 padrão: `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 padrão: `false`

Quando `derivsubst` for `true`, uma substituíção 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 objetos 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 padrão: `[]`

`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 manusear 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 numéricos, existe a função `romberg`; um integrador adaptativo que usa a regra da quadratura dos currais de Newton, chamada `quanc8`; e uma escolha 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` for details. Geralmente falando, Maxima somente manuseia integrais que são integráveis em termos de "funções elementares" (funções racionais, trigonometricas, logarítmicas, exponenciais, radicais, etc.) e umas poucas extensões (função de erro, dilogarithm). Isso não manuseia 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 todas as integrais que ocorrem 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. Todavia, isso deve obrigatóriamente ser realizado de forma que quando

uma alteração é feita em uma soma ou produto, essa mudança deve ser um artifício, i.e., `i = j+ ...`, não uma função de grau mais alto. E.g.,

```
(%i4) sum (a[i]*x^(i-2), i, 0, inf);
                           inf
                           ====
                           \          i - 2
(%o4)                       >     a  x
                           /       i
                           ====
                           i = 0
(%i5) changevar (%, i-2-n, n, i);
                          inf
                          ====
                          \                n
(%o5)                      >       a      x
                          /         n + 2
                          ====
                          n = - 2
```

dblint (*f, r, s, a, b*) Função

Uma rotina de integral dupla que foi escrita no alto-nível do Maxima e então traduzida e compilada para linguagem de máquina. Use `load (dblint)` para acessar esse pacote. Isso 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 isso primeiro calcula *r*`(x)` e *s*`(x)`; então o eixo Y entre *r*`(x)` e *s*`(x)` é subdividido e a integral ao longo do eixo Y é executada usando a regra de Simpson; então a integral ao longo do eixo X é concluída usando a regra de Simpson com os valores da função sendo as integrais-Y. Esse procedimento pode ser numericamente instável por uma grande variedade razões, mas razoávelmente rápido: evite usar isso sobre funções altamente oscilatórias e funções com singularidades (postes ou pontos de ramificação na região). As integrais Y dependem de quanto fragmentados *r*`(x)` e *s*`(x)` são, então se a ditância *s*`(x) -` *r*`(x)` varia rapidamente com X, nesse ponto pode ter erros substanciais provenientes de truncação com diferentes saltos-tamanhos nas várias integrais Y. Um pode incrementar `dblint_x` e `dblint_y` em uma tentativa para melhorar a convergência da reião, com sacrifício do tempo de computação. Os valores da função não são salvos, então se a função é muito desperdiçadora de tempo,você

terá de esperar por re-computação se você mudar qualquer coisa (desculpe). Isso é requerido 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 uma integral definida. `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 usuário, isso é suficiente para chamar `integrate`.

`defint` retorna uma expressão simbólica, e executa um dos dois: ou calcula a integral ou a forma substantiva da integral. Veja `quad_qag` e funções rellacionadas para aproximação numérica de integrais definidas.

erf (*x*) Função

Representa a função de erro, cuja derivada é: `2*exp(-x^2)/sqrt(%pi)`.

erfflag Variável de opção

Valor padrão: `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 polinômios cujo denominador tem somente fatores lineares e quadráticos. Usando a funções `laplace` e `ilt` juntas com as funções `solve` ou `linsolve` o usuário 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 simbolicamente calcular a integral de *expr* em relação a *x*. `integrate (`*expr*`,` *x*`)` é uma integral indefinida, enquanto `integrate (`*expr*`,` *x*`,` *a*`,` *b*`)` é uma integral definida, 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 definidas. Veja `residue` para computação de resíduos (integração complexa). Veja `antid` para uma forma alternativa de calcular integrais indefinidas.

A integral (uma expressão livre de `integrate`) é retornada se `integrate` obtém sucesso. De outra forma o valor de retorno é a forma substantiva da integral (o operador com apóstrofo `'integrate`) ou uma expressão contendo uma ou mais formas substantivas. A forma substantiva de `integrate` é mostrada com um sinal de integral.

Em algumas circunstâncias isso é útil para construir uma forma substantiva manualmente, colocando em `integrate` um apóstrofo, e.g., `'integrate (`*expr*`,` *x*`)`. Por exemplo, a 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` manuseia integrais definidas separadamente das indefinidas, e utiliza uma gama de heurísticas para manusear cada caso. Casos especiais de integrais definidas 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 relacionadas para as definições de funções `beta` e `psi`, e algumas integrais logarítmicas e trigonométricas. Processando funções racionais pode incluir computação de resíduo. Se um caso especial aplicável não é encontrado, tentativa será feita para calcular a integra indefinida e avaliar isso nos limites de integração.

Isso pode incluir pegar um limite como um limite de integração tendendo ao infinito ou a menos infinito; veja também `ldefint`.

Casos especiais de integrais indefinidas incluem funções trigonométricas, exponenciais e funções logarítmicas, e funções racionais. `integrate` pode também fazer uso de uma curta tabela de integais elementares.

`integrate` pode realizar uma mudança de variável se o integrando tem 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 acha uma integral indefinida, o algorítmo de Risch é executado. O sinalizador `risch` pode ser escolhido como um `evflag`, na chamada para `ev` ou na linha de comando, e.g., `ev (integrate (`*expr*`, `*x*`), risch)` ou `integrate (`*expr*`, `*x*`), risch`. Se `risch` está presente, `integrate` chama 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 usuário. 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 definidas e indefinidas 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 valor de retorno contém a forma substantiva `'integrate`. Nesse exemplo, Maxima pode extrair um fator do denominador de uma função racional, mas não pode fatorar o restante ou de outra forma achar sua integral. `grind` mostra a forma substantiva `'integrate` no resultado. Veja também `integrate_use_rootsof` para mais 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 uma 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 apóstrofo-apóstrofo `''` faz com que a integral seja avaliada, e o resultado 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 padrão: 0

`integração_constant_counter` é um contador que é atualizado a cada vez que uma constante de integração (nomeada pelo Maxima, e.g., `integrationconstant1`) é introduzida em uma expressão pela integração indefinida de uma equação.

integrate_use_rootsof Variável de opção

Valor padrão: `false`

Quando `integrate_use_rootsof` é `true` e o denominador de uma função racional não pode ser fatorado, `integrate` retorna a 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 uma integral não resolvida 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 da integral será expressa como um somatório 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(x  - x  + 1)
(%o4) ------------------------------------------------------
```

```
                            7

                                                          2 x + 1
                                        2             5 atan(-------)
                                   log(x  + x + 1)          sqrt(3)
                                 - --------------- + ---------------
                                         14              7 sqrt(3)
```

Alternativamente o usuário 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 polinômio cúbico. Algumas vezes isso ajudará Maxima a obter resultados mais úteis.

ldefint (*expr, x, a, b*) Função

Tenta calcular a integral definida de *expr* pelo uso de `limit` para avaliar a integral indefinida *expr* em relação a *x* no limite superior *b* e no limite inferior *a*. Se isso falha para calcular a integral definida, `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 a integral definida, 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.

qq Função

O pacote `qq` (que pode ser carregado com `load ("qq")`) contém uma função `quanc8` que pode pegar ou 3 ou 4 arguments. A versão de 3 argumentos calcula a integral da função especificada como primeiro argumento sobre o intervalo de lo a hi como em `quanc8 ('função, lo, hi)`. o nome da função pode receber apóstrofo. A versão de 4 argumentos calculará a integral da função ou expressão (primeiro argumento) em relação à variável (segundo argumento) no intervalo de `lo` a `hi` como em `quanc8(<f(x) or expressão in x>, x, lo, hi)`. O método usado é o da quadratura dos currais de Newton, e a rotina é adaptativa. Isso irá dessa forma gastar tempo dividindo o intervalo somente quando necessário para completar as condições de erro especificadas pelas variáveis `quanc8_relerr` (valor padrão=1.0e-4) e `quanc8_abserr` (valor padrão=1.0e-8) que dão o teste de erro relativo:

```
|integral(função) - valor calculado| < quanc8_relerr*|integral(função)|
```

e o teste de erro absoluto:

```
|integral(função) - valor calculado| < quanc8_abserr
```

`printfile ("qq.usg")` yields additional informação.

quanc8 (*expr, a, b*) Função

Um integrador adaptativo. Demonstração e arquivos de utilização são fornecidos. O método é para usar a regra da quadratura dos currais de Newton, daí o nome da função `quanc8`, disponível em versões de 3 ou 4 argumentos. Verificação de erro absoluto e erro relativo são usadas. Para usar isso faça `load ("qq")`. Veja também `qq`.

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 atualmente 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)
```

romberg (*expr, x, a, b*) Função

romberg (*expr, a, b*) Função

Integração de Romberg. Existem dois caminhos para usar essa função. O primeiro é um caminho ineficiente como a versão de integral definida de `integrate`: `romberg (<integrando>, <variável of integração>, <lower limit>, <upper limit>)`.

Exemplos:

```
(%i1) showtime: true$
(%i2) romberg (sin(y), y, 0, %pi);
Evaluação took 0.00 seconds (0.01 elapsed) using 25.293 KB.
(%o2)                  2.000000016288042
(%i3) 1/((x-1)^2+1/100) + 1/((x-2)^2+1/1000) + 1/((x-3)^2+1/200)$
(%i4) f(x) := ''%$
(%i5) rombergtol: 1e-6$
(%i6) rombergit: 15$
(%i7) romberg (f(x), x, -5, 5);
Evaluação took 11.97 seconds (12.21 elapsed) using 12.423 MB.
(%o7)                  173.6730736617464
```

O segundo é um caminho eficiente que é usado como segue:

```
romberg (<função name>, <lower limit>, <upper limit>);
```

Continuando o exemplo acima, temos:

```
(%i8) f(x) := (mode_declare ([função(f), x], float), ''(%th(5)))$
(%i9) translate(f);
(%o9)                          [f]
(%i10) romberg (f, -5, 5);
Evaluação took 3.51 seconds (3.86 elapsed) using 6.641 MB.
(%o10)                 173.6730736617464
```

O primeiro argumento deve ser uma função trauzida ou compilada. (Se for compilada isso deve ser declarado para retorno a `flonum`.) Se o primeiro argumento não for já traduzido, `romberg` não tentará traduzí-lo mas resultará um erro.

A precisão da integração é governada pelas variáveis globais `rombergtol` (valor padrão 1.E-4) e `rombergit` (valor padrão 11). `romberg` retornará um resultado se a diferença relativa em sucessivas aproximações for menor que `rombergtol`. Isso tentará dividir ao meio o tamanho do passo `rombergit` vezes antes que isso seja abandonado. O número de iterações e avaliações da função que `romberg` fará é governado por `rombergabs` e `rombergmin`.

`romberg` pode ser chamada recursivamente e dessa forma pode fazer integrais duplas e triplas.

Exemplo:

```
(%i1) assume (x > 0)$
(%i2) integrate (integrate (x*y/(x+y), y, 0, x/2), x, 1, 3)$
(%i3) radcan (%);
                    26 log(3) - 26 log(2) - 13
(%o3)             - --------------------------
                                3
(%i4) %,numer;
(%o4)                  .8193023963959073
(%i5) define_variable (x, 0.0, float, "Global variável in função F")$
(%i6) f(y) := (mode_declare (y, float), x*y/(x+y))$
(%i7) g(x) := romberg ('f, 0, x/2)$
(%i8) romberg (g, 1, 3);
(%o8)                  .8193022864324522
```

A vantagem com esse caminho é que a função `f` pode ser usada para outros propósitos, como imprimir gráficos. A desvantagem é que você tem que inventar um nome para ambas a função `f` e sua variável independente `x`. Ou, sem a variável global:

```
(%i1) g_1(x) := (mode_declare (x, float), romberg (x*y/(x+y), y, 0, x/2))$
(%i2) romberg (g_1, 1, 3);
(%o2)                    .8193022864324522
```

A vantagem aqui é que o código é menor.

```
(%i3) q (a, b) := romberg (romberg (x*y/(x+y), y, 0, x/2), x, a, b)$
(%i4) q (1, 3);
(%o4)                    .8193022864324522
```

Isso é sempre o caminho mais curto, e as variáveis não precisam ser declaradas porque elas estão no contexto de `romberg`. O uso de `romberg` para integrais multiplas pode ter grandes desvantagens, apesar disso. O amontoado de cálculos extras necessários por causa da informação geométrica descartada durante o processo pela expressão de integrais multiplas por esse caminho pode ser incrível. O usuário deverá ter certeza de entender e usar os comutadores `rombergtol` e `rombergit`.

rombergabs Variável de opção

Valor padrão: 0.0

Assumindo que estimativas sucessivas produzidas por `romberg` são `y[0]`, `y[1]`, `y[2]`, etc., então `romberg` retornará após `n` iterações se (grasseiramente falando)

```
(abs(y[n]-y[n-1]) <= rombergabs ou
abs(y[n]-y[n-1])/(if y[n]=0.0 then 1.0 else y[n]) <= rombergtol)
```

for `true`. (A condição sobre o número de iterações dadas por `rombergmin` deve também ser satisfeita.) Dessa forma se `rombergabs` é 0.0 (o padrão) você apenas pega o teste de erro relativo. A utilidade de uma variável adicional vem quando você executar uma integral, quando a contribuição dominante vem de uma pequena região. Então você pode fazer a integral sobre uma pequena região dominante primeiro, usando a verificação relativa de precisão, seguida pela integral sobre o restante da região usando a verificação absoluta de erro.

Exemplo: Suponha que você quer calcular

```
'integrate (exp(-x), x, 0, 50)
```

(numericamente) com uma precisão relativa de 1 parte em 10000000. Defina a função. `n` é o contador, então nós podemos ver quantas avaliações de função foram necessárias. Primeiro de tudo tente fazer a integral completa de uma só vez.

```
(%i1) f(x) := (mode_declare (n, integer, x, float), n:n+1, exp(-x))$
(%i2) translate(f)$
Warning-> n é an undefined global variable.
(%i3) block ([rombergtol: 1.e-6, romberabs: 0.0], n:0, romberg (f, 0, 50));
(%o3)                    1.000000000488271
(%i4) n;
(%o4)                           257
```

Que aproximadamente precisou de 257 avaliações de função . Agora faça a integral inteligentemente, primeiro fazendo `'integrate (exp(-x), x, 0, 10)` e então escolhendo `rombergabs` para 1.E-6 vezes (nessa integral parcial). Isso aproximdamente pega somente 130 avaliações de função.

```
(%i5) block ([rombergtol: 1.e-6, rombergabs:0.0, sum:0.0],
  n: 0, sum: romberg (f, 0, 10), rombergabs: sum*rombergtol, rombergtol:0.0,
       sum + romberg (f, 10, 50));
(%o5)                    1.000000001234793
(%i6) n;
(%o6)                             130
```

Então se `f(x)` onde a função pegou um longo tempo de computação, o segundo método fez a mesma tarefa 2 vezes mais rápido.

rombergit Variável de opção

Valor padrão: 11

A precisão do comando `romberg` de integração é governada pelas variáveis globais `rombergtol` e `rombergit`. `romberg` retornará um resultado se a diferença relativa em sucessivas approximações é menor que `rombergtol`. Isso tentará dividir ao meio o tamanho do passoe `rombergit` vezes antes disso ser abandonado.

rombergmin Variável de opção

Valor padrão: 0

`rombergmin` governa o número mínimo de avaliações de função que `romberg` fará. `romberg` avaliará seu primeiro argumento pelo menos `2^(rombergmin+2)+1` vezes. Isso é útil para integrar funções oscilatórias, onde o teste normal de convergência pode lgumas vezes inadequadamente passar.

rombergtol Variável de opção

Valor padrão: 1e-4

A precisão do comando de integração de `romberg` é governada pelas variáveis globais `rombergtol` e `rombergit`. `romberg` retornará um resultado se a difereça relativa em sucessivas aproximações é menor que `rombergtol`. Isso tentará dividir ao meio o tamanho do passo `rombergit` vezes antes disso ser abandonado.

tldefint (*expr*, *x*, *a*, *b*) Função

Equivalente a `ldefint` com `tlimswitch` escolhido para `true`.

20.3 Introdução a QUADPACK

QUADPACK é uma coleção de funções para aálculo numérico de integrais definidas 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

[1] Applied Mathematics and Programming Division, K.U. Leuven
[2] Applied Mathematics and Programming Division, K.U. Leuven
[3] Institut für Mathematik, T.U. Wien
[4] National Bureau of Standards, Washington, D.C., U.S.A

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 funçes não automáticas, mas elas não são expostas a nível de Maxima.

Informação adicionalsobre a bilioteca QUADPACK pode ser encontrada no livro do QUADPACK[7].

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 algorítmo 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 cosseno 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

[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.

$log(x-a)log(b-x)$, 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 a 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* seleciona 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 a 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 for encontrado;

`1` se muitos subintervalos foram concluídos;

`2` se erro excessivo é detectado;

`3` se ocorre comportamento extremamente ruim do integrando;

`6` se a entrada é inválida.

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 algorítmo de (Wynn, 1956).

`quad_qags` computes the 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 a 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` muitos subintervalos foram concluídos;

`2` erro excessivo é detectado;

`3` ocorreu comportamento excessivamente ruim do integrando;

`4` falhou para convergência

`5` integral é provavelmente divergente ou lentamente convergente

`6` se a entrada é inválida.

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 uma das 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 a 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` muitos subintervalos foram concluídos;

`2` erro excessivo é detectado;

`3` ocorreu comportamento excessivamente ruim do integrando;

`4` falhou para convergência;

`5` integral é provavelmente divergente ou lentamente convergente;

`6` se a entrada for inválida.

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 a 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` muitos subintervalos foram concluídos;

`2` erro excessivo é detectado;

`3` ocorreu comportamento excessivamente ruim do integrando;

`6` se a entrada é inválida.

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 cosseno 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 a integral

$$\int_a^\infty f(x)w(x)dx$$

A função peso w é selecionada 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 a 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` muitos subintervalos foram concluídos;

`2` erro excessivo é detectado;

`3` ocorreu um comportamento excessivamente ruim do integrando;

`6` se a entrada é invalida.

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 a integral usando a rotina Quadpack QAWO:

$$\int_a^b f(x)w(x)dx$$

A função peso w é selecionada 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 a 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` muitos subintervalos foram concluídos;

`2` erro excessivo é detectado;

`3` comportamento extremamente ruim do integrando;

`6` se a entrada é inválida.

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 a integral usando a rotina Quadpack QAWS:

$$\int_a^b f(x)w(x)dx$$

A função peso w é selecionada 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 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 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 a 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` muitos subintervalos foram concluídos;

`2` erro excessivo é detectado;

`3` ocorreu um comportamento excessivamente ruim do integrando;

`6` se a entrada é invalida.

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 padrão: `[]`

`%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 padrão: `false`

`algexact` afeta 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 exatas, apenas que `algsys` tentará primeiro pegar soluções exatas, 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 polinômios 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 fatoradas e quebradas em subsistemas.

(2) Para cada subsistema *S_i*, uma equação *E* e uma variável *x* são selecionados. 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 exata.

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 usuário 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 exatas 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 atualmente 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 polinômio *expr* ou equação polinômial *eqn* de uma variável.

O sinalizador `polyfactor` quando `true` faz com que `allroots` fatore o polinômio sobre os números reais se o polinômio for real, ou sobre os números complexos, se o polinômio for complexo.

`allroots` pode retornar resultados imprecisos no caso de multiplas raízes. Se o polinômio for real, `allroots (%i*`*p*`)`) pode retornar aproximações mais precisas que `allroots (`*p*`)`, como `allroots` invoca um algorítmo diferente naquele caso.

`allroots` rejeita não-polinômios. Isso requer que o numerador após a classificação (`rat`'ing) poderá ser um polinômio, e isso requer que o denominador seja quando muito um número complexo. Com um resultado disso `allroots` irá sempre retornar uma expressão equivalente (mas fatorada), se `polyfactor` for `true`.

Para polinômios complexos um algorítmo por Jenkins and Traub é usado (Algorithm 419, *Comm. ACM*, vol. 15, (1972), p. 97). Para polinômios reais o algorítmo 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 padrão: `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 padrão: `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 intermediárias (`%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 + ----- - -
                                  %t3     3
```

```
                                    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 padrão: `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, polinômio linear em (para esse caso) *g*`(`*t*`)` and *g*`(`*t*`+1)`

```
(%i1) eqn: (n + 1)*f(n) - (n + 3)*f(n + 1)/(n + 1) = (n - 1)/(n + 2);
                              (n + 3) f(n + 1)   n - 1
(%o1)         (n + 1) f(n) - ---------------- = -----
                                    n + 1         n + 2
(%i2) funcsolve (eqn, f(n));

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 padrão: `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 usuário 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 padrão: `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 exata.

De outra forma, isso é a palavra `approximate` ou `incomplete` correspondendo à forma inexata 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 usuário, 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,
(%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 polinômios 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` é `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.

```
(%i1) e1: x + z = y$
(%i2) e2: 2*a*x - y = 2*a^2$
(%i3) e3: y - 2*z = 2$
(%i4) linsolve ([e1, e2, e3], [x, y, z]);
(%o4)            [x = a + 1, y = 2 a, z = a - 1]
```

linsolvewarn Variável

Valor padrão: `true`

Quando `linsolvewarn` é `true`, `linsolve` imprime uma mensagem "Dependent equações eliminated".

linsolve_params Variável

Valor padrão: `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 padrão: `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 polinômio 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 sequüências 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 polinômio com coeficientes inteiros e n é um inteiro positivo retorna q, um polinômio 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 padrão: `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 intermediárias `%t1`, `t2`, etc., e atribui as soluções para eles.

realonly Variável

Valor padrão: `false`

Quando `realonly` é `true`, `algsys` retorna somente aquelas soluções que estão livres de `%i`.

realroots (*poly, bound*) Função

Acha todas as raízes reais de um polinômio também real de uma única variável poly dentro de uma tolerância de limite que, se menor que 1, faz com que todas as raízes da integral sejam achadas exatamente. O parâmetro limite pode ser arbitrariamente pequeno com o objetivo de encontrar qualquer precisão desejada. O primeiro argumento pode também ser uma equação. `realroots` escolhe `multiplicities`, útil em caso de multiplas raízes. `realroots (`p`)` é equivalente a `realroots (`p`, rootsepsilon)`. `rootsepsilon` é um número real usado para estabelecer um intervalo de confidência para as ra@'izes. Faça `example (realroots)` para um exemplo.

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 usuário, 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,
(%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 padrão: `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` afetando `rootscontract` como segue:

Problem	Value of rootsconmode	Result of applying rootscontract
x^(1/2)*y^(3/2)	false	(x*y^3)^(1/2)
x^(1/2)*y^(1/4)	false	x^(1/2)*y^(1/4)
x^(1/2)*y^(1/4)	true	(x*y^(1/2))^(1/2)
x^(1/2)*y^(1/3)	true	x^(1/2)*y^(1/3)
x^(1/2)*y^(1/4)	all	(x^2*y)^(1/4)
x^(1/2)*y^(1/3)	all	(x^3*y^2)^(1/6)

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 multiplo 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 padrão: 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 exceto 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 fatores então `solve` é chamada sobre cada um dos fatores. Finalmente `solve` usará as fórmulas quadráticas, cúbicas, ou quárticas onde necessário.

No caso onde E for um polinômio 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 afetam `solve`. `describe` pode então ser usada sobre o nome do comutador individual se seu proprósito 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 padrão: `true`

Quando `solvedecomposes` é `true`, `solve` chama `polydecomp` se perguntado para resolver polinômios.

solveexplicit Variável de opção

Valor padrão: `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 padrão: `true`

Quando `solvefactors` é `false`, `solve` não tenta fatorar a expressão. O `false` escolhido pode ser desejado em alguns casos onde a fatoração não é necessária.

solvenullwarn Variável de opção

Valor padrão: `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 padrão: `false`

Quando `solveradcan` é `true`, `solve` chama `radcan` que faz `solve` lento mas permitirá certamente que problemas contendo exponeniais e logarítmos sejam resolvidos.

solvetrigwarn Variável de opção

Valor padrão: `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 padrão: `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 Definições para Equações Diferenciais

bc2 (*solução, xval1, yval1, xval2, yval2*) Função

Resolve problema do valor limite para equações diferenciais de segunda ordem. Aqui: *solução* é uma solução geral para a equação, como encontrado por `ode2`, *xval1* é uma equação para a variável independente na forma *x* = *x0*, e *yval1* é uma equação para a variável dependente na forma *y* = *y0*. A *xval2* e a *yval2* são equações para essas variáveis em outro ponto. Veja `ode2` para exemplo de utilização.

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 *eqn*'s são equações diferenciais nas variáveis dependentes *x_1*, ..., *x_n*. A relação funcional deve ser explicitamente indicada em ambas as equações e as variáveis. Por Exemplo

```
'diff(f,x,2)=sin(x)+'diff(g,x);
'diff(f,x)+x^2-f=2*'diff(g,x,2);
```

não é o formato apropriado. O caminho correto é:

```
'diff(f(x),x,2)=sin(x)+'diff(g(x),x);
'diff(f(x),x)+x^2-f=2*'diff(g(x),x,2);
```

A chamada é então `desolve([%o3,%o4],[f(x),g(x)]);` .

Se as condições iniciais em 0 são conhecidas, elas podem ser fornecidas antes chamando `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 o problema do valor inicial para equação diferencial de primeira ordem. Aqui: *solução* é uma solução geral para a equação, como encontrado por `ode2`, *xval* é uma equação para a variável independente na forma *x* `=` *x0*, e *yval* é uma equação para a variável dependente na forma *y* `=` *y0*. Veja `ode2` para exemplo de utilização.

ic2 (*solução, xval, yval, dval*) Função

Resolve o problema do valor inicial para equação diferencial de segunda ordem. Aqui: *solução* é uma solução geral para a equação, como encontrado por `ode2`, *xval* é uma equação para a variável independente na forma *x* `=` *x0*, *yval* é uma equação para a variável dependente na forma *y* `=` *y0*, e *dval* é uma equação para a derivada da variável dependente com relação à variável independente avaliada no ponto *xval*. Veja `ode2` para exemplo de utilização.

ode2 (*eqn, dvar, ivar*) Função

A função `ode2` resolve equações diferenciais ordinária ou de primeira ou de segunda ordem. Recebe três argumentos: uma EDO *eqn*, a variável dependente *dvar*, e a variável independente*ivar*. Quando obtém sucesso, retorna ou uma solução (explícita ou implícita) para a variável dependente. `%c` é usado para representar a constante no caso de equações de primeira ordem, e `%k1` e `%k2` as constantes para equações de segunda ordem. Se `ode2` não pode obter a solução por alguma razão, retorna `false`, após talvez mostra uma mensagem de erro. O método implementado para equações diferenciais de primeira ordem na seqüência na qual eles são testados são: linear, separável, exato - talvez requerendo um fator de integração, homogêneos, equação de Bernoulli, e um método homogêneo geral. Para segunda ordem: coeficiente constante, exato, linear homogêneo com coeficientes não-constantes os quais podem ser transformados para coeficientes constates, o Euler ou equação equidimensional, o método de variação de parâmetros, e equações as quais são livres ou da variável independente ou da dependente de modo que elas possam ser reduzidas duas equações lineares de primeria ordem para serem resolvidas seqüêncialmente. No curso de resolver EDOs, muitas variáveis são escolhidas puramente para propósitos informativos: `método` denota o método de solução usado e.g. `linear`, `intfactor` denota qualquer fator de integração usado, `odeindex` denota o índice para o método de Bernoulli ou para o método homogêneo generalizado, e `yp` denota a solução particular para a técnica de variação de parâmetros.

Com o objetivo de resolver os problemas dos valores iniciais (PVIs) e problemas dos valores limite (PVLs), a rotina `ic1` está disponível para equações de primeira ordem, e `ic2` e `bc2` para segunda ordem PVIs e PVLs, respectively.

Example:

```
(%i1) x^2*'diff(y,x) + 3*y*x = sin(x)/x;
                      2 dy           sin(x)
(%o1)                 x  -- + 3 x y = ------
                         dx             x
(%i2) ode2(%,y,x);
                              %c - cos(x)
(%o2)                      y = -----------
                                     3
                                    x
(%i3) ic1(%o2,x=%pi,y=0);
                                cos(x) + 1
(%o3)                     y = - ----------
                                      3
                                     x
(%i4) 'diff(y,x,2) + y*'diff(y,x)^3 = 0;
                           2
                          d y      dy 3
(%o4)                     --- + y (--)  = 0
                            2      dx
                          dx
(%i5) ode2(%,y,x);
                      3
                     y  + 6 %k1 y
(%o5)                ------------ = x + %k2
                          6
(%i6) ratsimp(ic2(%o5,x=0,y=0,'diff(y,x)=2));
                             3
                            2 y  - 3 y
(%o6)                       - ---------- = x
                                  6
(%i7) bc2(%o5,x=0,y=1,x=1,y=3);
                           3
                          y  - 10 y       3
(%o7)                     --------- = x - -
                             6            2
```

23 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 cosseno. Após chamar `fft`, os coeficientes seno e cosseno, 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 padrão: 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 padrão: `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 canônica 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 padrão: 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 padrão: `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 padrão: 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.

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 `[-%pi, %pi]`.

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 padrão: `true`

Veja `foursimp`.

cosnpiflag Variável de opção

Valor padrão: `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 cosseno 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, %pi]`.

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 cosseno 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 Estatística

24.1 Definições para Estatística

gauss (*mean*, *sd*) Função

Retorna um número em ponto flutuante randômico de uma distribuição normal com usando *mean* e desvio padrão *sd*.

25 Arrays

25.1 Definições para Arrays

array (*name, dim_1, ..., dim_n*) Função
array (*name, type, dim_1, ..., dim_n*) Função
array ([*name_1, ..., name_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 ([`*name_1*`, ..., `*name_m*`], `*dim_1*`, ..., `*dim_n*`)` cria *m* arrays, todos da mesma dimensão.

Se o usuário 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 usuário 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, exceto 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`, exceto 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 (*name*, [*i_1*, ..., *i_n*]) Função

Retorna a expressão *name* `[`*i_1*`, ..., `*i_n*`]`.

Isso é um código remanescente de `funmake`, exceto o valor retornado é um array de referência não avaliado ao invés de uma chamada de função não avaliada.

arrays Variável de sistema

Valor padrão: `[]`

`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 `j`*number*. A quantidade *number* é determindad por referência a `gensumnum`, que pode ser alterada pelo usuário. Por exemplo, `gensumnum:0$` reseta isso.

fillarray (*A*, *B*) Função

Preenche o array *A* com *B*, que é uma lista ou um array.

Se *A* for um array de ponto flutuante (inteiro) então *B* poderá ser ou uma lista de números (inteiros) em ponto flutuante ou outro array em ponto flutuante (inteiro).

Se as dimensões do array forem diferentes *A* é preenchida na ordem da maior linha. Se não existem elementos livres em *B* o último elemento é usado para preencher todo o resto de *A*. Se existirem muitos os restantes serão descartados.

`fillarray` retorna esse primeiro argumento.

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 sequüência 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 objeto que ocupa espaço, mas depois de `y: false`, `y` não mais aponta para aquele objeto, então o objeto pode ser descartado.

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) +(%i6) arraymake (x, [i]);
(%o6) foo
 3
(%i7) ''%;
(%o7) +
```

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 objeto 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 objetos. 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`.

26 Matrizes e Álgebra Linear

/Matrices.texi/1.24/Sat Jun 24 06:45:55 2006/-ko/

26.1 Introdução a Matrizes e Álgebra Linear

26.1.1 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 distingüir `.` 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`.

26.1.2 Vetores

`vect` é um pacote de funções para análise vetorial. `load ("vect")` chama esse pacote, e `demo ("vect")` permite visualizar uma demonstração.

O pacote de análise vetorial 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 vetor 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.

26.1.3 auto

O pacote `eigen` contém muitas funções devotadas para a computação simbólica de autovalores e autovetores. 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 usuário entre sucessivas computações.

As funções no pacote `eigen` são `innerproduct`, `unitvector`, `columnvector`, `gramschmidt`, `eigenvalues`, `eigenvectors`, `uniteigenvectors`, e `similaritytransform`.

26.2 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 cofatores 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 polinômio 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 você quer 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 autovetores
(%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 padrão: `false`

Quando `detout` é `true`, o determinante de uma matriz cuja inversa é calculada é fatorado 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 corretamente.

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 padrão: `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 padrão: `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` afeta 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 padrão: `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 padrão: `false`

Quando `domxnctimes` é `true`, produtos não comutativos de matrizes são realizados.

dontfactor Variável de opção

Valor padrão: `[]`

`dontfactor` pode ser escolhido para uma lista de variáveis em relação a qual fatoração não é para ocorrer. (A lista é inicialmente vazia.) Fatoração também não pegará lugares com relação a quaisquer variáveis que são menos importantes, conforme a hierarquía de variável assumida para a forma expresão racional canônica (CRE), que essas na lista `dontfactor`.

doscmxops Variável de opção

Valor padrão: `false`

Quando `doscmxops` é `true`, operações escalar-matriz são realizadas.

doscmxplus Variável de opção

Valor padrão: `false`

Quando `doscmxplus` é `true`, operações escalar-matriz retornam uma matriz resultado. Esse comutador não é subsomado sob `doallmxops`.

dot0nscsimp Variável de opção

Valor padrão: `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 padrão: `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 padrão: `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 padrão: `true`

Quando `dotassoc` é `true`, uma expresão `(A.B).C` simplifica para `A.(B.C)`.

dotconstrules Variável de opção

Valor padrão: `true`

Quando `dotconstrules` é `true`, um produto não comutativo de uma constante e outro termo é simplificado para um produto comutativo. Ativando esse sinalizador efetivamente ativamos `dot0simp`, `dot0nscsimp`, e `dot1simp` também.

dotdistrib Variável de opção

Valor padrão: `false`

Quando `dotdistrib` é `true`, uma expresão `A.(B + C)` simplifica para `A.B + A.C`.

dotexptsimp Variável de opção

Valor padrão: `true`

Quando `dotexptsimp` é `true`, uma expresão `A.A` simplifica para `A^^2`.

dotident Variável de opção

Valor padrão: 1

`dotident` é o valor retornado por `X^^0`.

dotscrules Variável de opção

Valor padrão: `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 polinômio característico da matriz. Algumas vezes `solve` pode não estar habilitado a achar as raízes do polinômio; 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 autovetores da matriz correspondente para esses autovalores respectivamente. Os autovetores e os autovetores unitários da matriz são os autovetores direitos e os autovetores unitários direitos.

`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 afetam 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 autovetores 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 usuário 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 autovetores. 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 exceto 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, antisimétrica, 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 direto 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 selecionado `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 algorítmo 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 algorítmo é aplicado para as linhas de *x*. Se *x* é uma lista de listas, o algorítmo 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 alcançam o mesmo spaço que *x*. Se a dimensão do alcance 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 algorítmo para simplificar resultados intermediários. Como uma conseqüência, o valor de retorno pode conter inteiros fatorados.

`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 autovetores
(%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]
```

hach (*a, b, m, n, l*) Função

`hach` é um implementação algorítmo de programação linear de Hacijan.

`load ("kach")` chama essa função. `demo ("kach")` executa uma demonstração dessa função.

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 usuário calcular a inversa de uma matriz com entradas bfloat ou polinômios com coeficientes em ponto flutuante sem converter para a forma CRE.

Cofatores 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 fatorado 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 padrão: `[`

`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 padrão: `+`

`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 padrão: `*`

`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 padrão: `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 polinômio 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 polinômio característico. Para essas quantidade a função simétrica das raízes pode ser calculada, que nada mais são que os coeficientes do polinômio 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 algorítmo da árvore menor de Johnson-Gentleman. O argumento *n* é a ordem; isso é optional 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 padrão: `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 padrão: `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 vetor [indeterminação1, indeterminação2,...], e `dimension` é escolhida para o comprimento desse vetor. SF[1], SF[2], ..., SF[DIMENSION] são escohidos para fatores de escala de coordenada, e `sfprod` é escohido para o produto desse fatores 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 autovetores 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 objetivo de estar habilitado para a forma `rightmatrix`.

`load ("eigen")` chama essa função.

`simtran` é um sinônimo para `similaritytransform`.

sparse Variável de opção

Valor padrão: `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]`.

De outra forma *M* é uma lista, e o valor de retorno é uma matriz *N* de `length (m)` linhas e 1 coluna, tal que `N[i,1] = M[i]`.

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`, exceto 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 autovetores 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 autovetores 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 autovetores da matriz são conhecidos para o usuário 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 autovetores é dada a escolha do sinalizador `nondiagonalizable` pode não estar correta. Se esse é o caso por favor ecolha isso para o valor correto. O autor assume que o usuário sabe o que está fazendo e que não tentará diagonalizar uma matriz cujos autovetores não alcançam o mesmo espaço vetorial 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 vetor 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 padrão: `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
```

27 Funções Afins

27.1 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 dada como uma expressão geral, ela 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.

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*.

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 homogêneas, em ordem para o procedimento terminar.

Se você checou simplificações de envoltório 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.

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 nfluenciado pelos pesos das variáveis individuais.

declare_weight (*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`.

nc_degree (*p*) Função

Retorna o grau de um polinômio não comutativo *p*. Veja `declare_weights`.

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.

fast_central_elements ([*x_1*, ..., *x_n*], *n*) Função

Se `set_up_dot_simplifications` tiver sido feito previamente, ache o polinômio 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];
```

check_overlaps (*n*, *add_to_simps*) Função

Verifica as sobreposies através do grau *n*, tendo certeza que você tem regras de simplificaçõ suficiente em cada grau, para `dotsimp` trabalhar corretamente. Esse processo pode ter sua velocidade aumentada se você souber antes de começar souber de qual dimensão do espaço de monômios é. Se ele for de dimensão global finita, então `hilbert` pode ser usada. Se você não conhece as dimensões monomiais, não especifique um `rank_function`. Um opcional terceiro argumento `reset`, `false` diz para não se incomodar em perguntar sobre resetar coisas.

mono ([*x_1*, ..., *x_n*], *n*) Função

Retorna a lista de monômios independentes relativamente à simplificação atual do grau *n* nas variáveis *x_1*, ..., *x_n*.

monomial_dimensions (*n*) Função

Calcula a série de Hilbert através do grau *n* para a algebra corrente.

extract_linear_equations ([*p_1*, ..., *p_n*], [*m_1*, ..., *m_n*]) Função

Faz uma lista dos coeficientes dos polinômios 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 monômios.

list_nc_monomials ([*p_1*, ..., *p_n*]) Função

list_nc_monomials (*p*) Função

Retorna uma lista de monômios não comutativos que ocorrem em um polinômio *p* ou em uma lista de polinômios *p_1*, ..., *p_n*.

all_dotsimp_denoms Variável de opção

Valor padrão: `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 coletados por `dotsimp`.

28 itensor

28.1 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 objetos 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 exatamente 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 usuário. 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 direto. 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 intermediárias cujo crescimento fazem com que os programas terminem antes de serem completados. Através da experiência, um usuário 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 objetos 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 usuário possa encontrar dificuldade com a especificação da métrica, definição de função, e a avaliação de objetos "indexados" diferenciados.

28.1.1 Nova notação d tensores

Até agora, o pacote `itensor` no Maxima tinha usado uma notação que algumas vezes conduzia a ordenação incorreta 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 incorreto a menos que ocorra ser `a` um tensor simétrico. A razão para isso é que embora `itensor` mantenha corretamente 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 correto:

```
(%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!

28.1.2 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 "objeto indexado" . Um "objeto 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 objeto indexado, e os índices contravariantes através de uma lista como segundo argumento. Se o objeto 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 objeto 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 usuário 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 uma moldura métrica está sendo usada. Essa ordenação canônica 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 objeto indexado com relação a alguma coordenada cujos índices não aparecem como um argumento para o objeto indexado podem normalmente retornar zero. Isso é porque Maxima pode não saber que o tensor representado através do objeto indexado possívelmente depende implicitamente da respectiva coordenada. Pela modificação da função existente no Maxima, `diff`, em `itensor`, Maxima sabe assumir que todos os objetos 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 objetos. Atualmente, com relação às rotinas de simplificação, é assumido que objetos 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 objetos indexados completamente simétricos em suas listas de índices covariantes e simétricos em suas listas de índices contravariantes.

O pacote `itensor` geralmente trata tensores como objetos 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 molduras 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
```

28.2 Definições para itensor

28.2.1 Gerenciando objetos indexados

entertensor (*nome*) Função

É uma função que, através da linha de comando, permite criar um objeto 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 objetos 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 objetos 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 objetos 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 exatamente 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 exatamente duas vezes em cada termo alterado do conjunto `[%1, %2,...]`, se o segundo argumento opcional for omitido. De outra forma, os índices que ocorrem exatamente duas vezes são indexados começando no valor de *contador*. Cada índice que ocorre exatamente 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 padrão: `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 exatamente 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 objeto indexado tendo os índices apropriados (digamos `my_tensor`) irá retornar como resultado um objeto 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 objeto indexado; o último fornecido que for aplicado a uma contração particular irá ser usado. `contractions` é uma lista de objetos 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 objetos 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 objetos com os mesmos índices livres que *tensor*. Quando usada para atribuir valores a um tensor métrico no qual as componentes possuem índices que ocorrem exatamente duas vezes se deve ser cuidadoso para definir esses índices de forma a evitar a geração de índices que ocorrem exatamente duas vezes e que são multiplos. 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. Você 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 exatamente 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 padrão: `%`

É o prefixo para índices que ocorrem exatamente duas vezes (veja o exemplo sob índices `indices`).

icounter Variável de Opção

Valor padrão: `1`

Determina o sufixo numérico a ser usado na geração do próximo índice que ocorre exatamente 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 objeto 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 direta 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 exatamente 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 correto 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.

28.2.2 Simetrias de tensores

allsym Variável de Opção

Valor padrão: `false`. Se `true` então todos os objetos 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 objeto `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 exatamente 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 exatamente 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 correto.

28.2.3 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* irá ser 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 objeto indexado, isso equivale a anexar ao final os *v_i* como índices derivativos. Subseqüêntemente, 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 exatamente duas vezes `%i` é escolhido apropriadamente.

liediff (*v, ten*) Função

Calcula a derivada de Lie da expressão tensorial *ten* com relação ao campo vetorial *v*. *ten* pode ser qualquer expressão tensorial indexada; *v* pode ser o nome (sem índices) de um campo vetorial. 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 objetos indexados substituídas pela forma substantiva da função `idiff`. Seu argumento pode retornar aquele objeto indexado se a diferenciação for concluída. Isso é útil quando for desejado substituir um objeto 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 diretamente 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 atualmente, 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 objeto 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 vetor 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 objetos 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 objetos indexados.

makebox (*expr*) Função

Mostra *expr* da mesma maneira que `show`; todavia, qualquer tensor d'Alembertiano ocorrendo em *expr* irá ser 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
```

conseqüêntemente

```
 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 exatamente um índice derivativo.

28.2.4 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

$$\text{ichr2}([i,j],[k]) = g^{ks}\left(g_{is,j} + g_{js,i} - g_{ij,s}\right)/2$$

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:

$$\text{icurvature}^{h}_{ijk} = -\text{ichr2}^{h}_{ik,j} - \text{ichr2}^{h}_{\%1\,j}\,\text{ichr2}^{\%1}_{ik} + \text{ichr2}^{h}_{ij,k} + \text{ichr2}^{h}_{\%1}\,\text{ichr2}^{\%1}_{ij}$$

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 objetivo 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 objetos 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
```

28.2.5 Molduras móveis

Maxima atualmente tem a habilidade de executar cálculos usando molduras móveis. Essas podem ser molduras ortonormais (tetrads, vielbeins) ou uma moldura arbitrária.

Para usar molduras, você primeiro escolhe `iframe_flag` para `true`. Isso faz com que os símbolos de Christoffel, `ichr1` e `ichr2`, sejam substituídos pelas molduras mais gerais de coeficientes de conecção `icc1` e `icc2` em cálculos. Especialmente, o comportamento de `covdiff` e `icurvature` são alterados.

A moldura é definida através de dois tensores: o campo de moldura inversa (`ifri`), a base tetrad dual), e a métrica da moldura `ifg`. A métrica da moldura é a matriz identidade para molduras ortonormais, ou a métrica de Lorentz para molduras ortonormais no espaço-tempo de Minkowski. O campo de moldura inversa define a base da moldura (vetores unitários). Propriedades de contração são definidas para o campo de moldura e para a métrica da moldura.

Quando `iframe_flag` for `true`, muitas expressões `itensor` usam a métrica da moldura `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 moldura 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 moldura (`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 da moldura (`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 da moldura (`ifb`), essa função não faz nada.

ifb Variável

O suporte da moldura. A contribuição da métrica da moldura para os coeficientes de conecção é expressa usando o suporte da moldura:

```
                - ifb      + ifb      + ifb
                     c a b      b c a      a b c
ifc1    = ---------------------------------
    abc                  2
```

O suporte da moldura por si mesmo é definido em termos de campo de moldura e métrica da moldura. 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`:

```
          d      e                                        f
ifb =  ifr    ifr   (ifri      - ifri      - ifri    itr   )
   abc    b      c       a d,e       a e,d       a f    d e
```

Otherwise:

```
             e      d        d      e
ifb    = (ifr    ifr    - ifr    ifr   ) ifri
   abc       b      c,e      b,e    c        a d
```

icc1 Variável

Coeficientes de conecção de primeiro tipo. Em `itensor`, definido como

```
icc1    = ichr1    - ikt1    - inmc1
    abc        abc       abc       abc
```

Nessa expressão, se `iframe_flag` for `true`, o símbolo de Christoffel `ichr1` é substituído com o coeficiente de conecção da moldura `ifc1`. Se `itorsion_flag` for `false`, `ikt1` será omitido. `ikt1` é também omitido se uma base de moldura for usada, como a torsão está já calculada como parte do suporte da moldura. 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 moldura for usada, uma vez que a torsão já está calculada como parte do suporte da moldura. Ultimamente, como `inonmet_flag` é `false`, `inmc2` não estará presente.

ifc1 Variável

Coeficiente de moldura de primeiro tipo (também conhecido como coeficientes de rotação de Ricci). Esse tensor representa a contribuição da métrica da moldura 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 moldura de primeiro tipo. Esse tensor representa a contribuição da métrica da moldura para o coeficiente de conecção de primeiro tipo. Definido como uma permutação de suporte de moldura (`ifb`) com os índices apropriados incrementados e decrementados como necessário:

```
    c       cd
ifc2   = ifg   ifc1
    ab             abd
```

ifr Variável

O campo da moldura. Contrai (`ifri`) para e com a forma do campo inverso da moldura para formar a métrica da moldura (`ifg`).

ifri Variável

O campo inverso da moldura. Especifica a base da moldura (vetores base duais). Juntamente com a métrica da moldura, forma a base de todos os cálculos baseados em molduras.

ifg Variável

A métrica da moldura. O valor padrão é `kdelta`, mas pode ser mudada usando `components`.

ifgi Variável

O inverso da métrica da moldura. Contrai com a métrica da moldura (`ifg`) para `kdelta`.

iframe_bracket_form Variável de Opção

Valor padrão: `true`

Especifica como o suporte da moldura (`ifb`) é calculado.

28.2.6 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 vetor 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 vetor 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 uma moldura métrica for usada.)

inmc2 Variável

Permutação covariante de componentes do vetor 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 uma moldura métrica 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 uma moldura métrica 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 uma moldura métrica 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
```

28.2.7 Álgebra exterior

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 objetos, uma operação de multiplicação funciona como um produto exterior, 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 vetoriais, 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.

`~` Operator

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 `l` 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
```

`|` Operator

A barra vertical `|` denota a operação binária "contração com um vetor". Quando um tensor covariante totalmente antisimétrico é contraído com um vetor 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 vetor 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 incorretos.

extdiff (*expr, i*) Função

Calcula a derivada exterior de *expr* com relação ao índice *i*. A derivada exterior 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 padrão: `false`

Controla o comportamento de produto cunha e derivada exterior. 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.

28.2.8 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. Você pode tentar em seu lugar o comando `tentex`, o qual tenta traduzir expressões de tensores dentro de objetos TeX indexados apropriadamente.

tentex (*expr*) Função

Para usar a função `tentex`, você 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.

28.2.9 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 exatamente duas vezes são tornadas explícitas enquanto objetos indexados são transformados em arrays (os arrays subscritos estão na ordem de covariância seguidos de índices contravariantes dos objetos indexados). A derivada de um objeto indexado irá ser 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
```

28.2.10 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 objeto indexado
deri       Lista de índices de derivada de um objeto indexado
name       Retorna o nome de um objeto indexado
concan
irpmon
lc0
_lc2kdt0
_lcprod
_extlc
```

29 ctensor

29.1 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()`. Você é primeiramente solicitado a especificar 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 usuário é perguntado sobre isso.Cuidado deve ser tomado para evitar o conflito de nomes de coordenadas com outras definições de objetos.

No próximo passo, o usuário informa a métrica ou diretamente ou de um arquivo especificando sua posição ordinal. Como um exemplo de um arquivo de métrica comum, veja `share/tensor/metrics.mac`. A métrica está 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.

Um protocolo amostra é iniciado abaixo para a métrica estática, esfericamente simétrica (coordenadas padrão) que será aplicadas ao problema de derivação das equações de vácuo de Einstein (que levam à solução de Schwarzschild) como um exemplo. Muitas das funções em `ctensor` irão ser mostradas para a métrica padrão como exemplos.

```
(%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
```

29.2 Definições para ctensor

29.2.1 Inicialização e configuração

csetup () Função

É uma função no pacote `ctensor` (component tensor) que inicializa o pacote e permite ao usuário 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 usuário). 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 usuário 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 moldura inversa) e `lfg` (a métrica da moldura) sejam definidas. A partir dessas, a matriz da moldura `fr` e a métrica da moldura inversa `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 alte
```

`sistema_de_coordenadas` pode também ser uma lista de funções de transformação, seguida por uma lista contendo as varáveis coordenadas. Por exemplo, você 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`.

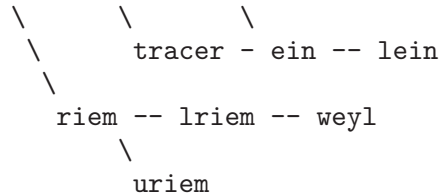
A função `init_ctensor` reinicializa o pacote `ctensor`. Essa função remove todos os arrays e matrizes usados por `ctensor`, coloca todos os sinalizadores de volta a seus valores padrão, retorna `dim` para 4, e retorna a métrica da moldura para a métrica da moldura de Lorentz.

29.2.2 Os tensores do espaço curvo

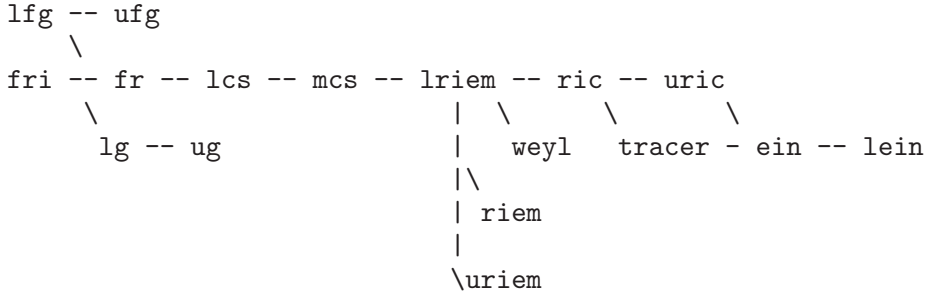
O principal propósito do pacote `ctensor` é calcular os tensores do espaç(tempo) curvo, mais notavelmente os tensores usados na relatividade geral.

Quando uma base métrica é usada, `ctensor` pode calcular os seguintes tensores:

```
lg  -- ug
  \      \
   lcs -- mcs -- ric -- uric
```



`ctensor` pode também usar molduras 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 diretamente. 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 fatoradas.

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:

```
                  l       _l        _l        _l   _m    _l   _m
R[i,j,k,l] =  R     = |       - |       + |     |   - |     |
                  ijk      ij,k      ik,j      mk   ij    mj   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 usuário 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 fatorada.

Se a variável `cframe_flag` for `false`, o tensor de Riemann é calculado diretamente dos símbolos de Christoffel. Se `cframe_flag` for `true`, o tensor covariante de Riemann é calculado primeiro dos coeficientes de campo da moldura.

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 diretamente dos coeficientes de campo da moldura. 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 objeto não é automaticamente simplificado devido ao fato 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 usuário. 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 fatorados também.

29.2.3 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.

29.2.4 Campos de moldura

Quando a variável `cframe_flag` for escolhida para `true`, o pacote `ctensor` executa seus cálculos usando uma moldura móvel.

frame_bracket (*fr*, *fri*, *diagframe*) Função

O delimitador da moldura (`fb[]`).

Calcula o delimitador da moldura conforme a seguinte definição:

```
   c          c          c         d     e
ifb   = ( ifri    - ifri    ) ifr   ifr
   ab         d,e        e,d       a     b
```

29.2.5 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 arquivo `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 uma moldura métrica ortonormal tetradimensional com assinatura métrica (-,+,+,+) está sendo usada. As componentes do tetrad nulo são relacionadas para a matriz moldura 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 moldura 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 moldura. Portanto, uma seqüência de cálculo expressiva pode começar com uma base de moldura, que é então usada para calcular `lg` (calculada automaticamente através de `cmetric`) e em seguida calcula `ug`. Nesse ponto, você 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 moldura em um estágio posterior pode retornar resultados inconsistentes, já que você pode terminar com um grande mistura de tensores, alguns calculados em uma base de moldura, alguns em uma base de coordenada, sem nenhum modo para distingüir 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 algorítmo 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.

29.2.6 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 usuário `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 vetor de não metricidade fornecido pelo usuário `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 objetivo 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 vetor de não metricidade *nm*.

29.2.7 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 usuário 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*. Atualmente, *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 usuário escolher o nome do vetor correspondente como o exemplo sob `contragrad` ilustra.

contragrad () Função

Calcula o gradiente contravariante de uma função escalar permitindo ao usuário escolher o nome do vetor 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 vetor (a divergência) onde n = `dim`. Se o argumento para a função for `g` então a divergência do tensor de Einstein irá ser formada e pode ser zero. Adicionalmente, a divergência (vetor) 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`.

29.2.8 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 usuário 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.

29.2.9 Variáveis usadas por `ctensor`

dim Variável de opção

Valor padrão: 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 padrão: `false`

Uma opção no pacote `ctensor`. Se `diagmetric` for `true` rotinas especiais calculam todos os objetos geométricos (que possuem o tensor métrico explicitamente) levando em consideração a diagonalidade da métrica. Tempo de execução 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. Atualmente, `ctrgsimp` afeta somente cálculos envolvendo uma moldura móvel.

cframe_flag Variável de opção

Faz com que cálculos sejam executados relativamente a uma moldura móvel em oposição a uma métrica holonômica. A moldura é definida através do array da moldura inversa `fri` e da métrica da moldura `lfg`. Para cálculos usando uma moldura Cartesiana, `lfg` pode ser a matriz unitária de dimensão apropriada; para cálculos em uma moldura 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 usuário.

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 vetor de não metricidade `nm` fornecido pelo usuário 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. atualmente, `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 padrão: `true`

Se `true` simplificação racional irá ser executada sobre as componentes não nulas de tensores de Einstein; se `ratfac` for `true` então as componentes irão também ser fatoradas.

ratriemann Variável de opção

Valor padrão: `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 fatorada.

ratweyl Variável de opção

Valor padrão: `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 fatoradas.

lfg Variável

A moldura métrica covariante. Por padrão, é inicializada para a moldura tetradimensional de Lorentz com assinatura (+,+,+,-). Usada quando `cframe_flag` for `true`.

ufg Variável

A métrica da moldura inversa. 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 da moldura, 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 usuário 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 usuário. 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 padrão: `[]`

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`.

29.2.10 Nomes reservados

Os seguintes nomes são usados internamente pelo pacote `ctensor` e não devem ser redefinidos:

```
Name         Description
--------------------------------------
_lg()        Avalia para lfg se a moldura métrica for usada,
                  para lg de outra forma
_ug()        Avalia para ufg se a moldura métrica for usada,
                  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 arquivo
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()    Moldura métrica, usado por cmetric() quando
                  cframe_flag:true
triemann()   Tensor de Riemann em base de moldura, usado quando
                  cframe_flag:true
tricci()     Tensor de Ricci em base de moldura, usada quando
                  cframe_flag:true
trrc()       Coeficientes de rotação de Ricci, usado por
                  christof()
yesp()
```

29.2.11 Changes

Em Novembro de 2004, o pacote `ctensor` foi extensivamente reescrito. Muitas funções e variáveis foram renomeadas com o objetivo 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
```

30 Pacote atensor

30.1 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 ativadas quando a função `init_atensor` é chamada.

A compatibilidade de `atensor` pode ser demonstrada pela definição da álgebra de quatérnios como uma álgera-Clifford Cl(0,2) com dois vetores fundamentais. As três unidades quaterniônicas imaginárias fundamentais são então os dois vetores 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 quatérnios, isso não foi usado nesse exemplo, no qual nós nos esforçamos para construir a tabela de multiplicação dos quatérnios 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 vetoriais símbolos indexados, onde o símbolo é aquele armazenado em `asymbol` e o índice 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 você deseja experimentar com uma álgebra não associativa, você pode também considerar a configuração de `dotassoc` para `false`. Nesse caso, todavia, `atensimp` não stará sempre habilitado a obter as simplificações desejadas.

30.2 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 quatérnios. A chamada `init_atensor(quaternion)` é equivalente a `init_atensor(clifford,0,0,2)`.

`pauli` implementa a álgebra de Pauli-spinors como a Clifford-álgebra Cl(3,0). Uma chamada a `init_atensor(pauli)` é equivalente a `init_atensor(clifford,3)`.

`dirac` implementa a álgebra de Dirac-spinors como a Clifford-álgebra 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` and `lie_envelop`.

adim Variável

A dimensionalidade da álgebra. `atensor` usa o valor de `adim` para determinar se um objeto indexado é uma base vetorial válida. Veja `abasep`.

aform Variável

Valor padrão para as formas bilineares `sf`, `af`, e `av`. O padrão é a matriz identidade `ident(3)`.

asymbol Variável

O símbolo para bases vetoriais.

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 vetoriais 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 vetoriais 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 vetoriais 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 vetorial `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`.

31 Séries

31.1 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 seção apresenta as variáveis globais que controlam a expansão.

31.2 Definições para Séries

cauchysum Variável de opção

Valor padrão: `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 polinômio 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 padrão: `true`

Quando `maxtayorder` for `true`, durante a manipulação algébrica de séries (truncadas) de Taylor, `taylor` tenta reter tantos termos quantos forem conhecidos serem corretos.

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 padrão: `[i, j, k, l, m, n]`

`niceindicespref` é a lista da qual `niceindices` pega 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, fatoriais, 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 objetivo 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 padrão: `false`

Quando `powerdisp` for `true`, uma adição é mostrada com seus termos em ordem do crescimento do expoente. Dessa forma um polinômio 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 padrão: `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 polinômio de grau igual ao maior expoente em *expr*. `revert2` retorna um polinômio 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*, *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 corretos.

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 padrão: 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 padrão: `true`

`taylor_logexpand` controla expansão de logarítmos em séries de `taylor`.

Quando `taylor_logexpand` for `true`, todos logarítmos 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 maematicamente correto 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 padrão: `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 padrão: `true`

Quando `taylor_truncate_polynomials` for `true`, polinômios são truncados baseados sobre a entrada de níveis de truncação.

De outra forma, entrada de polinômios para `taylor` são consideradas terem precisão infinita.

taytorat (*expr*) Função

Converte *expr* da forma `taylor` para a forma de expressão racional canônica (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 padrão: `false`

Quando `verbose` for `true`, `powerseries` mostra mensagens de progresso.

32 Teoria dos Números

32.1 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 polinômio 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 polinômio.

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, fatores 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 padrão: 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 fatorial. 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 em termos da constante `%phi`, que é `(1 + sqrt(5))/2`, aproximadamente 1.61803399.

Por padrão, Maxima não conhece `%phi`. Após executar `tellrat (%phi^2 - %phi - 1)` e `algebraic: true`, `ratsimp` pode simplificar algumas expressões contendo `%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) ratsimp (fibtophi (%));
(%o3)                          0
```

ifactors (*n*) Função

Para um inteiro positivo *n* retorna a fatoração de *n*. Se `n=p1^e1..pk^nk` for a decomposição de *n* em fatores primos, `ifactors` retorna `[[p1, e1], ... , [pk, ek]]`.

Os métodos de fatoração 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 fatoriais que diferem por um inteiro. `minfactorial` então converte um em um polinômio vezes o outro.

```
(%i1) n!/(n+2)!;
                                  n!
(%o1)                          --------
                               (n + 2)!
(%i2) minfactorial (%);
                                   1
(%o2)                      ---------------
                           (n + 1) (n + 2)
```

power_mod (*a, n, m*) Função

Usa umalgorítmo 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 usado para encontrar o inverso modular.

```
(%o1)                           2
(%i2) mod(3^15,5);
(%o2)                           2
(%i3) power_mod(2, -1, 5);
(%o3)                           3
(%i4) inv_mod(2,5);
(%o4)                           3
```

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 algorítmo utilizado é baseado no fato que os denominadores de uma expansão de fração parcial (os fatores 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)
```

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 341550071728321 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 padrão: 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 padrão: `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 padrão: `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)
```

33 Simetrias

33.1 Definições para Simetrias

comp2pui (*n*, *l*) Função

realiza a passagem das funções simétricas completas, dadas na lista *l*, às funções simétricas elementares de 0 a *n*. Se a lista *l* contém menos de *n*+1 elementos os valores formais vêm completá-los. O primeiro elemento da lista *l* fornece o cardinal do alfabeto se ele existir, se não existir coloca-se igual a *n*.

```
(%i1) comp2pui (3, [4, g]);
                            2                       2
(%o1)     [4, g, 2 h2 - g , 3 h3 - g h2 + g (g  - 2 h2)]
```

cont2part (*pc*, *lvar*) Função

Torna o polinômio 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]]
```

Outras funções de mudança de representação :

`contract`, `explose`, `part2cont`, `partpol`, `tcontract`, `tpartpol`.

contract (*psym*, *lvar*) Função

torna uma forma contraída (i.e. um monômio por órbita sobre a ação do grupo simétrico) do polinômio *psym* em variáveis contidas na lista *lvar*. A função `explose` realisa a operação inversa. A função `tcontract` testa adicionalmente a simetria do polinômio.

```
(%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
```

Outras funções de mudança de representação :

`cont2part`, `explose`, `part2cont`, `partpol`, `tcontract`, `tpartpol`.

direct ([*p_1*, ..., *p_n*], *y*, *f*, [*lvar_1*, ..., *lvar_n*]) Função

calcula a imágem direta (veja M. GIUSTI, D. LAZARD et A. VALIBOUZE, ISSAC 1988, Rome) associada à função *f*, nas listas de variáveis *lvar_1*, ..., *lvar_n*, e nos

polinômios *p_1*, ..., *p_n* de uma variável *y*. l'arite' da função *f* é importante para o cálculo. Assim, se a expressão de *f* não depende de uma variável, não somente é inútil fornecer essa variável como também diminui consideravelmente os cálculos se a variável não for fornecida.

```
(%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
```

Pesquisa de polinômios cujas raízes são a soma a+u ou a é a raíz de z^2 - e1* z + e2 e u é a raíz de z^2 - f1* z + f2

```
(%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` pode assumir dois sinalizadores: `elementaires` (elementares) e `puissances` (exponenciais - valor padrão) que permitem a decomposição de polinômios simétricos que aparecerem nesses cálculos pelas funções simétricas elementares ou pelas 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`).

ele2comp (*m, l*) Função

passa das funções simétricas elementares para funções completas. Semelhante a `comp2ele` e a `comp2pui`.

Outras funções de mudanças de base :

`comp2ele`, `comp2pui`, `ele2pui`, `elem`, `mon2schur`, `multi_elem`, `multi_pui`, `pui`, `pui2comp`, `pui2ele`, `puireduc`, `schur2comp`.

ele2polynome (*l, z*) Função

fornece o polinômio em *z* cujas funções simétricas elementares das raízes estiverem na lista *l*. *l* = [*n*, *e_1*, ..., *e_n*] onde *n* é o grau do polinômio 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
```

A recíproca: `polynome2ele` (*P*, *z*)

Veja também:

`polynome2ele`, `pui2polynome`.

ele2pui (*m, l*) Função

passa de funções simétricas elementares para funções completas. Similar a `comp2ele` e `comp2pui`.

Outras funções de mudanças de base :

`comp2ele`, `comp2pui`, `ele2comp`, `elem`, `mon2schur`, `multi_elem`, `multi_pui`, `pui`, `pui2comp`, `pui2ele`, `puireduc`, `schur2comp`.

elem (*ele, sym, lvar*) Função

decompõe o polinômio simétrico *sym*, nas variáveis contínuas da lista *lvar*, em funções simétricas elementares contidas na lista *ele*. Se o primeiro elemento de *ele* for fornecido esse será o cardinal do alfabeto se não for utilizado o grau do polinômio *sym*. Se falta valores para a lista *ele* valores formais do tipo "ei" são novamente colocados para completar a lista. O polinômio *sym* pode ser fornecido de 3 formas diferentes : contraída (`elem` deve protanto valer 1 que é seu valor padrão), particionada (`elem` deve valer 3) ou extendida (i.e. o polinômio por completo) (`elem` deve valer 2). A utilização da função `pui` se realiza sobre o mesmo modelo.

Sob um alfabeto de cardinal 3 com *e1*, a primeira função simétrica elementar, valendo 7, o polinômio simétrico em 3 variáveis cuja forma contraída (aqui, só depende de duas de suas variáveis) é x^4-2*x*y decompõe-se em funções simétricas elementares :

```
(%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 de mudanças de base :

`comp2ele`, `comp2pui`, `ele2comp`, `ele2pui`, `mon2schur`, `multi_elem`, `multi_pui`, `pui`, `pui2comp`, `pui2ele`, `puireduc`, `schur2comp`.

explose (*pc, lvar*) Função

toma o polinômio simétrico associado à forma contraída *pc*. A lista *lvar* contém variáveis.

```
(%i1) explose (a*x + 1, [x, y, z]);
(%o1)                  a z + a y + a x + 1
```

Outras funções de mudança de representação :

`contract`, `cont2part`, `part2cont`, `partpol`, `tcontract`, `tpartpol`.

kostka (*part_1, part_2*) Função

escrita por P. ESPERET, calcula o número de Kostka associado às partições *part_1* e *part_2*.

```
(%i1) kostka ([3, 3, 3], [2, 2, 2, 1, 1, 1]);
(%o1)                           6
```

lgtreillis (*n, m*) Função

torna a lista de partições de peso *n* e de largura *m*.

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

Veja também : `ltreillis`, `treillis` e `treinat`.

ltreillis (*n*, *m*) Função

torna a lista de partições de peso *n* e largura menor ou igual a *m*.

```
(%i1) ltreillis (4, 2);
(%o1)               [[4, 0], [3, 1], [2, 2]]
```

Veja também : `lgtreillis`, `treillis` e `treinat`.

mon2schur (*l*) Função

A lista *l* representa a função de Schur S_*l*: Temos *l* = [*i_1*, *i_2*, ..., *i_q*] com *i_1* <= *i_2* <= ... <= *i_q*. A função de Schur é S_[*i_1*, *i_2*, ..., *i_q*] é a menor da mariz infinita (h_{i-j}) *i* >= 1, *j* >= 1 composta das *q* primeiras linhas e de colunas *i_1* + 1, *i_2* + 2, ..., *i_q* + *q*.

Escreve-se essa função de Schur em função das formas monomiais utilizando as funções `treinat` e `kostka`. A forma retornada é um polinômio simétrico em uma de suas representações contraídas com as variáveis *x_1*, *x_2*, ...

```
(%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
```

queremos dizer que para 3 variáveis tem-se :

```
2 x1 x2 x3 + x1^2 x2 + x2^2 x1 + x1^2 x3 + x3^2 x1
 + x2^2 x3 + x3^2 x2
```

Outras funções de mudanças de base :

`comp2ele`, `comp2pui`, `ele2comp`, `ele2pui`, `elem`, `multi_elem`, `multi_pui`, `pui`, `pui2comp`, `pui2ele`, `puireduc`, `schur2comp`.

multi_elem (*l_elem*, *multi_pc*, *l_var*) Função

decompõe um polinômio multi-simétrico sob a forma multi-contraída *multi_pc* nos grupos de variáveis contidas na lista de listas *l_var* sobre os groupos de funções simétricas elementares 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 de mudanças de base :

`comp2ele`, `comp2pui`, `ele2comp`, `ele2pui`, `elem`, `mon2schur`, `multi_pui`, `pui`, `pui2comp`, `pui2ele`, `puireduc`, `schur2comp`.

multi_orbit (*P*, [*lvar_1*, *lvar_2*, ..., *lvar_p*]) Função

P é um polinômio no conjunto das variáveis contidas nas listas *lvar_1*, *lvar_2*, ..., *lvar_p*. Essa função leva novamente na órbita do polinômio *P* sob a ação do do produto dos grupos simétricos dos conjuntos de variáveis representados por essas *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` pela ação de um só grupo simétrico.

multi_pui Função

está para a função `pui` da mesma forma que a função `multi_elem` está 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
```

multinomial (*r*, *part*) Função

onde *r* é o peso da partição *part*. Essa função reporta ao coeficiente multinomial associado : se as partes das partições *part* forem *i_1*, *i_2*, ..., *i_k*, o resultado de `multinomial` é *r*`!/(`*i_1*`!` *i_2*`!` `...` *i_k*`!)`.

multsym (*ppart_1*, *ppart_2*, *n*) Função

realiza o produto de dois polinômios simétricos de *n* variáveis só trabalhando o módulo da ação do grupo simétrico de ordem *n*. Os polinômios estão em sua representação particionada.

Sejam os 2 polinômios simétricos em `x`, `y`: `3*(x + y) + 2*x*y` e `5*(x^2 + y^2)` cujas formas particionada são respectivamente `[[3, 1], [2, 1, 1]]` e `[[5, 2]]`, então seu produto será dado por :

```
(%i1) multsym ([[3, 1], [2, 1, 1]], [[5, 2]], 2);
(%o1)         [[10, 3, 1], [15, 3, 0], [15, 2, 1]]
```

seja `10*(x^3*y + y^3*x) + 15*(x^2*y + y^2*x) + 15*(x^3 + y^3)`.

Funções de mudança de representação de um polinômio simétrico :

`contract`, `cont2part`, `explose`, `part2cont`, `partpol`, `tcontract`, `tpartpol`.

orbit (*P*, *lvar*) Função

calcula a órbita de um polinômio *P* nas variáveis da lista *lvar* soba a ação do grupo simétrico do conjunto das variáveis contidas 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 polinômio.

part2cont (*ppart, lvar*) Função

passa da form particionada à forma contraída d um polinômio simétrico. A forma contraída é conseguida com as variáveis contidas em *lvar*.

```
(%i1) part2cont ([[2*a^3*b, 4, 1]], [x, y]);
                                3    4
(%o1)                      2 a  b x  y
```

Outras funções de mudança de representação :

`contract`, `cont2part`, `explose`, `partpol`, `tcontract`, `tpartpol`.

partpol (*psym, lvar*) Função

psym é um polinômio simétrico nas variáveis de *lvar*. Esta função retoma sua representação particionada.

```
(%i1) partpol (-a*(x + y) + 3*x*y, [x, y]);
(%o1)               [[3, 1, 1], [- a, 1, 0]]
```

Outras funções de mudança de representação :

`contract`, `cont2part`, `explose`, `part2cont`, `tcontract`, `tpartpol`.

permut (*l*) Função

retoma a lista de permutações da lista *l*.

polynome2ele (*P, x*) Função

fornece a lista *l* = [*n*, *e_1*, ..., *e_n*] onde *n* é o grau do polinômio *P* na variável *x* e *e_i* é a *i*-ézima 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 recíproca : `ele2polynome (`*l*`,` *x*`)`

prodrac (*l, k*) Função

l é uma lista que contém as funções simétricas elementares sob um conjunto *A*. `prodrac` produz o polinômio cujas raízes são os produtos *k* a *k* dos elementos de *A*.

pui (*l, sym, lvar*) Função

decompõe o polinômio simétrico *sym*, nas variáveis contidas a lista *lvar*, nas funções exponenciais contidas na lista *l*. Se o primeiro elemento de *l* for dado ele será o cardinal do alfabeto se não for dado toma-se o grau do polinômio *sym* para ser o cardinal do alfabeto. Se faltarem valores na lista *l*, valores formais do typo "pi" serão colocados na lista. O polinômio `sym` pode ser dado sob 3 formas diferentes : contraída (`pui` deve valer 1 - seu valor padrão), particionada (`pui` deve valer 3) ou estendida (i.e. o polinômio por completo) (`pui` deve valer 2). A função `elem` se utiliza da mesma maneira.

```
(%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 de mudanças de base :

`comp2ele`, `comp2pui`, `ele2comp`, `ele2pui`, `elem`, `mon2schur`, `multi_elem`, `multi_pui`, `pui2comp`, `pui2ele`, `puireduc`, `schur2comp`.

pui2comp (*n*, *lpui*) Função

produz a lista das *n* primeiras funções completas (com o cardinal em primeiro lugar) em função das funções exponenciais dadas na lista *lpui*. Se a lista *lpui* estiver vazia o cardianl será N, se não estiver vazia, será o primeiro elemento de forma análoga a `comp2ele` e a `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 de mudanças de base :

`comp2ele`, `comp2pui`, `ele2comp`, `ele2pui`, `elem`, `mon2schur`, `multi_elem`, `multi_pui`, `pui`, `pui2ele`, `puireduc`, `schur2comp`.

pui2ele (*n*, *lpui*) Função

realiza a transformação das funções exponenciais em funções simétricos elementares. Se o sinalizador `pui2ele` for `girard`, recupera-se a lista de funções simétricos elementares de 1 a *n*, e se for igual a `close`, recupera-se a *n*-ézima função simétrica elementar.

Outras funções de mudanças de base :

`comp2ele`, `comp2pui`, `ele2comp`, `ele2pui`, `elem`, `mon2schur`, `multi_elem`, `multi_pui`, `pui`, `pui2comp`, `puireduc`, `schur2comp`.

pui2polynome (*x*, *lpui*) Função

calcula o polinômio 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
```

Autres funções a‘ voir : `polynome2ele`, `ele2polynome`.

pui_direct (*orbite*, [*lvar_1*, ..., *lvar_n*], [*d_1*, *d_2*, ..., *d_n*]) Função

Seja *f* um polinômio em *n* blocos de variáveis *lvar_1*, ..., *lvar_n*. Seja *c_i* o número de variáveis em *lvar_i* . E *SC* o produto dos *n* grupos simétricos de grau *c_1*, ..., *c_n*. Esse grupo age naturalmente sobre *f*. A Lista *orbite* é a órbita, anotada de *SC*(*f*), da função *f* sob a ação de *SC*. (Essa lista pode ser obtida com a função : `multi_orbit`). Os `d_i` são inteiros tais que *c_1* <= *d_1*, *c_2* <= *d_2*, ..., *c_n* <= *d_n*. Seja *SD* o produto dos grupos simétricos *S_d1* x *S_d2* x ... x *S_dn*.

A função `pui_direct` retorna as *n* premeiras funções exponenciais de *SD*(*f*) dedzidas das funções exponenciais de *SC*(*f*) onde *n* é o cardinal de *SD*(*f*).

O resultado é produzido sob a forma multi-contraída em relação a *SD*. i.e. apenas se conserva um elemento 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 ]
```

puireduc (*n*, *lpui*) Função

lpui é uma lista cujo primeiro elemento é um inteiro *m*. `puireduc` fornece as *n* primeiras funções exponenciais em função das *m* primeira.

```
(%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
```

resolvante (*P*, *x*, *f*, [*x_1*, ..., *x_d*]) Função

calcula a resolvente do polinômio *P* em relação à variável *x* e de grau *n* >= *d* pela função *f* expressa nas variáveis *x_1*, ..., *x_d*. É importante para a eficácia dos cálculos não colocar na lista [*x_1*, ..., *x_d*] as variáveis não interferindo na função de transformação *f*.

Afim de tornar mais eficazes os cálculos pode-se colocar sinalizadores na variável `resolvante` para que os algorítmos adequados sejam utilizados :

Se a função *f* for unitária :

- um polinômio de uma variável,
- linear ,
- alternado,
- uma soma de variáveis,
- simétrico nas variáveis que aparecem em sua expressão,
- um produto de variáveis,
- a função da resolvente de Cayley (utilisável no grau 5)

```
(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
```

Pela resolvente de Cayley, os 2 últimos arguments são neutros e o polinômio 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 $\prod_{1\leq i<j\leq n-1} (x_i-x_j)$.

Veja também :

resolvante_produit_sym, resolvante_unitaire, resolvante , resolvante_klein, resolvante_klein3, resolvante_vierer, resolvante_diedrale, resolvante_bipartite.

resolvante_bipartite (*P*, *x*) Função

calcule la transformation de *P*(*x*) de degre *n* (*n* pair) par la função $x_1x_2\ldots x_{n/2}+x_{n/2+1}\ldotsx_n$

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

calcule la transformation de *P*(*x*) par la função *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

calcule la transformation de *P*(*x*) par la função *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

calcule la transformation de *P*(*x*) par la 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 produto do polinômio *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

calcul a resolvente do polinômio *P*(*x*) pelo polinômio *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`.

schur2comp (*P*, *l_var*) Função

P é um polinômio nas variáveis contidas na lista *l_var*. Cada uma das variáveis de *l_var* representa uma função simétrica completa. Representa-se em *l_var* a enésia função simétrica completa como a concatenação da letra `h` com o inteiro *i* : h*i*. Essa função fornece a expressão de *P* em função das funções de Schur.

```
(%i1) schur2comp (h1*h2 - h3, [h1, h2, h3]);
(%o1)                         s
                               1, 2
(%i2) schur2comp (a*h3, [h3]);
(%o2)                         s  a
                               3
```

somrac (*l, k*) Função

a lista *l* contém as funções simétricas elementares de um polinômio P . Calcula-se o polinômo cujas reízes são as somas K a K distintos das raízes de P.

Veja também `prodrac`.

tcontract (*pol, lvar*) Função

teste si le polinômio pol est simétrico en les variáveis contenues dans la liste *lvar*. Si oui il rend une forme contracte'e comme la função `contract`.

Outras funções de mudança de representação :

`contract`, `cont2part`, `explose`, `part2cont`, `partpol`, `tpartpol`.

tpartpol (*pol, lvar*) Função

testa se o polinômio *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`.

34 Grupos

34.1 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`.

As tabelas de multiplicação para as variáveis estão em `table:todd_coxeter_state[2]`. Então `table[i]` fornece a tabela para a i'ésima variável. `multiplos_co_conjuntos(co_conjunto,i) := table[varnum][co_conjunto];`

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
```

```
(%i7) table:todd_coxeter_state[2]$
(%i8) table[1];
(%o8) {Array: (SIGNED-BYTE 30) #(0 2 1 3 7 6 5 4 8 11 17 9 12 14 #

13 20 16 10 18 19 15 0 0 0 0 0 0 0 0 0 0 0 0

  0 0 0)}
```

Observe que somente os elementos de 1 a 20 desse array `%o8` são significativos. `table[1][4] = 7` indica coset4.var1 = coset7

35 Ambiente em Tempo de Execução

35.1 Introdução a Ambiente em Tempo de Execução

`maxima-init.mac` é um arquivo que é chamado automaticamente quando o Maxima inicia. Você pode usar `maxima-init.mac` para personalizar seu ambiente Maxima. `maxima-init.mac`, se existir, é tipicamente colocado no diretório chamado por `maxima_userdir`, embora possa estar em qualquer outro diretório procurado pela função `file_search`.

Aqui está um exemplo do arquivo `maxima-init.mac`:

```
setup_autoload ("specfun.mac", ultraspherical, assoc_legendre_p);
showtime:all;
```

Nesse Exemplo, `setup_autoload` diz ao Maxima para chamar o arquivo especificado (`specfun.mac`) se qualquer das funções (`ultraspherical`, `assoc_legendre_p`) forem chamadas sem estarem definidas. Dessa forma você não precisa lembrar de chamar o arquivo antes das funções.

A declaração `showtime: all` diz ao Maxima escolher a variável `showtime`. O arquivo `maxima-init.mac` pode conter qualquer outras atribuições ou outras declarações do Maxima.

35.2 Interrupções

O usuário 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 usuário. 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 usuário 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.

35.3 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 objeto *a* tem o recurso *f* na base dos fatos 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 diretório no qual Maxima cria alguns arquivos temporários. Em particular, arquivos temporários para impressão são criados no `maxima_tempdir`.

O valor inicial de `maxima_tempdir` é o diretório do usuário, se o maxima puder localizá-lo; de outra forma Maxima supõe um diretório adequado.

A `maxima_tempdir` pode ser atribuído uma seqüência de caracteres que corresponde a um diretório.

maxima_userdir Variável de sistema

`maxima_userdir` nomeia um diretório no qual Maxima espera encontrar seus próprios arquivos e os do arquivos do Lisp. (Maxima procura em alguns outros diretórios também; `file_search_maxima` e `file_search_lisp` possuem a lista completa.)

O valor inicial de `maxima_userdir` é um subdiretório do diretório do usuário, se Maxima puder localizá-lo; de outra forma Maxima supõe um diretório adequado.

A `maxima_userdir` pode ser atribuído uma seqüência de caracteres que corresponde a um diretório. 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 seqüência 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 seqüência de caracteres representando a data e hora atuais. A seqüência 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 seqüência de caracteres Lisp.

Exemplo:

```
(%i1) d: timedate ();
(%o1) 08:05:09 Wed, 11/02/2005 (GMT-7)
(%i2) print ("timedate mostra o tempo atual", d)$
timedate reports current time 08:05:09 Wed, 11/02/2005 (GMT-7)
```

36 Opções Diversas

36.1 Introdução a Opções Diversas

Nessa seção várias opções são tratadas pelo fato de possuirem um efeito global sobre a operação do Maxima. Também várias listas tais como a lista de todas as funções definidas pelo usuário, são discutidas.

36.2 Compartilhado

O diretório "share" do Maxima contém programas e outros arquivos de interesse para os usuários 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 diretório compartilhado dentro do sistema de arquivos do usuário.

`printfile ("share.usg")` imprime uma lista de pacotes desatualizados dos pacotes compartilhados. Usuários podem encontrar isso de forma mais detalhada navegando no diretório compartilhado usando um navegador de sistema de arquivo.

36.3 Definições para Opções Diversas

aliases Variável de sistema

Valor padrão: `[]`

`aliases` é a lista de átomos que possuem um alias definido pelo usuário (escolhido através das funções `alias`, `ordergreat`, `orderless` ou através da declaração do átomo como sendo um `noun` (substantivo) com `declare`).

alphabetic Declaração

`declare (`*char*`, alphabetic)` adiciona `char` (caracteres) ao alfabeto do Maxima, que inicialmente contém as letras de `A` até `Z`, de `a` até `z`, `%` e `_`. *char* é especificado como uma seqüência de caracteres de comprimento 1, e.g., `"~"`.

```
(%i1) declare ("~", alphabetic);
(%o1)                          done
(%i2) foo~bar;
(%o2)                         foo~bar
(%i3) atom (foo~bar);
(%o3)                          true
```

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 você pode somente lembra parte do nome de alguma coisa você pode usar esse comando para achar o restante do nome. Similarmente, você 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 padrão: `i`

`genindex` é o prefixo usado para gerar a próxima variável do somatório quando necessário.

gensumnum Variável de opção

Valor padrão: 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 padrão: `[]`

`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 usuário, não opções do Maxima ou comutadores, criados através de `:` ou `::` ou associando funcionalmente.

`functions` Todas as funções definidas pelo usuário, criadas através de `:=` ou `define`.

`arrays` Todos os arrays declarados e não declarados, criados através de `:`, `::`, ou `:=`.

`macros` Todas as macros definidas pelo usuário.

`myoptions` Todas as opções alguma vez alteradas pelo usuário (mesmo que tenham ou não elas tenham mais tarde retornadas para seus valores padrão).

`rules` Todos os modelos definidos pelo usuário que coincidirem e regras de simplificação, criadas através de `tellsimp`, `tellsimpafter`, `defmatch`, ou `defrule`.

`aliases` Todos os átomos que possuem um alias definido pelo usuário, 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 usuário, 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 usuário 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 usuário.)

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 padrão: `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`, exceto 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 usuário.

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 seqüência 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 usuário *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 usuário (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 fatores numéricos somente. Isso é ligeiramente diferente do comportamento de de `combine`, que coleta 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 fatores numérios do denominador. Esses resultados em forma ideal, e a possibilidade de reconhecer alguns cancelamentos.

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 (*nomedearquivo, 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`. *nomedearquivo* 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
```

37 Regras e Modelos

37.1 Introdução a Regras e Modelos

Essa seção descreve coincidências de modelos definidos pelo usuário 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 polinômios através de `tellrat`, e para álgebra comutativa e não comutativa no pacote `affine`.

37.2 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 padrão: `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 padrão: `default_let_rule_package`

`default_let_rule_package` é o nome do pacote de regras usado quando um não for explicitamente escolhido pelo usuário com `let` ou através de alteração do valor de `current_let_rule_package`.

defmatch (*prognome, modelo, x_1, ..., x_n*) Função

Cria uma função *prognome* (*expr*, *y_1*, ..., *y_n*) que testa *expr* para ver se essa expressão coincide com *modelo*.

modelo é uma expresão contendo as variáveis de modelo *x_1*, ..., *x_n* e parâmetros de modelo, se quaisquer. As variáveis de modelo são dadas explicitamente como argumentos para `defmatch` enquanto os parâmetros de modelo são declarados através da função `matchdeclare`.

O primeiro argumento para a função criada *prognome* é uma expressão a ser comparada contra o modelo e os outros argumentos são as variáveis atuais *y_1*, ..., *y_n* ne expressão que corresponde às variáveis correspondentes *x_1*, ..., *x_n* no modelo.

Se a tentativa de coincidência obtiver sucesso, *progname* retorna uma lista de equações cujos lados esquerdos são as variáveis de modelo e os parâmetros de modelo, e cujos lados direitos são expressões cujas variáveis de modelo e modelos coincidirão. Os parâmetros de modelo, mas não as variáveis de modelo, são atribuídos às subexpressões que elas coincidem. Se a coincidência falhar, *prognome* retorna `false`.

Quaisquer variáveis não declaradas como parâmetros de modelo em `matchdeclare` ou como variáveis em `defmatch` coincidem somente consigo mesmas.

Um modelo que não contiver nenhuma variável de modelo ou parâmetros retorna `true` se a coincidência ocorre.

Veja também `matchdeclare`, `defrule`, `tellsimp`, e `tellsimpafter`.

Exemplos:

Esse `defmatch` define a função `linearp (expr, y)`, que testa `expr` para ver se essa expressão é da forma `a*y + b` tal que `a` e `b` não contenham `y`.

```
(%i1) matchdeclare (a, freeof(x), b, freeof(x))$
(%i2) defmatch (linearp, a*x + b, x)$
(%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
```

Se o terceiro argumento para `defmatch` na linha (%i2) tiver sido omitido, então `linear` pode somente coincidir com expressões lineares em x, não em qualquer outra variável.

```
(%i1) matchdeclare ([a, f], true)$
(%i2) constinterval (l, h) := constantp (h - l)$
(%i3) matchdeclare (b, constinterval (a))$
(%i4) matchdeclare (x, atom)$
(%i5) (remove (integrate, outative),
          defmatch (checklimits, 'integrate (f, x, a, b)),
          declare (integrate, outative))$
(%i6) 'integrate (sin(t), t, %pi + x, 2*%pi + x);
                        x + 2 %pi
                       /
                       [
(%o6)                  I          sin(t) dt
                       ]
                       /
                        x + %pi
(%i7) checklimits (%);
(%o7)    [b = x + 2 %pi, a = x + %pi, x = t, f = sin(t)]
(%i8) a;
(%o8)                          x + %pi
(%i9) b;
(%o9)                         x + 2 %pi
(%i10) f;
(%o10)                         sin(t)
(%i11) x;
(%o11)                            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 irá ser 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 intermediária (`%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 a você 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 usuário. `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 atualmente 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 padrão: `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 defindas 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 padrão: `[default_let_rule_package]`

`let_rule_packages` é uma lista de todos os pacotes de regras `let` definidos pelo usuário 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, exceto 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 usuário) 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 algorítmo 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 afetam regras existentes.

`propvars (matchdeclare)` retorna a lista de todas as variáveis para as quais exista uma propriedade `matchdeclare`. `printprops (`*a*`, matchdeclare)` retorna o predi-

cado 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* and *delimitador_d*. Os delimitadores são especificados como seqüências 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 seqüências de caracteres, contanto que o analisador de expressões do Maxima possa distingüir 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 usuário. 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 seqüência 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 seqüência 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 usuário.

```
(%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 usuário (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 escondendo de você, você provavelmente quer 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 atual que coincide com *modelo*; variáveis de modelo em *substituição* são atribuidas a valores coincidentes na expressão atual.

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 conseqüência 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 afeta a regra, e não é afetado pela regra. Uma atribuição para uma variável de modelo que resulta em uma coincidência de regra com sucesso não afeta 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 usuário definidos por meio de `infix`, `prefix`, `postfix`, `matchfix`, e `nofix`, possuem nomes que são seqüências 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 usuário são seqüências 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 `^`.

38 Listas

38.1 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))
```

38.2 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 seqüência 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 atualmente ser um símbolo –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 fator 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 expressõ ou lista *expr*. Veja `first` para maiores detalhes.

get (*a*, *i*) Função

Recupera a propriedade de usuário 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* and *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 afetado 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 padrão: `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.

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.

39 Conjuntos

39.1 Introdução a Conjuntos

Maxima fornece funções de conjunto, tais como intersecção e união, para conjuntos finitos que são definidos por enumeração explícitamente. Maxima trata listas e conjuntos como objetos distintos. Esse recurso torna possível trabalhar com conjuntos que possuem elementos que são ou listas ou conjuntos.

Adicionalmente para funções de conjuntos finitos, Maxima fornece algumas funoes relacionadas a análise combinatória; essas incluem os números de Stirling de primero e de segundo tipo, os números de Bell, coefincientes multinomiais, partições de inteiros não negativos, e umas poucas outras. Maxima também define uma função delta de Kronecker.

39.1.1 Usage

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. Conjuntos são sempre mostrados entre chaves (`{ ... }`).

Se um elemento é listado mais de uma vez, a simplificação 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 (i.e., 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 conjuntos `x` e `y` são iguais fornecendo `is(x = y)` avaliando para `true`. Dessa forma `rat(x)` e `x` são iguais como elementos de conjuntos; conseqüentemente,

```
(%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 distintos elementos de conjunto; 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 elemeto 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 redundncias de outros conjuntos, você pode precisar 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 esta'simplificado quando seus elementos não são redundantes e o conjunto está ordenado. A versão corrente das funções de conjunto usam a função do Máxima `orderlessp` para ordenar conjuntos; odavia, *versões futuras das funções de conjunto podem 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 objetos distintos; funções tais como `union` e `intersection` reclamam se qualquer argumetno não for um conjunto. se você precisar aplicar uma função de conjunto a uma lista, use a função `setify` para converter essa lsita para um 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 elemetnos de conjunto 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 em um 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 seção ções para Conjuntos-snt [Definições para Conjuntos], página ções para Conjuntos-pg passui uma lista completa das funções de conjunto no Maxima.

39.1.2 Iterações entre Elementos de Conjuntos

Existem dois camainhos para fazer iterações sobre elementos de conjuntos. Um caminho é usar `map`; por exemplo:

```
(%i1) map (f, {a, b, c});
(%o1)                  {f(a), f(b), f(c)}
```

O outro caminho é 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
```

A função Maxima `first` e `rest` trabalham atualmente sobre conjuntos. Aplicada a um conjunto, `first` retorna o primeiro elemento mostrado de um conjunto; qual élemento que é mostrado pode ser dependente da implementação. Se `s` for um conjunto, então `rest(s)` é equivalente a `disjoin(first(s), s)`. Atualmente, existem outras funções do Maxima que trabalham corretamente sobre conjuntos. Em futuras versões das funções de conjunto, `first` e `rest` podem vir a funcionar diferentemente ou não completamente.

39.1.3 Bugs

As funções de conjunto usam a função Maxima `orderlessp` para organizar os elementos de cum conjunto e a função (a nível de Lisp) `like` para testar a igualdade entre elementos de conjuntos. Ambas essas funções possuem falhas conhecidas que podem se manifestar se você tentar usar conjuntos com elementos que são listas ou matrizes que contenham expressões na forma racional canônica (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 fique exitante com um erro (a mensagem de erro depende de qual a versão do Lisp seu Maxima está usando). 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`; elas não são caudadas por falhas nas funções de conjunto. Para ilustrar, tente 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 sejam corrigidas, não construa conjuntos com com elementos que sejam listas ou matrizes contendo expressões na forma racional canônica (CRE); um conjunto com um elemento na forma CRE, todavia, pode não ser um problema:

```
(%i1) {x, rat (x)};
(%o1)                           {x}
```

A `orderlessp` do Maxima possui outra falha que pode causr problemas com funções de conjunto, sabidamente o predicado de ordenação `orderlessp` é não 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 conjutno bem como com funções 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 simplificado usando `ratsimp`.

Os mecanismos `orderless` e `ordergreat` do Maxima são incompatíveis com as funções de conjunto. Se você rpecisar usar ou `orderless` ou `ordergreat`, chame todas essas funções antes de construir quaisquer conjuntos, e não chame `unorder`.

Se você encontrar alguma coisa que você pense ser uma falha em alguma funçõ de conjunto, por favor relate isso para a base de dados de falhas do Maxima. Veja `bug_report`.

39.1.4 Authors

Stavros Macrakis de Cambridge, Massachusetts e Barton Willis da Universidade e Nebraska e Kearney (UNK) escreveram as fnções de conjunto do Maxima e sua documentação.

39.2 Definições para Conjuntos

adjoin (*x*, *a*) Função

Retorna a união do conjunto *a* com `{`*x*`}`.

`adjoin` reclama se *a* não for um conjunto literal.

`adjoin(`*x*`,` *a*`)` e `union(set(`*x*`),` *a*`)` são equivalentes; todavia, `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 number. `belln(n)` é o número de partições de um conjunto *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` distribui sobre equações, listas, matrizes e conjuntos.

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

Retorna o número de elementos distintos do conjunto *a*.

`cardinality` ignora elementos redundantes mesmo quando a simplificação está dessabilitada.

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` reclama 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` reclama 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*`)` simplifica para um conjunto de inteiros quando *n* for um inteiro não nulo. O cojunto dos divisores inclui os elementos 1 e *n*. Os divisores de um inteiro negativo são os divisores de seu valor absoluto.

`divisors` distribui sobre equações, listas, matrizes, e conjuntos.

Exemplos:

Podemos verificar que 28 é um número perfeito: a adição de seus divisores (exceto 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)` retorna os divisores sem fazer a reavaliação de `divisors(8)`.

```
(%i1) divisors (a);
(%o1)                        divisors(a)
(%i2) subst (8, a, %);
(%o2)                       {1, 2, 4, 8}
```

`divisors` distribui sobre equações, listas, matrizes, e conjuntos.

```
(%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` reclama 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 a 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* nõ for um relação de equivalência, `equiv_classes` aceita sem reclamação, mas o resultado é geralmente incorreto 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 multiplo de 3.

```
(%i1) equiv_classes ({1, 2, 3, 4, 5, 6, 7}, lambda ([x, y], remainder (x - y,
(%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 sgundo 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 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* devem ter o mesmo comprimento. Quando `maperror` for falso, argumentos listas são efetivamente truncados para o comprimento da menor lista.

Retorna valores do predicado *f* que avaliam (via `is`) para alguma coisa outra que não `true` ou `false` são governados através do sinalizador 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
```

Retorna valores do predicado *f* que avalia para alguma coisa outra que não `true` ou `false` são governados por meio 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) 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

Retorna o subconjunto de *s* para o qual a função *f* toma valore máximos ou mínimos.

`extremal_subset(`*s*`,` *f*`, max)` retorna o subconjunto do conjunto ou lista *s* para os quais a função real *f* assume valor maximo.

`extremal_subset(`*s*`,` *f*`, min)` retorna o subconjuno do conjunto ou lista *s* para a qual a função real *f* assume 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

Recebe argumentos de subexpressões que possuem o mesmo operator como *expr* e constrói uma expressão a partir desses argumentos coletados.

subexpressões nas quais o operador é diferente do operador principal de `expr` são copiadas sem modificação, mesmo se elas, in turn, contiverem a mesma subexpressão na qual o operador seja o mesmo que em `expr`.

Pode ser possível para `flatten` construir 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, expressãoes racionais canônicas (CRE), não podem usar a função `flatten`; nesses casos, `flatten` retorna seus argumentos sem modificação.

Exemplos:

Aplicado a uma lista, `flatten` reune todos os elementos de lista que são 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` reune todos os elementos de conjunto que são conjuntos.

```
(%i1) flatten ({a, {b}, {{c}}});
(%o1)                         {a, b, c}
(%i2) flatten ({a, {[a], {a}}});
(%o2)                         {a, [a]}
```

`flatten` é similar ao efeito de declarar o operador principal para ser enário. Todavia, `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
```

Pode ser possível para `flatten` construir 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 oeradr de conjutno em *a* por um operadro de lista, e retorna o resultado. `full_listify` substitui operadores de conjunto em subexpressões restantes, mesmo se o operadro principal não for conjunto (`set`).

`listify` substitui somente 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, essa não lista é retornada em sua forma original e sem modificações.

`setify` substitui somente 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

Retorna partições inteiras de *n*, isto é, listas de inteiros cuja soma dos elementos de cada lista é *n*.

`integer_partitions(`*n*`)` retorna o conjunto de todas as partições do inteiro *n*. Cada partição é uma lista ordenada do maior para o menor.

`integer_partitions(`*n*`,` *len*`)` retorna todas as partições que possuem comprimento *len* ou menor; nesse caso, zeros são anexado ao final de cada partição de comprimento menor que *len* terms to make each partition have exactly *len* terms. Each partition is a list sorted from greatest to least.

Uma lista $[a_1, ..., a_m]$ é uma partição de inteiros não negativos n quando (1) cada a_i é um inteiro não nulo, e (2) $a_1 + ... + a_m = n$. Dessa forma 0 não tem partiçãoes.

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` reclama se qualquer argumento 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 identicos ou demonstadamente equivalentes, e simplifica para 0 quando *x* e *y* demonstradamente não equivalentes. De outra forma, se não for certo que *x* e *y* são equivalentes, e `kron_delta` simplifica 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*.

Specificamente, `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 e também não for 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`.

De outra forma, `sign(abs(`*x* - *y*`))` é alguma coisa outra que não `pos` ou `zero`, ou se for `zero` e *x* - *y* for umnúmero em ponto flutuante. Nesses casos, `kron_delta` retorna um expressão substantiva.

`kron_delta` é declarada para ser simétrica. Isto é, `kron_delta(`*x*`,` *y*`)` é igual a `kron_delta(`*y*`,` *x*`)`.

Exemplos:

Os argumentos de `kron_delta` são identicos. `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 serem 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) Warning:  Float to bigfloat conversion of 0.250.25 - 0.25b0;
(%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) Warning:  Float to bigfloat conversion of 0.25kron_delta (0.25, 0.25b0);
(%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 operadores *F* para uma função de `n` operadores usando composição, onde *s* é uma lista.

`lreduce(`*F*`,` *s*`)` returns `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 primeiramente aplicada à lista de elementos *leftmost - mais à esquerda*, 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 de dois argumentos internos (já definidos por padrão) 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 *s*é um conjunto ou lista de listas. Para gerar cada elemento do conjunto, *expr* é avaliada com as variáveis *x* paralelamente a um elemento 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 subscritos. 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` distribui sobre equações, listas, matrizes, e conjuntos.

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

Retorna o coeficiente multinomial.

Quando cada *a_k* for um inteiro não negativo, o coeficiente multinomial fornece o número de formas possíveis de colocar *a_1* + ... + *a_n* objetos distintos em *n* caixas com *a_k* elementos na *k*'ésima caixa. Em geral, `multinomial_coeff` (*a_1*, ..., *a_n*) avalia para (*a_1* + ... + *a_n*)!/(*a_1*! ... *a_n*!).

`multinomial_coeff()` (sem argumentos) avalia para 1.

`minfactorial` pode estar apta a simplificar o valor retornado 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

Retorna 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

Retorna 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 atualmente 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 argumetno 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 argumetno 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)$
(%o3)                         90 = 90
(%i4) cardinality(p) = stirling2 (6, 3);
```

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)$
(%o3)                           {3}
(%i4) map (cardinality, p);
```

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)$
(%o3)                 {{0, 1, 2, 3, 4, 5}}
(%i4) map (lambda ([x], apply (union, listify (x))), p);
```

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 efetivamente 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 argumetnos 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 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.

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 exatamente 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 exatamente 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]
```

40 Definição de Função

40.1 Introdução a Definição de Função

40.2 Função

40.2.1 Ordinary functions

Para definir uma função no Maxima você usa o operador :=. E.g.

```
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.

Você pode também definir uma função com um número variável de argumentos, teno um argumento final que é atribuído para uma lista de argumentos extras:

```
(%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 de uma função é uma expressão. Desse modo Se você quer uma seqüência de expressões, você faz

```
f(x) := (expr1, expr2, ...., exprn);
```

e o valor de *exprn* é que é retornado pela função.

Se você deseja fazer um `return` de alguma expressão dentro da função então você deve 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. Uma vez que você retorna do `block`, ou descarta-o, os valores antigos (quaisquer que sejam) das variáveis serão restaurados. É certamente uma boa idéia para proteger suas variáveis nesse caminho. Note que as atribuições em variáveis do bloco, são concluídas em paralelo. Isso significa, que se tiver usado `c: a` acima, o valor de `c` será o valor de `a` a partir do momento em que vocêntrou no bloco, mas antes `a` foi associado. 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 você acessar o valor antigo. Dessa forma o lado direito de atribuições, é avaliado no contexto inserido, antes que qualquer avaliação ocorra. Usando apenas `block ([x], ...` faremos com que o `x` ter a si mesmo como valor, apenas como tivesse você entrar numa breve sessão **Maxima**.

Os atuais argumentos para uma função são tratados exatamente 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`.

40.2.2 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 exatamente 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, exceto para `kill(`a`)` ou `remarray(`a`)`, o qual remove também remove a definição da função de array.

40.3 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 seqüêncialmente, 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 seqüêncialmente.

```
(%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 usuário. 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 multiplos 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<>
```

40.4 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 seqüência e retorna o valor da última expressão avaliada. A seqüência 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` (exceto 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 usuário 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 (*filename, f_1, ..., f_n*) Função

Traduz funções Maxima *f_1*, ..., *f_n* para Lisp e escreve o código traduzido no arquivo *filename*.

As traduções Lisp não são avaliadas, nem é o arquivo 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 (*functions*) 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 (functions)` compila todas as funções definidas pelo usuário.

`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 uma função chamada *f* com argumentos *x_1*, ..., *x_n* e corpo da função *expr*.

`define` não avalia seu primeiro argumento na maioria dos casos, e avalia seu segundo argumento a menos que explicitamente seja pedido o contrário. Todavia, se o primeiro argumento for uma expressão da forma `ev (`*expr*`)`, `funmake (`*expr*`)`, ou `arraymake (`*expr*`)`, o primeiro argumento será avaliado; isso permite para o nome da função seja calculado, também como o corpo.

`define` é similar ao operador de definição de função `:=`, mas quando `define` aparece dentro da função, a definição é criada usando o valor de `expr` em tempo de execução em lugar de em tempo de definição da função que a contém.

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`.

o comando `define` cria funções de array (chamadas com argumentsos entre colchêtes `[ ]`) da mesma forma que funções comuns.

Exemplos:

```
(%i1) foo: 2^bar;
                                bar
(%o1)                          2
(%i2) g(x) := (f_1 (y) :=   foo*x*y,
               f_2 (y) := ''foo*x*y,
       define (f_3 (y),     foo*x*y),
       define (f_4 (y),   ''foo*x*y));
                                                      bar
(%o2) g(x) := (f_1(y) := foo x y, f_2(y) := 2    x y,
                                                           bar
               define(f_3(y), foo x y), define(f_4(y), 2    x y))
(%i3) functions;
(%o3)                          [g(x)]
(%i4) g(a);
                                      bar
(%o4)                  f_4(y) := a 2    y
(%i5) functions;
(%o5)        [g(x), f_1(y), f_2(y), f_3(y), f_4(y)]
(%i6) dispfun (f_1, f_2, f_3, f_4);
(%t6)                   f_1(y) := foo x y

                                    bar
(%t7)                  f_2(y) := 2    x y

                                      bar
(%t8)                  f_3(y) := a 2    y

                                      bar
(%t9)                  f_4(y) := a 2    y

(%o9)                         done
```

define_variable (*name, default_value, mode*) Função

Introduz uma variável global dentro do ambiente Maxima. `define_variable` é útil em pacotes escritos pelo usuário, 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=
 -- 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 usuário *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 selecionadas por um valor subscrito particular, ou uma função subscrita definida com uma constante subscrita.

`dispfun (all)` mostra todas as funções definidas pelo usuário como dadas pelas `functions`, `arrays`, e listas de `macros`, omitindo funções subscritas definidas com constantes subscritas.

`dispfun` cria um Rótulo de expressão intermediária (`%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 intermediárias 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 padrão: `[]`

`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 usuário ou em um arquivo 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 famíla de funções subscritas selecionada 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 intermediária 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 padrão: `false`

`macroexpansion` controla recursos avançados que afetam 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 subseqüênte 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 padrão: `true`

Quando `mode_checkp` é `true`, `mode_declare` verifica os modos de associação de variáveis.

mode_check_errorp Variável de opção

Valor padrão: `false`

Quando `mode_check_errorp` é `true`, `mode_declare` chama a função `"error"`.

mode_check_warnp Variável de opção

Valor padrão: `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 subseqüênte 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 objetos 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`. Todavia, 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 . Isso tem númerosas utilidades, e.g., se você soube que `first (l)` retornou um número então você pode escrever `mode_identity (number, first (l))`. Todavia, um mais eficiente caminho para fazer isso é definir uma nova primitiva,

```
firstnumb (x) ::= buildq ([x], mode_identity (number, x));
```

e usar `firstnumb` toda vez que você pegar o primeiro de uma lista de números.

transcompile Variável de opção

Valor padrão: `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 (*functions*) Função
translate (*all*) Função

Traduz funções definidas pelo usuário *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 (functions)` traduz todas as funções definidas pelo usuário.

Funções a serem traduzidas incluir˜ao uma chamada para `mode_declare` no início quando possível com o objetivo 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 correto será gerado mas não será um código ótimo. Dessa forma, o usuário 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 usuário 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 canônicas(CRE). Também a escolha `prederror: false` não traduzirá.

`savedef` - se `true` fará com que a versão Maxima de uma função usuário 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_filename*) Função
translate_file (*maxima_filename, lisp_filename*) Função

Traduz um arquivo com código Maxima para um arquivo com código Lisp. `translate_file` retorna uma lista de três nomes de arquivo: O nome do arquivo Maxima, o nome do arquivo Lisp, e o nome do arquivo 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")` exceto por certas restrições, o uso de `''` e `%`, por exemplo.

`translate_file` (*maxima_filename*) traduz um arquivo Maxima *maxima_filename* para um similarmente chamado arquivo Lisp. Por exemplo, `foo.mac` é traduzido em

`foo.LISP`. O nome de arquivo Maxima pod incluir nome ou nomes de diretório(s), nesse caso o arquivo de saída Lisp é escrito para o mesmo diretório que a entrada Maxima.

`translate_file` (*maxima_filename*, *lisp_filename*) traduz um arquivo Maxima *maxima_filename* em um arquivo Lisp *lisp_filename*. `translate_file` ignora a extensão do nome do arquivo, se qualquer, de `lisp_filename`; a extensão do arquivo de saída Lisp é sempre `LISP`. O nome de arquivo Lisp pode incluir um nome ou nomes de diretórios), nesse caso o arquivo de saída Lisp é escrito para o diretório especificado.

`translate_file` também escreve um arquivo de mensagens de alerta do tradutor em vários graus de severidade. A extensão do nome de arquivo desse arquivo é `UNLISP`. Esse arquivo pode conter informação valiosa, apesar de possivelmente obscura, para rastrear erros no código traduzido. O arquivo `UNLISP` é sempre escrito para o mesmo diretório 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`, `tr_warn_undefined_variable`, and `tr_windy`.

transrun Variável de opção

Valor padrão: `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 padrão: `true`

Se `translate_fast_arrays` for `false`, referências a arrays no Código Lisp emitidas por `translate_file` são afetadas 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 padrão: `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 afeta 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 padrão: `false`

Quando `tr_file_tty_messagesp` é `true`, messagens geradas por `translate_file` durante a tradução de um arquivo são mostradas sobre o console e inseridas dentro do arquivo UNLISP. Quando `false`, messagens sobre traduções de arquivos são somente inseridas dentro do arquivo UNLISP.

tr_float_can_branch_complex Variável de opção

Valor padrão: `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 padrão: `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 corretas 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 padrão: `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 padrão: 100

`tr_optimize_max_loop` é número máximo de vezes do passo de macro-expansão e otimização que o tradutor irá executar considerando uma forma. Isso é para capturar erros de expansão de macro, e propriedades de otimização não terminadas.

tr_semicompile Variável de opção

Valor padrão: `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 padrão:

```
[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 afetam 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 padrão: `true`

- Emite um alerta quando chamadas de função estão sendo feitas por um caminho que pode não ser correto devido a declarações impróprias que foram feitas em tempo de tradução.

tr_warn_fexpr Variável de opção

Valor padrão: `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 padrão: `compfile`

- Emite um alerta se a função `meval` recebe chamadas. Se `meval` é chamada isso indica problemas na tradução.

tr_warn_mode Variável

Valor padrão: `all`

- Emite um alerta quando a variáveis forem atribuídos valores inapropriados para seu modo.

tr_warn_undeclared Variável de opção

Valor padrão: `compile`

- Determina quando enviar alertas sobre variáveis não declaradas para o TTY.

tr_warn_undefined_variable Variável de opção

Valor padrão: `all`

- Emite um alerta quando variáveis globais indefinidas forem vistas.

tr_windy Variável de opção

Valor padrão: `true`

- Gera comentários "de grande ajuda" e dicas de programação.

compile_file (*filename*) Função
compile_file (*filename, compiled_filename*) Função
compile_file (*filename, compiled_filename, lisp_filename*) Função

Traduz o arquivo Maxima *filename* 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 arquivos: o arquivo original do Maxima, o nome da tradução Lisp, uma arquivo de notas sobre a tradução, e o nome do arquivo 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 arquivo Maxima, atribuições para sinalizadores traduzidos (`tr_numer`, etc.) não têm efeito sobre a tradução.

filename pode não conter declarações `:lisp`.

`compile_file` avalia seus argumentos.

declare_translated (*f_1, f_2, ...*) Função

Quando traduzindo um arquivo do código Maxima para Lisp, é importante para o programa tradutor saber quais funções no arquivo são para serem chamadas como funções traduzidas ou compiladas, e quais outras são apenas funções Maxima ou indefinidas. Colocando essa declaração no topo do arquivo, 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.

41 Fluxo de Programa

41.1 Introdução a Fluxo de Programa

Maxima fornece um `do` para ciclos iterativos, também contruções mais primitivas tais como `go`.

41.2 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 ativa.

`backtrace()` imprime toda a pilha de chamadas.

`backtrace (`*n*`)` imprime as *n* mais recentes chamadas a funções, incluindo a função correntemente ativa.

`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 ativa.

```
(%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 afetar 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 atualização 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 fato, 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 arquivos `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 seqüência de caracteres de formato, que junta todas as seqüências de caracteres entre os argumentos *expr_1*, ..., *expr_n*, e os elementos restantes são os valores de quaisquer argumentos que não são seqüências de caracteres.

`errormsg()` formata e imprime `error`. Isso efetivamente 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

A declaração `if` é usada para execução condicional. A sintaxe é:

```
if <condição> then <expr_1> else <expr_2>
```

O resultado de uma declaração `if` será *expr_1* se condição for `true` e *expr_2* de outra forma. *expr_1* e *expr_2* são quaisquer expressões Maxima (incluindo declarações `if` aninhadas), e *condição* é uma expressão que avalia para `true` ou `false` e é composto de operadores relacionais e lógicos que são os seguintes:

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 objeto. 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 padrão: `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 objeto.

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 (exceto para simplificações e o caso onde `map` faz um `apply`).

prederror Variável de opção

Valor padrão: `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 fatoração 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))
```

Nesse caso, você pega a mesma resposta em ambos os caminhos.

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.

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 afetam 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
```

42 Depurando

42.1 Depurando o Código Fonte

Maxima tem um depurador interno de código fonte. O usuário pode escolher um ponto de parada em uma função, e então caminhar linha por linha a partir daí. A pilha de chamadas po ser examinada, juntamente com as variáveis associadas àquele nível.

O comando `:help` ou `:h` mostra a lista de comando de depuração. (Em geral, comandos podem ser abreviados se a abreviação for única. Se não for única, as alternativas podem ser listadas.) Dentro do depurador, o usuário pode também usar qualquer 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 usuário 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 abaixo.

```
(%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 arquivo `/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

Se o usuário estiver rodando o código sob o GNU emacs em uma janela shell (shel dbl), ou está rodando a versão de interface gráfica, `xmaxima`, então se ele para em um ponto de parada, ele verá sua posição corrente no arquivo fonte a qua será mostrada na outra metade da janela, ou em vermelho brilhante, ou com um pequeno seta apontando na direita da linha. Ele pode avançar uma linha por vez digitando M-n (Alt-n).

Sob Emacs você pode executar em um shell `dbl`, o qual requer o arquivo `dbl.el` no diretório elisp. Tenha certeza que instalou os arquivos elisp ou adicionou o diretório elisp do Macima ao seu caminho: e.g., adicione o seguinte ao seu arquivo '`.emacs`' ou ao seu arquivo `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 você 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 usuário pode escolher um ponto de parada em certa linha do arquivo digitando `C-x space`. Isso encontra qual a função que o cursor está posicionado, e então mostra qual a linha daquela função que 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 usuário deve ter maxima-mode.el habilitado na janela na qual o arquivo `foobar.mac` estiver interagindo. Existe comandos adicional disponíveis naquela janela de arquivo, tais como avaliando a função dentro do Maxima, através da digitação de `Alt-Control-x`.

42.2 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 uma forma Lisp você pode digitar `:lisp` seguido pela forma a ser avaliada.

```
(%i1) :lisp (+ 2 3)
5
```

O número de argumentos tomados depende do comando em particular. Também, você não precisa digitar 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 seqüência de caracteres, então essa seqüência de caracteres é assumida referir-se a um arquivo, e `n` é o deslocamente a partir do início do arquivo. O deslocamento é opcional. Se for omitido, é assumido ser zero (primeira linha da função ou do arquivo).

`:bt` Imprime na tela uma lista da pilha de frames

`:continue`
: Continua a computação

`:delete` Remove o ponto de parada selecionado, ou todos se nenum for especificado

`:disable` Desabilita os pontos de parada selecionados, 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`, exceto `: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

42.3 Definições para Depuração

refcheck Variável de opção

Valor padrão: `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 padrão: `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 atualmente 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 si mesma, e.g., `X: ’X`.

setcheckbreak Variável de opção

Valor padrão: `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 atribuição seja concluída. Nesse ponto, `setval` retém o valor para o qual a variável está para ser atribuída. Conseqüentemente, 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. Conseqüentemente, 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 coletadas. `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 atualmente na lista de funções estatisticamente monitoradas.

Após `untimer (f)` ser executada, `timer_info (f)` ainda retorna estatisticas de tempo previamente coletadas, embora `timer_info()` (sem argumentos) não retorna informações sobre qualquer função que não estiver atualmente 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 atualmente 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 atualmente 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 objetos 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`.

43 augmented_lagrangian

43.1 Definições para augmented_lagrangian

niter Variável de opção

Valor padrão: 10

Número de iterações para `augmented_lagrangian_method`.

augmented_lagrangian_method (*FOM, xx, constraints, yy*) Função

Método do Lagrangiano Aumentado para otimização restrita. *FOM* é o algarismo da expressão de método, *xx* é uma lista de variáveis sobre as quais minimizar, *constraints* é uma lista de expressões a serem mantidas iguais a zero, e *yy* é uma lista de suposições iniciais para *xx*.

Atualmente esse código minimiza o Lagrangiano aumentado resolvendo para um ponto estacionário de seu gradiente. Isso é terrivelmente fraco, e o código pode ser melhorado anexando um gradiente conjugado ou um algorítmo de minimização quasi-Newton.

Para referência veja

`http://www-fp.mcs.anl.gov/otc/Guide/OptWeb/continuous/constrained/nonlinearcon`

e

`http://www.cs.ubc.ca/spider/ascher/542/chap10.pdf`

O pacote `mnewton` (para resolver grad $L = 0$) tem que ser chamado antes do `augmented_lagrangian_method`.

Exemplo:

```
(%i1) load (mnewton)$

(%i2) load("augmented_lagrangian")$

(%i3) FOM: x^2 + 2*y^2;
                            2    2
(%o3)                     2 y  + x
(%i4) xx: [x, y];
(%o4)                       [x, y]
(%i5) C: [x + y - 1];
(%o5)                    [y + x - 1]
(%i6) yy: [1, 1];
(%o6)                       [1, 1]
(%i7) augmented_lagrangian_method (FOM, xx, C, yy);
(%o7)           [0.6478349834, 0.3239174917]
```

Para usar essa função escreva primeiro `load("mnewton")` e em seguida `load("augmented_lagrangian")`. Veja também `niter`.

44 bode

44.1 Definitions for bode

bode_gain (*H, range, ...plot_opts...*) Function

Function to draw Bode gain plots.

Examples (1 through 7 from

`http://www.swarthmore.edu/NatSci/echeeve1/Ref/Bode/BodeHow.html`,

8 from Ron Crummett):

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

(%i2) H1 (s) := 100 * (1 + s) / ((s + 10) * (s + 100))$

(%i3) bode_gain (H1 (s), [w, 1/1000, 1000])$

(%i4) H2 (s) := 1 / (1 + s/omega0)$

(%i5) bode_gain (H2 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i6) H3 (s) := 1 / (1 + s/omega0)^2$

(%i7) bode_gain (H3 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i8) H4 (s) := 1 + s/omega0$

(%i9) bode_gain (H4 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i10) H5 (s) := 1/s$

(%i11) bode_gain (H5 (s), [w, 1/1000, 1000])$

(%i12) H6 (s) := 1/((s/omega0)^2 + 2 * zeta * (s/omega0) + 1)$

(%i13) bode_gain (H6 (s), [w, 1/1000, 1000]),
                  omega0 = 10, zeta = 1/10$

(%i14) H7 (s) := (s/omega0)^2 + 2 * zeta * (s/omega0) + 1$

(%i15) bode_gain (H7 (s), [w, 1/1000, 1000]),
                  omega0 = 10, zeta = 1/10$

(%i16) H8 (s) := 0.5 / (0.0001 * s^3 + 0.002 * s^2 + 0.01 * s)$

(%i17) bode_gain (H8 (s), [w, 1/1000, 1000])$
```

To use this function write first `load("bode")`. See also `bode_phase`

bode_phase (*H, range, ...plot_opts...*) Function

Function to draw Bode phase plots.

Examples (1 through 7 from

`http://www.swarthmore.edu/NatSci/echeeve1/Ref/Bode/BodeHow.html`,

8 from Ron Crummett):

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

(%i2) H1 (s) := 100 * (1 + s) / ((s + 10) * (s + 100))$

(%i3) bode_phase (H1 (s), [w, 1/1000, 1000])$

(%i4) H2 (s) := 1 / (1 + s/omega0)$

(%i5) bode_phase (H2 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i6) H3 (s) := 1 / (1 + s/omega0)^2$

(%i7) bode_phase (H3 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i8) H4 (s) := 1 + s/omega0$

(%i9) bode_phase (H4 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i10) H5 (s) := 1/s$

(%i11) bode_phase (H5 (s), [w, 1/1000, 1000])$

(%i12) H6 (s) := 1/((s/omega0)^2 + 2 * zeta * (s/omega0) + 1)$

(%i13) bode_phase (H6 (s), [w, 1/1000, 1000]),
                   omega0 = 10, zeta = 1/10$

(%i14) H7 (s) := (s/omega0)^2 + 2 * zeta * (s/omega0) + 1$

(%i15) bode_phase (H7 (s), [w, 1/1000, 1000]),
                   omega0 = 10, zeta = 1/10$

(%i16) H8 (s) := 0.5 / (0.0001 * s^3 + 0.002 * s^2 + 0.01 * s)$

(%i17) bode_phase (H8 (s), [w, 1/1000, 1000])$

(%i18) block ([bode_phase_unwrap : false],
              bode_phase (H8 (s), [w, 1/1000, 1000]));

(%i19) block ([bode_phase_unwrap : true],
              bode_phase (H8 (s), [w, 1/1000, 1000]));
```

To use this function write first `load("bode")`. See also `bode_gain`

45 cholesky

45.1 Definitions for cholesky

cholesky (*A*) Function

Compute Cholesky decomposition of *A*, a lower-triangular matrix *L* such that `L . transpose(L) = A`.

Some examples follow.

Example 1:

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

(%i2) A : matrix ([a, b, c], [d, e, f], [g, h, i]);
                        [ a  b  c ]
                        [         ]
(%o2)                   [ d  e  f ]
                        [         ]
                        [ g  h  i ]
(%i3) A2 : transpose (A) . A;
      [   2    2    2                                      ]
      [  g  + d  + a     g h + d e + a b   g i + d f + a c ]
      [                                                    ]
(%o3) [                    2    2    2                    ]
      [ g h + d e + a b   h  + e  + b     h i + e f + b c  ]
      [                                                    ]
      [                                    2    2    2     ]
      [ g i + d f + a c   h i + e f + b c   i  + f  + c     ]
(%i4) B : cholesky (A2)$

(%i5) B . transpose (B) - A2;
                        [ 0  0  0 ]
                        [         ]
(%o5)                   [ 0  0  0 ]
                        [         ]
                        [ 0  0  0 ]
```

Example 2:

```
(%i6) A : matrix ([2, 3, 4], [-2, 2,- 3], [11, -2, 3]);
                      [  2   3    4  ]
                      [              ]
(%o6)                 [ - 2  2   - 3 ]
                      [              ]
                      [ 11   - 2  3  ]
(%i7) A2 : transpose (A) . A;
                      [ 129   - 20  47 ]
                      [                ]
(%o7)                 [ - 20   17   0  ]
                      [                ]
```

```
                    [  47    0    34 ]
(%i8) B : cholesky (A2);
       [  sqrt(129)         0             0      ]
       [                                         ]
       [      20         sqrt(1793)              ]
       [ - ---------     ----------         0      ]
(%o8)  [   sqrt(129)     sqrt(129)                ]
       [                                         ]
       [     47        940 sqrt(129)      153     ]
       [  ---------    --------------  ---------- ]
       [  sqrt(129)    129 sqrt(1793)  sqrt(1793) ]
(%i9) B . transpose (B) - A2;
                        [ 0  0  0 ]
                        [         ]
(%o9)                   [ 0  0  0 ]
                        [         ]
                        [ 0  0  0 ]
```

To use this function write first `load("cholesky")`.

46 descriptive

46.1 Introduction to descriptive

Package `descriptive` contains a set of functions for making descriptive statistical computations and graphing. Together with the source code there are three data sets in your Maxima tree: `pidigits.data`, `wind.data` and `biomed.data`. They can be also downloaded from the web site `www.biomates.net`.

Any statistics manual can be used as a reference to the functions in package `descriptive`.

For comments, bugs or suggestions, please contact me at *'mario AT edu DOT xunta DOT es'*.

Here is a simple example on how the descriptive functions in `descriptive` do they work, depending on the nature of their arguments, lists or matrices,

```
(%i1) load (descriptive)$
(%i2) /* univariate sample */   mean ([a, b, c]);
                               c + b + a
(%o2)                          ---------
                                   3
(%i3) matrix ([a, b], [c, d], [e, f]);
                               [ a  b ]
                               [      ]
(%o3)                          [ c  d ]
                               [      ]
                               [ e  f ]
(%i4) /* multivariate sample */ mean (%);
                         e + c + a  f + d + b
(%o4)                   [---------, ---------]
                             3          3
```

Note that in multivariate samples the mean is calculated for each column.

In case of several samples with possible different sizes, the Maxima function `map` can be used to get the desired results for each sample,

```
(%i1) load (descriptive)$
(%i2) map (mean, [[a, b, c], [d, e]]);
                           c + b + a  e + d
(%o2)                     [---------, -----]
                               3        2
```

In this case, two samples of sizes 3 and 2 were stored into a list.

Univariate samples must be stored in lists like

```
(%i1) s1 : [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5];
(%o1)           [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
```

and multivariate samples in matrices as in

```
(%i1) s2 : matrix ([13.17, 9.29], [14.71, 16.88], [18.50, 16.88],
             [10.58, 6.63], [13.33, 13.25], [13.21,  8.12]);
```

```
                          [ 13.17  9.29  ]
                          [              ]
                          [ 14.71  16.88 ]
                          [              ]
                          [ 18.5   16.88 ]
(%o1)                     [              ]
                          [ 10.58  6.63  ]
                          [              ]
                          [ 13.33  13.25 ]
                          [              ]
                          [ 13.21  8.12  ]
```

In this case, the number of columns equals the random variable dimension and the number of rows is the sample size.

Data can be introduced by hand, but big samples are usually stored in plain text files. For example, file `pidigits.data` contains the first 100 digits of number `%pi`:

```
3
1
4
1
5
9
2
6
5
3 ...
```

In order to load these digits in Maxima,

```
(%i1) load (numericalio)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) length (s1);
(%o3)                          100
```

On the other hand, file `wind.data` contains daily average wind speeds at 5 meteorological stations in the Republic of Ireland (This is part of a data set taken at 12 meteorological stations. The original file is freely downloadable from the StatLib Data Repository and its analysis is discused in Haslett, J., Raftery, A. E. (1989) *Space-time Modelling with Long-memory Dependence: Assessing Ireland's Wind Power Resource, with Discussion.* Applied Statistics 38, 1-50). This loads the data:

```
(%i1) 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]
```

Some samples contain non numeric data. As an example, file `biomed.data` (which is part of another bigger one downloaded from the StatLib Data Repository) contains four blood measures taken from two groups of patients, `A` and `B`, of different ages,

```
(%i1) 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]
```

The first individual belongs to group `A`, is 30 years old and his/her blood measures were 167.0, 89.0, 25.6 and 364.

One must take care when working with categorical data. In the next example, symbol `a` is asigned a value in some previous moment and then a sample with categorical value `a` is taken,

```
(%i1) a : 1$
(%i2) matrix ([a, 3], [b, 5]);
                             [ 1  3 ]
(%o2)                        [      ]
                             [ b  5 ]
```

46.2 Definitions for data manipulation

continuous_freq (*list*) Function
continuous_freq (*list*, *m*) Function

The argument of `continuous_freq` must be a list of numbers, which will be then grouped in intervals and counted how many of them belong to each group. Optionally, function `continuous_freq` admits a second argument indicating the number of classes, 10 is default,

```
(%i1) load (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]]
```

The first list contains the interval limits and the second the corresponding counts: there are 16 digits inside the interval `[0, 1.8]`, that is 0's and 1's, 24 digits in `(1.8, 3.6]`, that is 2's and 3's, and so on.

discrete_freq (*list*) Function

Counts absolute frequencies in discrete samples, both numeric and categorical. Its unique argument is a list,

```
(%i1) load (descriptive)$
(%i2) 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]]
```

The first list gives the sample values and the second their absolute frequencies. Commands `? col` and `? transpose` should help you to understand the last input.

subsample (*data_matrix, logical_expression*) Function
subsample (*data_matrix, logical_expression, col_num, col_num, ...*) Function

This is a sort of variation of the Maxima `submatrix` function. The first argument is the name of the data matrix, the second is a quoted logical expression and optional additional arguments are the numbers of the columns to be taken. Its behaviour is better understood with examples,

```
(%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 ]
```

These are multivariate records in which the wind speeds in the first meteorological station were greater than 18. See that in the quoted logical expression the *i*-th component is refered to as `%c[i]`. Symbol `%c[i]` is used inside function `subsample`, therefore when used as a categorical variable, Maxima gets confused. In the following example, we request only the first, second and fifth components of those records with wind speeds greater or equal than 16 in station number 1 and lesser than 25 knots in station number 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 ]
```

Here is an example with the categorical variables of `biomed.data`. We want the records corresponding to those patients in group `B` who are older than 38 years,

```
(%i1) load (descriptive)$
(%i2) 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 ]
```

Probably, the statistical analysis will involve only the blood measures,

```
(%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 ]
```

This is the multivariate mean of `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
```

Here, the first component is meaningless, since `A` and `B` are categorical, the second component is the mean age of individuals in rational form, and the fourth and last values exhibit some strange behaviour. This is because symbol `NA` is used here to indicate *non available* data, and the two means are of course nonsense. A possible

solution would be to take out from the matrix those rows with NA symbols, although this deserves some loss of information,

```
(%i1) load (descriptive)$
(%i2) 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
```

46.3 Definitions for descriptive statistics

mean (*list*) Function
mean (*matrix*) Function

This is the sample mean, defined as

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Example:

```
(%i1) load (descriptive)$
(%i2) 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*) Function
var (*matrix*) Function

This is the sample variance, defined as

$$\frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^2$$

Example:

```
(%i1) load (descriptive)$
(%i2) load (numericalio)$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) var (s1), numer;
(%o4)                  8.425899999999999
```

See also function var1.

var1 (*list*) Function
var1 (*matrix*) Function

This is the sample variance, defined as

$$\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2$$

Example:

```
(%i1) load (descriptive)$
(%i2) 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 (*list*) Function
std (*matrix*) Function

This is the the square root of function `var`, the variance with denominator n.

Example:

```
(%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]
```

See also functions `var` and `std1`.

std1 (*list*) Function
std1 (*matrix*) Function

This is the the square root of function `var1`, the variance with denominator $n-1$.

Example:

```
(%i1) load (descriptive)$
(%i2) 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]
```

See also functions `var1` and `std`.

noncentral_moment (*list*, *k*) Function
noncentral_moment (*matrix*, *k*) Function

The non central moment of order k, defined as

$$\frac{1}{n}\sum_{i=1}^{n} x_i^k$$

Example:

```
(%i1) load (descriptive)$
(%i2) 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]
```

See also function `central_moment`.

central_moment (*list*, *k*) Function
central_moment (*matrix*, *k*) Function

The central moment of order k, defined as

$$\frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^k$$

Example:

```
(%i1) load (descriptive)$
(%i2) load (numericalio)$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) central_moment (s1, 2), numer; /* the variance */
(%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]
```

See also functions `central_moment` and `mean`.

cv (*list*) Function
cv (*matrix*) Function

The variation coefficient is the quotient between the sample standard deviation (`std`) and the `mean`,

```
(%i1) load (descriptive)$
(%i2) 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]
```

See also functions `std` and `mean`.

mini (*list*) Function
mini (*matrix*) Function

This is the minimum value of the sample *list*,

```
(%i1) load (descriptive)$
(%i2) 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]
```

See also function `maxi`.

maxi (*list*) Function
maxi (*matrix*) Function

This is the maximum value of the sample *list*,

```
(%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]
```

See also function `mini`.

range (*list*) Function
range (*matrix*) Function

The range is the difference between the extreme values.

Example:

```
(%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 (*list, p*) Function
quantile (*matrix, p*) Function

This is the *p*-`quantile`, with *p* a number in $[0, 1]$, of the sample *list*. Although there are several definitions for the sample quantile (Hyndman, R. J., Fan, Y. (1996) *Sample quantiles in statistical packages*. American Statistician, 50, 361-365), the one based on linear interpolation is implemented in package `descriptive`.

Example:

```
(%i1) load (descriptive)$
(%i2) load (numericalio)$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) /* 1st and 3rd quartiles */ [quantile (s1, 1/4), quantile (s1, 3/4)], nu
(%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 (*list*) Function
median (*matrix*) Function

Once the sample is ordered, if the sample size is odd the median is the central value, otherwise it is the mean of the two central values.

Example:

```
(%i1) load (descriptive)$
(%i2) 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]
```

The median is the 1/2-`quantile`.

See also function `quantile`.

qrange (*list*) Function
qrange (*list*) Function

The interquartilic range is the difference between the third and first quartiles, `quantile(list,3/4) - quantile(list,1/4)`,

```
(%i1) load (descriptive)$
(%i2) 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]
```

See also function `quantile`.

mean_deviation (*list*) Function
mean_deviation (*list*) Function

The mean deviation, defined as

$$\frac{1}{n}\sum_{i=1}^{n}|x_i - \bar{x}|$$

Example:

```
(%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]
```

See also function `mean`.

median_deviation (*list*) Function
median_deviation (*matrix*) Function

The median deviation, defined as

$$\frac{1}{n}\sum_{i=1}^{n}|x_i - med|$$

where `med` is the median of *list*.

Example:

```
(%i1) load (descriptive)$
(%i2) 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]
```

See also function `mean`.

harmonic_mean (*list*) Function
harmonic_mean (*list*) Function

The harmonic mean, defined as

$$\frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

Example:

```
(%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]
```

See also functions `mean` and `geometric_mean`.

geometric_mean (*list*) Function
geometric_mean (*matrix*) Function

The geometric mean, defined as

$$\left(\prod_{i=1}^{n} x_i \right)^{\frac{1}{n}}$$

Example:

```
(%i1) load (descriptive)$
(%i2) 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]
```

See also functions `mean` and `harmonic_mean`.

kurtosis (*list*) Function
kurtosis (*matrix*) Function

The kurtosis coefficient, defined as

$$\frac{1}{ns^4} \sum_{i=1}^{n} (x_i - \bar{x})^4 - 3$$

Example:

```
(%i1) load (descriptive)$
(%i2) 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]
```

See also functions `mean`, `var` and `skewness`.

skewness (*list*) Function
skewness (*matrix*) Function

The skewness coefficient, defined as

$$\frac{1}{ns^3}\sum_{i=1}^{n}(x_i-\bar{x})^3$$

Example:

```
(%i1) load (descriptive)$
(%i2) 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]
```

See also functions `mean`, `var` and `kurtosis`.

pearson_skewness (*list*) Function
pearson_skewness (*matrix*) Function

Pearson's skewness coefficient, defined as

$$\frac{3\ (\bar{x}-med)}{s}$$

where *med* is the median of *list*.

Example:

```
(%i1) load (descriptive)$
(%i2) 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]
```

See also functions `mean`, `var` and `median`.

quartile_skewness (*list*) Function
quartile_skewness (*matrix*) Function

The quartile skewness coefficient, defined as

$$\frac{c_{\frac{3}{4}} - 2\,c_{\frac{1}{2}} + c_{\frac{1}{4}}}{c_{\frac{3}{4}} - c_{\frac{1}{4}}}$$

where c_p is the p-quantile of sample *list*.

Example:

```
(%i1) load (descriptive)$
(%i2) 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]
```

See also function `quantile`.

46.4 Definitions for specific multivariate descriptive statistics

cov (*matrix*) Function

The covariance matrix of the multivariate sample, defined as

$$S = \frac{1}{n}\sum_{j=1}^{n} \left(X_j - \bar{X}\right)\,\left(X_j - \bar{X}\right)'$$

where X_j is the j-th row of the sample matrix.

Example:

```
(%i1) load (descriptive)$
(%i2) load (numericalio)$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) fpprintprec : 7$  /* change precision for pretty output */
(%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 ]
```

See also function `cov1`.

cov1 (*matrix*) Function

The covariance matrix of the multivariate sample, defined as

$$\frac{1}{n-1}\sum_{j=1}^{n}\left(X_j-\bar{X}\right)\left(X_j-\bar{X}\right)'$$

where X_j is the j-th row of the sample matrix.

Example:

```
(%i1) load (descriptive)$
(%i2) load (numericalio)$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) fpprintprec : 7$ /* change precision for pretty output */
(%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 ]
```

See also function `cov`.

global_variances (*matrix*) Function
global_variances (*matrix, logical_value*) Function

Function `global_variances` returns a list of global variance measures:

- *total variance*: `trace(S_1)`,
- *mean variance*: `trace(S_1)/p`,
- *generalized variance*: `determinant(S_1)`,
- *generalized standard deviation*: `sqrt(determinant(S_1))`,
- *efective variance* `determinant(S_1)^(1/p)`, (defined in: Peña, D. (2002) *Análisis de datos multivariantes*; McGraw-Hill, Madrid.)
- *efective standard deviation*: `determinant(S_1)^(1/(2*p))`.

where p is the dimension of the multivariate random variable and S_1 the covariance matrix returned by `cov1`.

Example:

```
(%i1) load (descriptive)$
(%i2) 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]
```

Function `global_variances` has an optional logical argument: `global_variances(x,true)` tells Maxima that `x` is the data matrix, making the same as

`global_variances(x)`. On the other hand, `global_variances(x,false)` means that `x` is not the data matrix, but the covariance matrix, avoiding its recalculation,

```
(%i1) load (descriptive)$
(%i2) 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]
```

See also `cov` and `cov1`.

cor (*matrix*) Function
cor (*matrix*, *logical_value*) Function

The correlation matrix of the multivariate sample.

Example:

```
(%i1) load (descriptive)$
(%i2) 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    ]
```

Function `cor` has an optional logical argument: `cor(x,true)` tells Maxima that `x` is the data matrix, making the same as `cor(x)`. On the other hand, `cor(x,false)` means that `x` is not the data matrix, but the covariance matrix, avoiding its recalculation,

```
(%i1) load (descriptive)$
(%i2) 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    ]
```

See also `cov` and `cov1`.

list_correlations (*matrix*) Function
list_correlations (*matrix, logical_value*) Function

Function `list_correlations` returns a list of correlation measures:

- *precision matrix*: the inverse of the covariance matrix S_1,

$$S_1^{-1} = \left(s^{ij}\right)_{i,j=1,2,...,p}$$

- *multiple correlation vector*: $(R_1^2, R_2^2, ..., R_p^2)$, with

$$R_i^2 = 1 - \frac{1}{s^{ii}s_{ii}}$$

  being an indicator of the goodness of fit of the linear multivariate regression model on X_i when the rest of variables are used as regressors.

- *partial correlation matrix*: with element (i, j) being

$$r_{ij.rest} = -\frac{s^{ij}}{\sqrt{s^{ii}s^{jj}}}$$

Example:

```
(%i1) load (descriptive)$
(%i2) 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   ]
```

Function `list_correlations` also has an optional logical argument: `list_correlations(x,true)` tells Maxima that `x` is the data matrix, making the same as `list_correlations(x)`. On the other hand, `list_correlations(x,false)` means that `x` is not the data matrix, but the covariance matrix, avoiding its recalculation.

See also `cov` and `cov1`.

46.5 Definitions for statistical graphs

dataplot (*list*) Function
dataplot (*list, option_1, option_2, ...*) Function
dataplot (*matrix*) Function
dataplot (*matrix, option_1, option_2, ...*) Function

Funtion `dataplot` permits direct visualization of sample data, both univariate (*list*) and multivariate (*matrix*). Giving values to the following *options* some aspects of the plot can be controlled:

- `'outputdev`, default `"x"`, indicates the output device; correct values are `"x"`, `"eps"` and `"png"`, for the screen, postscript and png format files, respectively.
- `'maintitle`, default `""`, is the main title between double quotes.
- `'axisnames`, default `["x","y","z"]`, is a list with the names of axis `x`, `y` and `z`.
- `'joined`, default `false`, a logical value to select points in 2D to be joined or isolated.
- `'picturescales`, default `[1.0, 1.0]`, scaling factors for the size of the plot.
- `'threedim`, default `true`, tells Maxima whether to plot a three column matrix with a 3D diagram or a multivariate scatterplot. See examples bellow.
- `'axisrot`, default `[60, 30]`, changes the point of view when `'threedim` is set to `true` and data are stored in a three column matrix. The first number is the rotation angle of the *x*-axis, and the second number is the rotation angle of the *z*-axis, both measured in degrees.
- `'nclasses`, default `10`, is the number of classes for the histograms in the diagonal of multivariate scatterplots.
- `'pointstyle`, default `0`, is an integer to indicate how to display sample points.

For example, with the following input a simple plot of the first twenty digits of `%pi` is requested and the output stored in an eps file.

```
(%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 that one dimensional data are plotted as a time series. In the next case, same more data with different settings,

```
(%i1) load (descriptive)$
(%i2) load (numericalio)$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) dataplot (makelist (s1[k], k, 1, 50), 'maintitle = "First pi digits",
 'axisnames = ["digit order", "digit value"], 'pointstyle = 1,
 'joined = true)$
```

Function `dataplot` can be used to plot points in the plane. The next example is a scatterplot of the pairs of wind speeds corresponding to the first and fifth meteorological stations,

```
(%i1) load (descriptive)$
(%i2) load (numericalio)$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) dataplot (submatrix (s2, 2, 3, 4), 'pointstyle = 1,
 'maintitle = "Pairs of wind speeds measured in knots",
 'axisnames = ["Wind speed in A", "Wind speed in E"])$
```

If points are stored in a two column matrix, `dataplot` can plot them directly, but if they are formatted as a list of pairs, their must be transformed to a matrix as in the following example.

```
(%i1) load (descriptive)$
(%i2) x : [[-1, 2], [5, 7], [5, -3], [-6, -9], [-4, 6]]$
(%i3) dataplot (apply ('matrix, x), 'maintitle = "Points",
 'joined = true, 'axisnames = ["", ""], 'picturescales = [0.5, 1.0])$
```

Points in three dimensional space can be seen as a projection on the plane. In this example, plots of wind speeds corresponding to three meteorological stations are requested, first in a 3D plot and then in a multivariate scatterplot.

```
(%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 = "Pairs of wind speeds measured in knots",
 'axisnames = ["Station A", "Station B", "Station C"])$
(%i5) /* Multivariate scatterplot */ dataplot (submatrix (s2, 4, 5),
 'nclasses = 6, 'threedim = false)$
```

Note that in the last example, the number of classes in the histograms of the diagonal is set to 6, and that option `'threedim` is set to `false`.

For more than three dimensions only multivariate scatterplots are possible, as in

```
(%i1) load (descriptive)$
(%i2) load (numericalio)$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) dataplot (s2)$
```

histogram (*list*) Function
histogram (*list, option_1, option_2, ...*) Function
histogram (*one_column_matrix*) Function
histogram (*one_column_matrix, option_1, option_2, ...*) Function

This function plots an histogram. Sample data must be stored in a list of numbers or a one column matrix. Giving values to the following *options* some aspects of the plot can be controlled:

- `'outputdev`, default `"x"`, indicates the output device; correct values are `"x"`, `"eps"` and `"png"`, for the screen, postscript and png format files, respectively.
- `'maintitle`, default `""`, is the main title between double quotes.
- `'axisnames`, default `["x", "Fr."]`, is a list with the names of axis `x` and `y`.
- `'picturescales`, default `[1.0, 1.0]`, scaling factors for the size of the plot.
- `'nclasses`, default `10`, is the number of classes or bars.
- `'relbarwidth`, default `0.9`, a decimal number between 0 and 1 to control bars width.
- `'barcolor`, default `1`, an integer to indicate bars color.
- `'colorintensity`, default `1`, a decimal number between 0 and 1 to fix color intensity.

In the next two examples, histograms are requested for the first 100 digits of number `%pi` and for the wind speeds in the third meteorological station.

```
(%i1) load (descriptive)$
(%i2) load (numericalio)$
(%i3) s1 : read_list (file_search ("pidigits.data"))$
(%i4) histogram (s1, 'maintitle = "pi digits", 'axisnames = ["", "Absolute fre
 '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 that in the first case, `s1` is a list and in the second example, `col(s2,3)` is a matrix.

See also function `barsplot`.

barsplot (*list*) Function
barsplot (*list, option_1, option_2, ...*) Function
barsplot (*one_column_matrix*) Function
barsplot (*one_column_matrix, option_1, option_2, ...*) Function

Similar to `histogram` but for discrete, numeric or categorical, statistical variables. These are the options,

- `'outputdev`, default `"x"`, indicates the output device; correct values are `"x"`, `"eps"` and `"png"`, for the screen, postscript and png format files, respectively.
- `'maintitle`, default `""`, is the main title between double quotes.
- `'axisnames`, default `["x", "Fr."]`, is a list with the names of axis `x` and `y`.
- `'picturescales`, default `[1.0, 1.0]`, scaling factors for the size of the plot.
- `'relbarwidth`, default `0.9`, a decimal number between 0 and 1 to control bars width.

- 'barcolor, default 1, an integer to indicate bars color.
- 'colorintensity, default 1, a decimal number between 0 and 1 to fix color intensity.

This example plots the barchart for groups A and B of patients in sample s3,

```
(%i1) load (descriptive)$
(%i2) load (numericalio)$
(%i3) s3 : read_matrix (file_search ("biomed.data"))$
(%i4) barsplot (col (s3, 1), 'maintitle = "Groups of patients",
 'axisnames = ["Group", "# of individuals"], 'colorintensity = 0.2)$
```

The first column in sample s3 stores the categorical values A and B, also known sometimes as factors. On the other hand, the positive integer numbers in the second column are ages, in years, which is a discrete variable, so we can plot the absolute frequencies for these values,

```
(%i1) load (descriptive)$
(%i2) load (numericalio)$
(%i3) s3 : read_matrix (file_search ("biomed.data"))$
(%i4) barsplot (col (s3, 2), 'maintitle = "Ages",
 'axisnames = ["Years", "# of individuals"], 'colorintensity = 0.2,
 'relbarwidth = 0.6)$
```

See also function histogram.

boxplot (*data*) Function
boxplot (*data, option_1, option_2, ...*) Function

This function plots box diagrams. Argument *data* can be a list, which is not of great interest, since these diagrams are mainly used for comparing different samples, or a matrix, so it is possible to compare two or more components of a multivariate statistical variable. But it is also allowed *data* to be a list of samples with possible different sample sizes, in fact this is the only function in package descriptive that admits this type of data structure. See example bellow. These are the options,

- 'outputdev, default "x", indicates the output device; correct values are "x", "eps" and "png", for the screen, postscript and png format files, respectively.
- 'maintitle, default "", is the main title between double quotes.
- 'axisnames, default ["sample", "y"], is a list with the names of axis x and y.
- 'picturescales, default [1.0, 1.0], scaling factors for the size of the plot.

Examples:

```
(%i1) load (descriptive)$
(%i2) load (numericalio)$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) boxplot (s2, 'maintitle = "Windspeed in knots",
 'axisnames = ["Seasons", ""])$
(%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)$
```

47 diag

47.1 Definitions for diag

diag (*lm*) Function

Constructs a square matrix with the matrices of *lm* in the diagonal. *lm* is a list of matrices or scalars.

Example:

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

(%i2) a1:matrix([1,2,3],[0,4,5],[0,0,6])$

(%i3) a2:matrix([1,1],[1,0])$

(%i4) diag([a1,x,a2]);
                    [ 1  2  3  0  0  0 ]
                    [                  ]
                    [ 0  4  5  0  0  0 ]
                    [                  ]
                    [ 0  0  6  0  0  0 ]
(%o4)               [                  ]
                    [ 0  0  0  x  0  0 ]
                    [                  ]
                    [ 0  0  0  0  1  1 ]
                    [                  ]
                    [ 0  0  0  0  1  0 ]
```

To use this function write first `load("diag")`.

JF (*lambda*,*n*) Function

Returns the Jordan cell of order *n* with eigenvalue *lambda*.

Example:

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

(%i2) JF(2,5);
                     [ 2  1  0  0  0 ]
                     [               ]
                     [ 0  2  1  0  0 ]
                     [               ]
(%o2)                [ 0  0  2  1  0 ]
                     [               ]
                     [ 0  0  0  2  1 ]
                     [               ]
                     [ 0  0  0  0  2 ]
(%i3) JF(3,2);
                          [ 3  1 ]
```

```
(%o3)                          [      ]
                               [ 0  3 ]
```

To use this function write first `load("diag")`.

jordan (*mat*) Function

Returns the Jordan form of matrix *mat*, but codified in a Maxima list. To get the corresponding matrix, call function `dispJordan` using as argument the output of `JF`.

Example:

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

(%i3) a:matrix([2,0,0,0,0,0,0,0],
               [1,2,0,0,0,0,0,0],
               [-4,1,2,0,0,0,0,0],
               [2,0,0,2,0,0,0,0],
               [-7,2,0,0,2,0,0,0],
               [9,0,-2,0,1,2,0,0],
               [-34,7,1,-2,-1,1,2,0],
               [145,-17,-16,3,9,-2,0,3])$

(%i34) jordan(a);
(%o4)             [[2, 3, 3, 1], [3, 1]]
(%i5) dispJordan(%);
               [ 2  1  0  0  0  0  0  0 ]
               [                        ]
               [ 0  2  1  0  0  0  0  0 ]
               [                        ]
               [ 0  0  2  0  0  0  0  0 ]
               [                        ]
               [ 0  0  0  2  1  0  0  0 ]
(%o5)          [                        ]
               [ 0  0  0  0  2  1  0  0 ]
               [                        ]
               [ 0  0  0  0  0  2  0  0 ]
               [                        ]
               [ 0  0  0  0  0  0  2  0 ]
               [                        ]
               [ 0  0  0  0  0  0  0  3 ]
```

To use this function write first `load("diag")`. See also `dispJordan` and `minimalPoly`.

dispJordan (*l*) Function

Returns the Jordan matrix associated to the codification given by the Maxima list *l*, which is the output given by function `jordan`.

Example:

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

(%i2) b1:matrix([0,0,1,1,1],
                [0,0,0,1,1],
```

```
                [0,0,0,0,1],
                [0,0,0,0,0],
                [0,0,0,0,0])$

(%i3) jordan(b1);
(%o3)                  [[0, 3, 2]]
(%i4) dispJordan(%);
                    [ 0  1  0  0  0 ]
                    [               ]
                    [ 0  0  1  0  0 ]
                    [               ]
(%o4)               [ 0  0  0  0  0 ]
                    [               ]
                    [ 0  0  0  0  1 ]
                    [               ]
                    [ 0  0  0  0  0 ]
```

To use this function write first `load("diag")`. See also `jordan` and `minimalPoly`.

minimalPoly (*l*) Function

Returns the minimal polynomial associated to the codification given by the Maxima list *l*, which is the output given by function `jordan`.

Example:

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

(%i2) a:matrix([2,1,2,0],
               [-2,2,1,2],
               [-2,-1,-1,1],
               [3,1,2,-1])$

(%i3) jordan(a);
(%o3)               [[- 1, 1], [1, 3]]
(%i4) minimalPoly(%);
                              3
(%o4)                (x - 1)  (x + 1)
```

To use this function write first `load("diag")`. See also `jordan` and `dispJordan`.

ModeMatrix (*A*,*l*) Function

Returns the matrix M such that $(Mm1).A.M = J$, where J is the Jordan form of A. The Maxima list *l* is the codified form of the Jordan form as returned by function `jordan`.

Example:

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

(%i2) a:matrix([2,1,2,0],
          [-2,2,1,2],
          [-2,-1,-1,1],
          [3,1,2,-1])$
```

```
(%i3) jordan(a);
(%o3)                [[- 1, 1], [1, 3]]
(%i4) M: ModeMatrix(a,%);
                    [  1    - 1   1   1 ]
                    [                   ]
                    [   1               ]
                    [ - -   - 1   0   0 ]
                    [   9               ]
                    [                   ]
(%o4)               [   13              ]
                    [ - --   1   - 1  0 ]
                    [   9               ]
                    [                   ]
                    [  17               ]
                    [  --   - 1   1   1 ]
                    [  9                ]
(%i5) is(  (M^^-1).a.M = dispJordan(%o3)  );
(%o5)                        true
```

Note that `dispJordan(%o3)` is the Jordan form of matrix `a`.

To use this function write first `load("diag")`. See also `jordan` and `dispJordan`.

mat_function (*f*,*mat*) Function

Returns $f(mat)$, where f is an analytic function and *mat* a matrix. This computation is based on Cauchy's integral formula, which states that if `f(x)` is analytic and

```
mat=diag([JF(m1,n1),...,JF(mk,nk)]),
```

then

```
f(mat)=ModeMatrix*diag([f(JF(m1,n1)),...,f(JF(mk,nk))])*ModeMatrix^^(-1)
```

Note that there are about 6 or 8 other methods for this calculation.

Some examples follow.

Example 1:

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

(%i2) b2:matrix([0,1,0], [0,0,1], [-1,-3,-3])$

(%i3) mat_function(exp,t*b2);
                2   - t
               t  %e          - t     - t
(%o3) matrix([-------- + t %e    + %e   ,
                 2
        - t     - t                            - t
 2    %e      %e         - t           - t   %e
t  (- ----- - ----- + %e   ) + t (2 %e    - -----)
       t       2                              t
              t
                          - t           - t     - t
       - t       - t   %e         2  %e      %e
```

```
 + 2 %e   , t (%e    - -----) + t  (----- - -----)
                          t            2       t
                2   - t           - t     - t
    - t      t  %e        2    %e      %e        - t
 + %e   ], [- --------, - t  (- ----- - ----- + %e   ),
                 2                t       2
                                           t
       - t     - t      2   - t
   2  %e      %e        t  %e          - t
- t  (----- - -----)], [-------- - t %e   ,
       2       t          2
       - t     - t                              - t
 2    %e      %e        - t           - t   %e
t  (- ----- - ----- + %e   ) - t (2 %e    - -----),
       t       2                               t
              t
      - t     - t                 - t
 2  %e      %e            - t   %e
t  (----- - -----) - t (%e    - -----)])
      2       t                   t
(%i4) ratsimp(%);
                [   2              - t ]
                [ (t  + 2 t + 2) %e    ]
                [ -------------------- ]
                [          2           ]
                [                      ]
                [         2   - t      ]
(%o4)  Col 1 = [        t  %e         ]
                [      - --------      ]
                [           2          ]
                [                      ]
                [     2          - t   ]
                [   (t  - 2 t) %e      ]
                [   ----------------   ]
                [          2           ]
         [      2        - t    ]
         [    (t  + t) %e       ]
         [                      ]
 Col 2 = [     2            - t ]
         [ - (t  - t - 1) %e    ]
         [                      ]
         [     2          - t   ]
         [   (t  - 3 t) %e      ]
         [        2   - t       ]
         [       t  %e          ]
         [       --------       ]
         [          2           ]
         [                      ]
         [      2         - t  ]
```

```
                  [    (t  - 2 t) %e     ]
                  [  - ---------------   ]
                  [           2          ]
                  [                      ]
                  [   2              - t ]
                  [ (t  - 4 t + 2) %e    ]
                  [ -------------------- ]
                  [          2           ]
```

Example 2:

```
(%i5) b1:matrix([0,0,1,1,1],
                [0,0,0,1,1],
                [0,0,0,0,1],
                [0,0,0,0,0],
                [0,0,0,0,0])$

(%i6) mat_function(exp,t*b1);
                  [              2     ]
                  [             t      ]
                  [ 1  0  t  t  -- + t ]
                  [             2      ]
                  [                    ]
(%o6)             [ 0  1  0  t    t    ]
                  [                    ]
                  [ 0  0  1  0    t    ]
                  [                    ]
                  [ 0  0  0  1    0    ]
                  [                    ]
                  [ 0  0  0  0    1    ]
(%i7) minimalPoly(jordan(b1));
                           3
(%o7)                     x
(%i8) ident(5)+t*b1+1/2*(t^2)*b1^^2;
                  [              2     ]
                  [             t      ]
                  [ 1  0  t  t  -- + t ]
                  [             2      ]
                  [                    ]
(%o8)             [ 0  1  0  t    t    ]
                  [                    ]
                  [ 0  0  1  0    t    ]
                  [                    ]
                  [ 0  0  0  1    0    ]
                  [                    ]
                  [ 0  0  0  0    1    ]
(%i9) mat_function(exp,%i*t*b1);
              [                          2 ]
              [                         t  ]
              [ 1  0  %i t  %i t  %i t - -- ]
```

```
                [                          2  ]
                [                             ]
(%o9)           [ 0  1   0    %i t    %i t    ]
                [                             ]
                [ 0  0   1     0      %i t    ]
                [                             ]
                [ 0  0   0     1        0     ]
                [                             ]
                [ 0  0   0     0        1     ]
(%i10) mat_function(cos,t*b1)+%i*mat_function(sin,t*b1);
                 [                            2 ]
                 [                           t  ]
                 [ 1  0  %i t  %i t  %i t - -- ]
                 [                           2  ]
                 [                              ]
(%o10)           [ 0  1   0    %i t    %i t    ]
                 [                             ]
                 [ 0  0   1     0      %i t    ]
                 [                             ]
                 [ 0  0   0     1        0     ]
                 [                             ]
                 [ 0  0   0     0        1     ]
```

Example 3:

```
(%i11) a1:matrix([2,1,0,0,0,0],
                 [-1,4,0,0,0,0],
                 [-1,1,2,1,0,0],
                 [-1,1,-1,4,0,0],
                 [-1,1,-1,1,3,0],
                 [-1,1,-1,1,1,2])$

(%i12) fpow(x):=block([k],declare(k,integer),x^k)$

(%i13) mat_function(fpow,a1);
                [  k      k - 1 ]          [      k - 1    ]
                [ 3  - k 3      ]          [   k 3         ]
                [               ]          [               ]
                [       k - 1   ]          [  k      k - 1 ]
                [  - k 3        ]          [ 3  + k 3      ]
                [               ]          [               ]
                [       k - 1   ]          [      k - 1    ]
                [  - k 3        ]          [   k 3         ]
(%o13)  Col 1 = [               ] Col 2 = [               ]
                [       k - 1   ]          [      k - 1    ]
                [  - k 3        ]          [   k 3         ]
                [               ]          [               ]
                [       k - 1   ]          [      k - 1    ]
                [  - k 3        ]          [   k 3         ]
                [               ]          [               ]
                [       k - 1   ]          [      k - 1    ]
```

```
                      [  - k 3       ]         [   k 3        ]
             [       0       ]         [       0       ]
             [               ]         [               ]
             [       0       ]         [       0       ]
             [               ]         [               ]
             [  k      k - 1 ]         [      k - 1    ]
             [ 3  - k 3      ]         [   k 3         ]
             [               ]         [               ]
Col 3 =      [       k - 1   ] Col 4 = [  k      k - 1 ]
             [  - k 3        ]         [ 3  + k 3      ]
             [               ]         [               ]
             [       k - 1   ]         [      k - 1    ]
             [  - k 3        ]         [   k 3         ]
             [               ]         [               ]
             [       k - 1   ]         [      k - 1    ]
             [  - k 3        ]         [   k 3         ]
             [    0    ]
             [         ]         [ 0  ]
             [    0    ]         [    ]
             [         ]         [ 0  ]
             [    0    ]         [    ]
             [         ]         [ 0  ]
Col 5 =      [    0    ] Col 6 = [    ]
             [         ]         [ 0  ]
             [    k    ]         [    ]
             [   3     ]         [ 0  ]
             [         ]         [    ]
             [  k    k ]         [  k ]
             [ 3  - 2  ]         [ 2  ]
```

To use this function write first `load("diag")`.

48 distrib

48.1 Introduction to distrib

Package `distrib` contains a set of functions for making probability computations on both discrete and continuous univariate models.

What follows is a short reminder of basic probabilistic related definitions.

Let $f(x)$ be the *density function* of an absolute continuous random variable X. The *distribution function* is defined as

$$F(x) = \int_{-\infty}^{x} f(u)\ du$$

which equals the probability *Pr(X <= x)*.

The *mean* value is a localization parameter and is defined as

$$E[X] = \int_{-\infty}^{\infty} x\, f(x)\ dx$$

The *variance* is a measure of variation,

$$V[X] = \int_{-\infty}^{\infty} f(x)\ (x - E[X])^2\ dx$$

which is a positive real number. The square root of the variance is the *standard deviation*, $D[X] = sqrt(V[X])$, and it is another measure of variation.

The *skewness coefficient* is a measure of non-symmetry,

$$SK[X] = \frac{\int_{-\infty}^{\infty} f(x)\ (x - E[X])^3\ dx}{D[X]^3}$$

And the *kurtosis coefficient* measures the peakedness of the distribution,

$$KU[X] = \frac{\int_{-\infty}^{\infty} f(x)\ (x - E[X])^4\ dx}{D[X]^4} - 3$$

If X is gaussian, $KU[X] = 0$. In fact, both skewness and kurtosis are shape parameters used to measure the non–gaussianity of a distribution.

If the random variable X is discrete, the density, or *probability*, function $f(x)$ takes positive values within certain countable set of numbers x_i, and zero elsewhere. In this case, the distribution function is

$$F(x) = \sum_{x_i \le x} f(x_i)$$

The mean, variance, standard deviation, skewness coefficient and kurtosis coefficient take the form

$$E[X] = \sum_{x_i} x_i f(x_i),$$

$$V\left[X\right]=\sum_{x_i} f\left(x_i\right)\left(x_i-E\left[X\right]\right)^2,$$

$$D\left[X\right]=\sqrt{V\left[X\right]},$$

$$SK\left[X\right]=\frac{\sum_{x_i} f\left(x\right)\ \left(x-E\left[X\right]\right)^3\ dx}{D\left[X\right]^3}$$

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.

Package `distrib` includes functions for simulating random variates. Some of these functions make use of optional variables indicating the algorithm to be used. The general inverse method (based on the fact that if u is an uniform random number in $(0,1)$, then *F^(-1)(u)* is a random variate with distribution F) is implemented in most cases; this is a suboptimal method in terms of timing, but useful for comparing with other algorithms. In this example, the performance of algorithms `ahrens_cheng` and `inverse` for simulating chi-square variates are compared by means of their histograms:

```
(%i1) load(descriptive)$
(%i2) showtime:true$
Evaluation took 0.00 seconds (0.00 elapsed) using 80 bytes.
(%i3) rchi2_algorithm: 'ahrens_cheng$ histogram(rchi2(10,500))$
Evaluation took 0.00 seconds (0.00 elapsed) using 80 bytes.
Evaluation took 0.70 seconds (0.77 elapsed) using 5.517 MB.
(%i5) rchi2_algorithm: 'inverse$ histogram(rchi2(10,500))$
Evaluation took 0.00 seconds (0.00 elapsed) using 80 bytes.
Evaluation took 10.37 seconds (10.45 elapsed) using 321.278 MB.
```

In order to make visual comparisons among algorithms for a discrete variate, function `barsplot` of the `descriptive` package should be used.

Note that some work remains to be done, since these simulating functions are not yet checked by more rigurous goodness of fit tests.

Please, consult an introductory manual on probability and statistics for more information about all this mathematical stuff.

There is a naming convention in package `distrib`. Every function name has two parts, the first one makes reference to the function or parameter we want to calculate,

```
Functions:
   Density function            (den*)
   Distribution function       (dis*)
   Quantile                    (q*)
   Mean                        (mean*)
   Variance                    (var*)
   Standard deviation          (std*)
   Skewness coefficient        (skw*)
   Kurtosis coefficient        (kur*)
```

```
   Random variate              (r*)
```

The second part is an explicit reference to the probabilistic model,

```
Continuous distributions:
   Normal              (*normal)
   Student             (*student)
   Chi^2               (*chi2)
   F                   (*f)
   Exponential         (*exp)
   Lognormal           (*logn)
   Gamma               (*gamma)
   Beta                (*beta)
   Continuous uniform  (*contu)
   Logistic            (*log)
   Pareto              (*pareto)
   Weibull             (*weibull)
   Rayleigh            (*rayleigh)
   Laplace             (*laplace)
   Cauchy              (*cauchy)
   Gumbel              (*gumbel)

Discrete distributions:
   Binomial             (*binomial)
   Poisson              (*poisson)
   Bernoulli            (*bernoulli)
   Geometric            (*geo)
   Discrete uniform     (*discu)
   Hypergeometric       (*hypergeo)
   Negative binomial    (*negbinom)
```

For example, `denstudent(x,n)` is the density function of the Student distribution with n degrees of freedom, `stdpareto(a,b)` is the standard deviation of the Pareto distribution with parameters a and b and `kurpoisson(m)` is the kurtosis coefficient of the Poisson distribution with mean m.

In order to make use of package `distrib` you need first to load it by typing

```
(%i1) load(distrib)$
```

For comments, bugs or suggestions, please contact the author at *'mario AT edu DOT xunta DOT es'*.

48.2 Definitions for continuous distributions

dennormal (*x,m,s*) Function

Returns the value at x of the density function of a normal random variable $N(m, s)$, with $s > 0$.

disnormal (*x,m,s*) Function

Returns the value at x of the distribution function of a normal random variable $N(m, s)$, with $s > 0$. This function is defined in terms of Maxima's built-in error function `erf`.

```
(%i1) assume(s>0)$ disnormal(x,m,s);
                            x - m
                         erf(---------)
                             sqrt(2) s    1
(%o2)                    -------------- + -
                               2          2
```

See also `erf`.

qnormal (*q*,*m*,*s*) Function

Returns the *q*-quantile of a normal random variable $N(m, s)$, with $s > 0$; in other words, this is the inverse of `disnormal`. Argument q must be an element of $[0, 1]$.

meannormal (*m*,*s*) Function

Returns the mean of a normal random variable $N(m, s)$, with $s > 0$, namely m.

varnormal (*m*,*s*) Function

Returns the variance of a normal random variable $N(m, s)$, with $s > 0$, namely *s^2*.

stdnormal (*m*,*s*) Function

Returns the standard deviation of a normal random variable $N(m, s)$, with $s > 0$, namely s.

skwnormal (*m*,*s*) Function

Returns the skewness coefficient of a normal random variable $N(m, s)$, with $s > 0$, which is always equal to 0.

kurnormal (*m*,*s*) Function

Returns the kurtosis coefficient of a normal random variable $N(m, s)$, with $s > 0$, which is always equal to 0.

rnormal_algorithm Option variable

Default value: `box_mueller`

This is the selected algorithm for simulating random normal variates. Implemented algorithms are `box_mueller` and `inverse`:

- `box_mueller`, based on algorithm described in Knuth, D.E. (1981) *Seminumerical Algorithms. The Art of Computer Programming.*. Addison-Wesley.
- `inverse`, based on the general inverse method.

See also `rnormal`.

rnormal (*m*,*s*) Function
rnormal (*m*,*s*,*n*) Function

Returns a normal random variate $N(m, s)$, with $s > 0$. Calling `rnormal` with a third argument n, a random sample of size n will be simulated.

There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `rnormal_algorithm`, which defaults to `box_mueller`.

There is also a built-in Maxima function for simulating random normal variates based on the so called Marsaglia's Ziggurat method.

See also `rnormal_algorithm` and `gauss`.

denstudent (*x*,*n*) Function

Returns the value at x of the density function of a Student random variable $t(n)$, with $n > 0$.

disstudent (*x*,*n*) Function

Returns the value at x of the distribution function of a Student random variable $t(n)$, with $n > 0$. This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) disstudent(1/2, 7/3);
                                  1  7
(%o1)                  disstudent(-, -)
                                  2  3
(%i2) %,numer;
(%o2)                 .6698450596140417
```

qstudent (*q*,*n*) Function

Returns the q-quantile of a Student random variable $t(n)$, with $n > 0$; in other words, this is the inverse of `disstudent`. Argument q must be an element of $[0, 1]$.

meanstudent (*n*) Function

Returns the mean of a Student random variable $t(n)$, with $n > 0$, which is always equal to 0.

varstudent (*n*) Function

Returns the variance of a Student random variable $t(n)$, with $n > 2$.

```
(%i1) assume(n>2)$  varstudent(n);
                              n
(%o2)                       -----
                            n - 2
```

stdstudent (*n*) Function

Returns the standard deviation of a Student random variable $t(n)$, with $n > 2$.

skwstudent (*n*) Function

Returns the skewness coefficient of a Student random variable $t(n)$, with $n > 3$, which is always equal to 0.

kurstudent (*n*) Function

Returns the kurtosis coefficient of a Student random variable $t(n)$, with $n > 4$.

rstudent_algorithm Option variable

Default value: `ratio`

This is the selected algorithm for simulating random Student variates. Implemented algorithms are `inverse` and `ratio`:

- `inverse`, based on the general inverse method.
- `ratio`, based on the fact that if Z is a normal random variable $N(0,1)$ and S^2 is chi square random variable with n degrees of freedom, $Chi^2(n)$, then

$$X = \frac{Z}{\sqrt{\frac{S^2}{n}}}$$

  is a Student random variable with n degrees of freedom, $t(n)$.

See also `rstudent`.

rstudent (*n*) Function

rstudent (*n*,*m*) Function

Returns a Student random variate $t(n)$, with $n > 0$. Calling `rstudent` with a second argument m, a random sample of size m will be simulated.

There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `rstudent_algorithm`, which defaults to `ratio`.

See also `rstudent_algorithm`.

denchi2 (*x*,*n*) Function

Returns the value at x of the density function of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a nominal form based on the gamma density is returned.

```
(%i1) denchi2(x,n);
                                  n
(%o1)                  dengamma(x, -, 2)
                                  2
(%i2) assume(x>0, n>0)$  denchi2(x,n);
                        n/2 - 1   - x/2
                       x        %e
(%o2)                  ----------------
                         n/2       n
                        2    gamma(-)
                                   2
```

dischi2 (*x*,*n*) Function

Returns the value at x of the distribution function of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression based on the gamma distribution, since the $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$.

```
(%i1) dischi2(3,4);
(%o1)                disgamma(3, 2, 2)
(%i2) dischi2(3,4),numer;
(%o2)                .4421745996289249
```

qchi2 (*q*,*n*) Function

Returns the *q*-quantile of a Chi-square random variable $Chi^2(n)$, with $n > 0$; in other words, this is the inverse of `dischi2`. Argument q must be an element of $[0, 1]$.

This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression based on the gamma quantile function, since the $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$.

```
(%i1) qchi2(0.99,9);
(%o1)                21.66599433346194
(%i2) qchi2(0.99,n);
                                    n
(%o2)                qgamma(0.99, -, 2)
                                    2
```

meanchi2 (*n*) Function

Returns the mean of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a nominal form based on the gamma mean is returned.

```
(%i1) meanchi2(n);
                                    n
(%o1)                   meangamma(-, 2)
                                    2
(%i2) assume(n>0)$ meanchi2(n);
(%o3)                           n
```

varchi2 (*n*) Function

Returns the variance of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a nominal form based on the gamma variance is returned.

```
(%i1) varchi2(n);
                                  n
(%o1)                 vargamma(-, 2)
                                  2
(%i2) assume(n>0)$ varchi2(n);
(%o3)                         2 n
```

stdchi2 (*n*) Function

Returns the standard deviation of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a nominal form based on the gamma standard deviation is returned.

```
(%i1) stdchi2(n);
                                n
(%o1)                  stdgamma(-, 2)
                                2
(%i2) assume(n>0)$ stdchi2(n);
(%o3)                 sqrt(2) sqrt(n)
```

skwchi2 (n) Function

Returns the skewness coefficient of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a nominal form based on the gamma skewness coefficient is returned.

```
(%i1) skwchi2(n);
                                n
(%o1)                  skwgamma(-, 2)
                                2
(%i2) assume(n>0)$ skwchi2(n);
                          2 sqrt(2)
(%o3)                     ---------
                           sqrt(n)
```

kurchi2 (n) Function

Returns the kurtosis coefficient of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a nominal form based on the gamma kurtosis coefficient is returned.

```
(%i1) kurchi2(n);
                                n
(%o1)                  kurgamma(-, 2)
                                2
(%i2) assume(n>0)$ kurchi2(n);
                              12
(%o3)                         --
                              n
```

rchi2_algorithm Option variable

Default value: `ahrens_cheng`

This is the selected algorithm for simulating random Chi-square variates. Implemented algorithms are `ahrens_cheng` and `inverse`:

- `ahrens_cheng`, based on the random simulation of gamma variates. See `rgamma_algorithm` for details.
- `inverse`, based on the general inverse method.

See also `rchi2`.

rchi2 (*n*) Function
rchi2 (*n,m*) Function

Returns a Chi-square random variate $Chi^2(n)$, with $n > 0$. Calling `rchi2` with a second argument m, a random sample of size m will be simulated.

There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `rchi2_algorithm`, which defaults to `ahrens_cheng`.

See also `rchi2_algorithm`.

denf (*x,m,n*) Function

Returns the value at x of the density function of a F random variable $F(m, n)$, with $m, n > 0$.

disf (*x,m,n*) Function

Returns the value at x of the distribution function of a F random variable $F(m, n)$, with $m, n > 0$. This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) disf(2,3,9/4);
                                  9
(%o1)                   disf(2, 3, -)
                                  4
(%i2) %,numer;
(%o2)                 0.66756728179008
```

qf (*q,m,n*) Function

Returns the q-quantile of a F random variable $F(m, n)$, with $m, n > 0$; in other words, this is the inverse of `disf`. Argument q must be an element of $[0, 1]$.

This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) qf(2/5,sqrt(3),5);
                             2
(%o1)                 qf(-, sqrt(3), 5)
                             5
(%i2) %,numer;
(%o2)                 0.518947838573693
```

meanf (*m,n*) Function

Returns the mean of a F random variable $F(m, n)$, with $m > 0, n > 2$.

varf (*m,n*) Function

Returns the variance of a F random variable $F(m, n)$, with $m > 0, n > 4$.

stdf (*m,n*) Function

Returns the standard deviation of a F random variable $F(m, n)$, with $m > 0, n > 4$.

skwf (*m,n*) Function

Returns the skewness coefficient of a F random variable $F(m, n)$, with $m > 0, n > 6$.

kurf (*m*,*n*) Function

Returns the kurtosis coefficient of a F random variable $F(m,n)$, with $m > 0, n > 8$.

rf_algorithm Option variable

Default value: `inverse`

This is the selected algorithm for simulating random F variates. Implemented algorithms are `ratio` and `inverse`:

- `ratio`, based on the fact that if X is a $Chi^2(m)$ random variable and Y is a $Chi^2(n)$ random variable, then

$$F = \frac{nX}{mY}$$

  is a F random variable with m and n degrees of freedom, $F(m,n)$.
- `inverse`, based on the general inverse method.

See also `rf`.

rf (*m*,*n*) Function

rf (*m*,*n*,*k*) Function

Returns a F random variate $F(m,n)$, with $m, n > 0$. Calling `rf` with a third argument k, a random sample of size k will be simulated.

There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `rf_algorithm`, which defaults to `inverse`.

See also `rf_algorithm`.

denexp (*x*,*m*) Function

Returns the value at x of the density function of an exponential random variable $Exp(m)$, with $m > 0$.

The $Exp(m)$ random variable is equivalent to the Weibull $Wei(1, 1/m)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull density is returned.

```
(%i1) denexp(x,m);
                                      1
(%o1)              denweibull(x, 1, -)
                                      m
(%i2) assume(x>0,m>0)$  denexp(x,m);
                                - m x
(%o3)                     m %e
```

disexp (*x*,*m*) Function

Returns the value at x of the distribution function of an exponential random variable $Exp(m)$, with $m > 0$.

The $Exp(m)$ random variable is equivalent to the Weibull $Wei(1, 1/m)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull distribution is returned.

```
(%i1) disexp(x,m);
                                        1
(%o1)                disisweibull(x, 1, -)
                                        m
(%i2) assume(x>0,m>0)$  disexp(x,m);
                                  - m x
(%o3)                     1 - %e
```

qexp (*q*,*m*) Function

Returns the *q*-quantile of an exponential random variable $Exp(m)$, with $m > 0$; in other words, this is the inverse of `disexp`. Argument q must be an element of $[0, 1]$.

The $Exp(m)$ random variable is equivalent to the Weibull $Wei(1, 1/m)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull quantile is returned.

```
(%i51) qexp(0.56,5);
(%o1)                .1641961104139661
(%i2) qexp(0.56,m);
                                         1
(%o2)                qweibull(0.56, 1, -)
                                         m
```

meanexp (*m*) Function

Returns the mean of an exponential random variable $Exp(m)$, with $m > 0$.

The $Exp(m)$ random variable is equivalent to the Weibull $Wei(1, 1/m)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull mean is returned.

```
(%i1) meanexp(m);
                                       1
(%o1)                meanweibull(1, -)
                                       m
(%i2) assume(m>0)$  meanexp(m);
                            1
(%o3)                       -
                            m
```

varexp (*m*) Function

Returns the variance of an exponential random variable $Exp(m)$, with $m > 0$.

The $Exp(m)$ random variable is equivalent to the Weibull $Wei(1, 1/m)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull variance is returned.

```
(%i1) varexp(m);
                                       1
(%o2)                 varweibull(1, -)
                                       m
(%i3) assume(m>0)$  varexp(m);
                            1
(%o4)                       --
```

```
 2
m
```

stdexp (m) Function

Returns the standard deviation of an exponential random variable $Exp(m)$, with $m > 0$.

The $Exp(m)$ random variable is equivalent to the Weibull $Wei(1, 1/m)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull standard deviation is returned.

```
(%i1) stdexp(m);
                                    1
(%o1)                stdweibull(1, -)
                                    m
(%i2) assume(m>0)$  stdexp(m);
                            1
(%o3)                       -
                            m
```

skwexp (m) Function

Returns the skewness coefficient of an exponential random variable $Exp(m)$, with $m > 0$.

The $Exp(m)$ random variable is equivalent to the Weibull $Wei(1, 1/m)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull skewness coefficient is returned.

```
(%i1) skwexp(m);
                                    1
(%o1)                skwweibull(1, -)
                                    m
(%i2) assume(m>0)$  skwexp(m);
(%o3)                       2
```

kurexp (m) Function

Returns the kurtosis coefficient of an exponential random variable $Exp(m)$, with $m > 0$.

The $Exp(m)$ random variable is equivalent to the Weibull $Wei(1, 1/m)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull kurtosis coefficient is returned.

```
(%i1) kurexp(m);
                                    1
(%o1)                kurweibull(1, -)
                                    m
(%i2) assume(m>0)$  kurexp(m);
(%o3)                       6
```

rexp_algorithm Option variable

Default value: `inverse`

This is the selected algorithm for simulating random exponential variates. Implemented algorithms are `inverse`, `ahrens_cheng` and `ahrens_dieter`

- `inverse`, based on the general inverse method.
- `ahrens_cheng`, based on the fact that the $Exp(m)$ random variable is equivalent to the $Gamma(1, 1/m)$. See `rgamma_algorithm` for details.
- `ahrens_dieter`, based on algorithm described in Ahrens, J.H. and Dieter, U. (1972) *Computer methods for sampling from the exponential and normal distributions.*. Comm, ACM, 15, Oct., 873-882.

See also `rexp`.

rexp (*m*) Function
rexp (*m,k*) Function

Returns an exponential random variate $Exp(m)$, with $m > 0$. Calling `rexp2` with a second argument k, a random sample of size k will be simulated.

There are three algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `rexp_algorithm`, which defaults to `inverse`.

See also `rexp_algorithm`.

denlogn (*x,m,s*) Function

Returns the value at x of the density function of a log-normal random variable $log - N(m, s)$, with $s > 0$.

dislogn (*x,m,s*) Function

Returns the value at x of the distribution function of a log-normal random variable $log - N(m, s)$, with $s > 0$. This function is defined in terms of Maxima's built-in error function `erf`.

```
(%i1) assume(s>0)$  dislogn(x,m,s);
                        log(x) - m
                    erf(----------)
                        sqrt(2) s     1
(%o2)               --------------- + -
                           2          2
```

See also `erf`.

qlogn (*q,m,s*) Function

Returns the q-quantile of a log-normal random variable $log - N(m, s)$, with $s > 0$; in other words, this is the inverse of `dislogn`. Argument q must be an element of $[0, 1]$.

meanlogn (*m,s*) Function

Returns the mean of a log-normal random variable $log - N(m, s)$, with $s > 0$.

varlogn (*m,s*) Function

Returns the variance of a log-normal random variable $log - N(m, s)$, with $s > 0$.

stdlogn (*m*,*s*) Function

Returns the standard deviation of a log-normal random variable $log - N(m, s)$, with $s > 0$.

skwlogn (*m*,*s*) Function

Returns the skewness coefficient of a log-normal random variable $log - N(m, s)$, with $s > 0$.

kurlogn (*m*,*s*) Function

Returns the kurtosis coefficient of a log-normal random variable $log - N(m, s)$, with $s > 0$.

rlogn (*m*,*s*) Function
rlogn (*m*,*s*,*n*) Function

Returns a log-normal random variate $log - N(m, s)$, with $s > 0$. Calling `rlogn` with a third argument n, a random sample of size n will be simulated.

Log-normal variates are simulated by means of random normal variates. There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `rnormal_algorithm`, which defaults to `box_mueller`.

See also `rnormal_algorithm`.

dengamma (*x*,*a*,*b*) Function

Returns the value at x of the density function of a gamma random variable $Gamma(a, b)$, with $a, b > 0$.

disgamma (*x*,*a*,*b*) Function

Returns the value at x of the distribution function of a gamma random variable $Gamma(a, b)$, with $a, b > 0$.

This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) disgamma(3,5,21);
(%o1)                disgamma(3, 5, 21)
(%i2) %,numer;
(%o2)               4.402663157135039E-7
```

qgamma (*q*,*a*,*b*) Function

Returns the q-quantile of a gamma random variable $Gamma(a, b)$, with $a, b > 0$; in other words, this is the inverse of `disgamma`. Argument q must be an element of $[0, 1]$.

meangamma (*a*,*b*) Function

Returns the mean of a gamma random variable $Gamma(a, b)$, with $a, b > 0$.

vargamma (*a*,*b*) Function

Returns the variance of a gamma random variable $Gamma(a, b)$, with $a, b > 0$.

stdgamma (*a*,*b*) Function

Returns the standard deviation of a gamma random variable $Gamma(a, b)$, with $a, b > 0$.

skwgamma (*a*,*b*) Function

Returns the skewness coefficient of a gamma random variable $Gamma(a, b)$, with $a, b > 0$.

kurgamma (*a*,*b*) Function

Returns the kurtosis coefficient of a gamma random variable $Gamma(a, b)$, with $a, b > 0$.

rgamma_algorithm Option variable

Default value: `ahrens_cheng`

This is the selected algorithm for simulating random gamma variates. Implemented algorithms are `ahrens_cheng` and `inverse`

- `ahrens_cheng`, this is a combinantion of two procedures, depending on the value of parameter *a*:

  For $a >= 1$, Cheng, R.C.H. and Feast, G.M. (1979). *Some simple gamma variate generators*. Appl. Stat., 28, 3, 290-295.

  For $0 < a < 1$, Ahrens, J.H. and Dieter, U. (1974). *Computer methods for sampling from gamma, beta, poisson and binomial distributions*. Computing, 12, 223-246.

- `inverse`, based on the general inverse method.

See also `rgamma`.

rgamma (*a*,*b*) Function
rgamma (*a*,*b*,*n*) Function

Returns a gamma random variate $Gamma(a, b)$, with $a, b > 0$. Calling `rgamma` with a third argument *n*, a random sample of size *n* will be simulated.

There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `rgamma_algorithm`, which defaults to `ahrens_cheng`.

See also `rgamma_algorithm`.

denbeta (*x*,*a*,*b*) Function

Returns the value at x of the density function of a beta random variable $Beta(a, b)$, with $a, b > 0$.

disbeta (*x*,*a*,*b*) Function

Returns the value at x of the distribution function of a beta random variable $Beta(a, b)$, with $a, b > 0$.

This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) disgamma(1/3,15,2);
                                 1
(%o1)                 disgamma(-, 15, 2)
                                 3
(%i2) %,numer;
(%o2)              1.391214268475648E-24
```

qbeta (*q*,*a*,*b*) Function

Returns the *q*-quantile of a beta random variable $Beta(a, b)$, with $a, b > 0$; in other words, this is the inverse of `disbeta`. Argument q must be an element of $[0, 1]$.

meanbeta (*a*,*b*) Function

Returns the mean of a beta random variable $Beta(a, b)$, with $a, b > 0$.

varbeta (*a*,*b*) Function

Returns the variance of a beta random variable $Beta(a, b)$, with $a, b > 0$.

stdbeta (*a*,*b*) Function

Returns the standard deviation of a beta random variable $Beta(a, b)$, with $a, b > 0$.

skwbeta (*a*,*b*) Function

Returns the skewness coefficient of a beta random variable $Beta(a, b)$, with $a, b > 0$.

kurbeta (*a*,*b*) Function

Returns the kurtosis coefficient of a beta random variable $Beta(a, b)$, with $a, b > 0$.

rbeta_algorithm Option variable

Default value: `cheng`

This is the selected algorithm for simulating random beta variates. Implemented algorithms are `cheng`, `inverse` and `ratio`

- `cheng`, this is the algorithm defined in Cheng, R.C.H. (1978). *Generating Beta Variates with Nonintegral Shape Parameters*. Communications of the ACM, 21:317-322
- `inverse`, based on the general inverse method.
- `ratio`, based on the fact that if X is a random variable $Gamma(a, 1)$ and Y is $Gamma(b, 1)$, then the ratio $X/(X + Y)$ is distributed as $Beta(a, b)$.

See also `rbeta`.

rbeta (*a*,*b*) Function
rbeta (*a*,*b*,*n*) Function

Returns a beta random variate $Beta(a, b)$, with $a, b > 0$. Calling `rbeta` with a third argument n, a random sample of size n will be simulated.

There are three algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `rbeta_algorithm`, which defaults to `cheng`.

See also `rbeta_algorithm`.

dencontu (*x*,*a*,*b*) Function
Returns the value at x of the density function of a continuous uniform random variable $cUnif(a, b)$, with $a < b$.

discontu (*x*,*a*,*b*) Function
Returns the value at x of the distribution function of a continuous uniform random variable $cUnif(a, b)$, with $a < b$.

qcontu (*q*,*a*,*b*) Function
Returns the q-quantile of a continuous uniform random variable $cUnif(a, b)$, with $a < b$; in other words, this is the inverse of `discontu`. Argument q must be an element of $[0, 1]$.

meancontu (*a*,*b*) Function
Returns the mean of a continuous uniform random variable $cUnif(a, b)$, with $a < b$.

varcontu (*a*,*b*) Function
Returns the variance of a continuous uniform random variable $cUnif(a, b)$, with $a < b$.

stdcontu (*a*,*b*) Function
Returns the standard deviation of a continuous uniform random variable $cUnif(a, b)$, with $a < b$.

skwcontu (*a*,*b*) Function
Returns the skewness coefficient of a continuous uniform random variable $cUnif(a, b)$, with $a < b$.

kurcontu (*a*,*b*) Function
Returns the kurtosis coefficient of a continuous uniform random variable $cUnif(a, b)$, with $a < b$.

rcontu (*a*,*b*) Function
rcontu (*a*,*b*,*n*) Function
Returns a continuous uniform random variate $cUnif(a, b)$, with $a < b$. Calling `rcontu` with a third argument n, a random sample of size n will be simulated.

This is a direct application of the `random` built-in Maxima function.

See also `random`.

denlog (*x*,*a*,*b*) Function
Returns the value at x of the density function of a logistic random variable $log(a, b)$, with $b > 0$.

dislog (*x*,*a*,*b*) Function
Returns the value at x of the distribution function of a logistic random variable $log(a, b)$, with $b > 0$.

qlog (*q*,*a*,*b*) Function

Returns the *q*-quantile of a logistic random variable $log(a, b)$, with $b > 0$; in other words, this is the inverse of `dislog`. Argument q must be an element of $[0, 1]$.

meanlog (*a*,*b*) Function

Returns the mean of a logistic random variable $log(a, b)$, with $b > 0$.

varlog (*a*,*b*) Function

Returns the variance of a logistic random variable $log(a, b)$, with $b > 0$.

stdlog (*a*,*b*) Function

Returns the standard deviation of a logistic random variable $log(a, b)$, with $b > 0$.

skwlog (*a*,*b*) Function

Returns the skewness coefficient of a logistic random variable $log(a, b)$, with $b > 0$.

kurlog (*a*,*b*) Function

Returns the kurtosis coefficient of a logistic random variable $log(a, b)$, with $b > 0$.

rlog (*a*,*b*) Function
rlog (*a*,*b*,*n*) Function

Returns a logistic random variate $log(a, b)$, with $b > 0$. Calling `rlog` with a third argument n, a random sample of size n will be simulated.

Only the inverse method is implemented.

denpareto (*x*,*a*,*b*) Function

Returns the value at x of the density function of a Pareto random variable $Par(a, b)$, with $a, b > 0$.

dispareto (*x*,*a*,*b*) Function

Returns the value at x of the distribution function of a Pareto random variable $Par(a, b)$, with $a, b > 0$.

qpareto (*q*,*a*,*b*) Function

Returns the *q*-quantile of a Pareto random variable $Par(a, b)$, with $a, b > 0$; in other words, this is the inverse of `dispareto`. Argument q must be an element of $[0, 1]$.

meanpareto (*a*,*b*) Function

Returns the mean of a Pareto random variable $Par(a, b)$, with $a > 1, b > 0$.

varpareto (*a*,*b*) Function

Returns the variance of a Pareto random variable $Par(a, b)$, with $a > 2, b > 0$.

stdpareto (*a*,*b*) Function

Returns the standard deviation of a Pareto random variable $Par(a, b)$, with $a > 2, b > 0$.

skwpareto (*a*,*b*) Function

Returns the skewness coefficient of a Pareto random variable $Par(a, b)$, with $a > 3, b > 0$.

kurpareto (*a*,*b*) Function

Returns the kurtosis coefficient of a Pareto random variable $Par(a, b)$, with $a > 4, b > 0$.

rpareto (*a*,*b*) Function
rpareto (*a*,*b*,*n*) Function

Returns a Pareto random variate $Par(a, b)$, with $a > 0, b > 0$. Calling `rpareto` with a third argument n, a random sample of size n will be simulated.

Only the inverse method is implemented.

denweibull (*x*,*a*,*b*) Function

Returns the value at x of the density function of a Weibull random variable $Wei(a, b)$, with $a, b > 0$.

disweibull (*x*,*a*,*b*) Function

Returns the value at x of the distribution function of a Weibull random variable $Wei(a, b)$, with $a, b > 0$.

qweibull (*q*,*a*,*b*) Function

Returns the q-quantile of a Weibull random variable $Wei(a, b)$, with $a, b > 0$; in other words, this is the inverse of `disweibull`. Argument q must be an element of $[0, 1]$.

meanweibull (*a*,*b*) Function

Returns the mean of a Weibull random variable $Wei(a, b)$, with $a, b > 0$.

varweibull (*a*,*b*) Function

Returns the variance of a Weibull random variable $Wei(a, b)$, with $a, b > 0$.

stdweibull (*a*,*b*) Function

Returns the standard deviation of a Weibull random variable $Wei(a, b)$, with $a, b > 0$.

skwweibull (*a*,*b*) Function

Returns the skewness coefficient of a Weibull random variable $Wei(a, b)$, with $a, b > 0$.

kurweibull (*a*,*b*) Function

Returns the kurtosis coefficient of a Weibull random variable $Wei(a, b)$, with $a, b > 0$.

rweibull (*a*,*b*) Function
rweibull (*a*,*b*,*n*) Function

Returns a Weibull random variate $Wei(a, b)$, with $a, b > 0$. Calling `rweibull` with a third argument n, a random sample of size n will be simulated.

Only the inverse method is implemented.

denrayleigh (*x*,*b*) Function

Returns the value at x of the density function of a Rayleigh random variable $Ray(b)$, with $b > 0$.

The $Ray(b)$ random variable is equivalent to the $Wei(2, 1/b)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull density is returned.

```
(%i1) denrayleigh(x,b);
                                      1
(%o1)              denweibull(x, 2, -)
                                      b
(%i2) assume(x>0,b>0)$ denrayleigh(x,b);
                                   2  2
                          2     - b  x
(%o3)                  2 b  x %e
```

disrayleigh (*x*,*b*) Function

Returns the value at x of the distribution function of a Rayleigh random variable $Ray(b)$, with $b > 0$.

The $Ray(b)$ random variable is equivalent to the $Wei(2, 1/b)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull distribution is returned.

```
(%i1) disrayleigh(x,b);
                                      1
(%o1)              disweibull(x, 2, -)
                                      b
(%i2) assume(x>0,b>0)$ disrayleigh(x,b);
                                 2  2
                              - b  x
(%o3)                  1 - %e
```

qrayleigh (*q*,*b*) Function

Returns the q-quantile of a Rayleigh random variable $Ray(b)$, with $b > 0$; in other words, this is the inverse of `disrayleigh`. Argument q must be an element of $[0, 1]$.

The $Ray(b)$ random variable is equivalent to the $Wei(2, 1/b)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull quantile is returned.

```
(%i1) qrayleigh(0.99,b);
                                       1
(%o1)              qweibull(0.99, 2, -)
                                       b
(%i2) assume(x>0,b>0)$ qrayleigh(0.99,b);
                     2.145966026289347
(%o3)                ------------------
                             b
```

meanrayleigh (*b*) Function

Returns the mean of a Rayleigh random variable $Ray(b)$, with $b > 0$.

The $Ray(b)$ random variable is equivalent to the $Wei(2, 1/b)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull mean is returned.

```
(%i1) meanrayleigh(b);
                                    1
(%o1)                meanweibull(2, -)
                                    b
(%i2) assume(b>0)$ meanrayleigh(b);
                         sqrt(%pi)
(%o3)                    ---------
                            2 b
```

varrayleigh (b) Function

Returns the variance of a Rayleigh random variable $Ray(b)$, with $b > 0$.

The $Ray(b)$ random variable is equivalent to the $Wei(2, 1/b)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull variance is returned.

```
(%i1) varrayleigh(b);
                                   1
(%o1)                varweibull(2, -)
                                   b
(%i2) assume(b>0)$ varrayleigh(b);
                              %pi
                          1 - ---
                               4
(%o3)                     -------
                             2
                            b
```

stdrayleigh (b) Function

Returns the standard deviation of a Rayleigh random variable $Ray(b)$, with $b > 0$.

The $Ray(b)$ random variable is equivalent to the $Wei(2, 1/b)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull standard deviation is returned.

```
(%i1) stdrayleigh(b);
                                   1
(%o1)                stdweibull(2, -)
                                   b
(%i2) assume(b>0)$ stdrayleigh(b);
                                %pi
                      sqrt(1 - ---)
                                 4
(%o3)                 -------------
                            b
```

skwrayleigh (b) Function

Returns the skewness coefficient of a Rayleigh random variable $Ray(b)$, with $b > 0$.

The $Ray(b)$ random variable is equivalent to the $Wei(2, 1/b)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull skewness coefficient is returned.

```
(%i1) skwrayleigh(b);
                                  1
(%o1)                 skwweibull(2, -)
                                  b
(%i2) assume(b>0)$ skwrayleigh(b);
                     3/2
                   %pi      3 sqrt(%pi)
                   ------ - -----------
                     4           4
(%o3)              --------------------
                              %pi 3/2
                          (1 - ---)
                                4
```

kurrayleigh (b) Function

Returns the kurtosis coefficient of a Rayleigh random variable $Ray(b)$, with $b > 0$.

The $Ray(b)$ random variable is equivalent to the $Wei(2, 1/b)$, therefore when Maxima has not enough information to get the result, a nominal form based on the Weibull kurtosis coefficient is returned.

```
(%i1) kurrayleigh(b);
                                  1
(%o1)                kurweibull(2, -)
                                  b
(%i2) assume(b>0)$ kurrayleigh(b);
                                    2
                              3 %pi
                         2 - ------
                                16
(%o3)                    ---------- - 3
                                %pi 2
                           (1 - ---)
                                 4
```

rrayleigh (b) Function

rrayleigh (b,n) Function

Returns a Rayleigh random variate $Ray(b)$, with $b > 0$. Calling `rrayleigh` with a second argument n, a random sample of size n will be simulated.

Only the inverse method is implemented.

denlaplace (x,a,b) Function

Returns the value at x of the density function of a Laplace random variable $Lap(a, b)$, with $b > 0$.

dislaplace (*x*,*a*,*b*) Function
Returns the value at x of the distribution function of a Laplace random variable $Lap(a, b)$, with $b > 0$.

qlaplace (*q*,*a*,*b*) Function
Returns the q-quantile of a Laplace random variable $Lap(a, b)$, with $b > 0$; in other words, this is the inverse of `dislaplace`. Argument q must be an element of $[0, 1]$.

meanlaplace (*a*,*b*) Function
Returns the mean of a Laplace random variable $Lap(a, b)$, with $b > 0$.

varlaplace (*a*,*b*) Function
Returns the variance of a Laplace random variable $Lap(a, b)$, with $b > 0$.

stdlaplace (*a*,*b*) Function
Returns the standard deviation of a Laplace random variable $Lap(a, b)$, with $b > 0$.

skwlaplace (*a*,*b*) Function
Returns the skewness coefficient of a Laplace random variable $Lap(a, b)$, with $b > 0$.

kurlaplace (*a*,*b*) Function
Returns the kurtosis coefficient of a Laplace random variable $Lap(a, b)$, with $b > 0$.

rlaplace (*a*,*b*) Function
rlaplace (*a*,*b*,*n*) Function
Returns a Laplace random variate $Lap(a, b)$, with $b > 0$. Calling `rlaplace` with a third argument n, a random sample of size n will be simulated.

Only the inverse method is implemented.

dencauchy (*x*,*a*,*b*) Function
Returns the value at x of the density function of a Cauchy random variable $Cau(a, b)$, with $b > 0$.

discauchy (*x*,*a*,*b*) Function
Returns the value at x of the distribution function of a Cauchy random variable $Cau(a, b)$, with $b > 0$.

qcauchy (*q*,*a*,*b*) Function
Returns the q-quantile of a Cauchy random variable $Cau(a, b)$, with $b > 0$; in other words, this is the inverse of `discauchy`. Argument q must be an element of $[0, 1]$.

rcauchy (*a*,*b*) Function
rcauchy (*a*,*b*,*n*) Function
Returns a Cauchy random variate $Cau(a, b)$, with $b > 0$. Calling `rcauchy` with a third argument n, a random sample of size n will be simulated.

Only the inverse method is implemented.

dengumbel (*x*,*a*,*b*) Function

Returns the value at x of the density function of a Gumbel random variable $Gum(a, b)$, with $b > 0$.

disgumbel (*x*,*a*,*b*) Function

Returns the value at x of the distribution function of a Gumbel random variable $Gum(a, b)$, with $b > 0$.

qgumbel (*q*,*a*,*b*) Function

Returns the q-quantile of a Gumbel random variable $Gum(a, b)$, with $b > 0$; in other words, this is the inverse of `disgumbel`. Argument q must be an element of $[0, 1]$.

meangumbel (*a*,*b*) Function

Returns the mean of a Gumbel random variable $Gum(a, b)$, with $b > 0$.

```
(%i1) assume(b>0)$  meangumbel(a,b);
(%o2)                   %gamma b + a
```

where symbol `%gamma` stands for the Euler-Mascheroni constant. See also `%gamma`.

vargumbel (*a*,*b*) Function

Returns the variance of a Gumbel random variable $Gum(a, b)$, with $b > 0$.

stdgumbel (*a*,*b*) Function

Returns the standard deviation of a Gumbel random variable $Gum(a, b)$, with $b > 0$.

skwgumbel (*a*,*b*) Function

Returns the skewness coefficient of a Gumbel random variable $Gum(a, b)$, with $b > 0$.

```
(%i1) assume(b>0)$  skwgumbel(a,b);
                    12 sqrt(6) zeta(3)
(%o2)               ------------------
                              3
                            %pi
(%i3) numer:true$ skwgumbel(a,b);
(%o4)               1.139547099404649
```

where `zeta` stands for the Riemann's zeta function.

kurgumbel (*a*,*b*) Function

Returns the kurtosis coefficient of a Gumbel random variable $Gum(a, b)$, with $b > 0$.

rgumbel (*a*,*b*) Function
rgumbel (*a*,*b*,*n*) Function

Returns a Gumbel random variate $Gum(a, b)$, with $b > 0$. Calling `rgumbel` with a third argument n, a random sample of size n will be simulated.

Only the inverse method is implemented.

48.3 Definitions for discrete distributions

denbinomial (*x*,*n*,*p*) Function

Returns the value at x of the probability function of a binomial random variable $B(n, p)$, with $0 < p < 1$ and n a positive integer.

disbinomial (*x*,*n*,*p*) Function

Returns the value at x of the distribution function of a binomial random variable $B(n, p)$, with $0 < p < 1$ and n a positive integer.

This function is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) disbinomial(5,7,1/6);
                                        1
(%o1)              disbinomial(5, 7, -)
                                        6
(%i2) disbinomial(5,7,1/6),numer;
(%o2)               .9998713991769548
```

qbinomial (*q*,*n*,*p*) Function

Returns the q-quantile of a binomial random variable $B(n, p)$, with $0 < p < 1$ and n a positive integer; in other words, this is the inverse of `disbinomial`. Argument q must be an element of $[0, 1]$.

meanbinomial (*n*,*p*) Function

Returns the mean of a binomial random variable $B(n, p)$, with $0 < p < 1$ and n a positive integer.

varbinomial (*n*,*p*) Function

Returns the variance of a binomial random variable $B(n, p)$, with $0 < p < 1$ and n a positive integer.

stdbinomial (*n*,*p*) Function

Returns the standard deviation of a binomial random variable $B(n, p)$, with $0 < p < 1$ and n a positive integer.

skwbinomial (*n*,*p*) Function

Returns the skewness coefficient of a binomial random variable $B(n, p)$, with $0 < p < 1$ and n a positive integer.

kurbinomial (*n*,*p*) Function

Returns the kurtosis coefficient of a binomial random variable $B(n, p)$, with $0 < p < 1$ and n a positive integer.

rbinomial_algorithm Option variable

Default value: `kachit`

This is the selected algorithm for simulating random binomial variates. Implemented algorithms are `kachit`, `bernoulli` and `inverse`:

- `kachit`, based on algorithm described in Kachitvichyanukul, V. and Schmeiser, B.W. (1988) *Binomial Random Variate Generation.* Communications of the ACM, 31, Feb., 216.
- `bernoulli`, based on simulation of Bernoulli trials.
- `inverse`, based on the general inverse method.

See also `rbinomial`.

rbinomial (*n*,*p*) Function
rbinomial (*n*,*p*,*m*) Function

Returns a binomial random variate $B(n, p)$, with $0 < p < 1$ and n a positive integer. Calling `rbinomial` with a third argument m, a random sample of size m will be simulated.

There are three algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `rbinomial_algorithm`, which defaults to `kachit`.

See also `rbinomial_algorithm`.

denpoisson (*x*,*m*) Function

Returns the value at x of the probability function of a Poisson random variable $Poi(m)$, with $m > 0$.

dispoisson (*x*,*m*) Function

Returns the value at x of the distribution function of a Poisson random variable $Poi(m)$, with $m > 0$.

This function is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) dispoisson(3,5);
(%o1)                dispoisson(3, 5)
(%i2) dispoisson(3,5),numer;
(%o2)               .2650259152973617
```

qpoisson (*q*,*m*) Function

Returns the q-quantile of a Poisson random variable $Poi(m)$, with $m > 0$; in other words, this is the inverse of `dispoisson`. Argument q must be an element of $[0, 1]$.

meanpoisson (*m*) Function

Returns the mean of a Poisson random variable $Poi(m)$, with $m > 0$.

varpoisson (*m*) Function

Returns the variance of a Poisson random variable $Poi(m)$, with $m > 0$.

stdpoisson (*m*) Function

Returns the standard deviation of a Poisson random variable $Poi(m)$, with $m > 0$.

skwpoisson (*m*) Function

Returns the skewness coefficient of a Poisson random variable $Poi(m)$, with $m > 0$.

kurpoisson (*m*) Function

Returns the kurtosis coefficient of a Poisson random variable $Poi(m)$, with $m > 0$.

rpoisson_algorithm Option variable

Default value: `kachit`

This is the selected algorithm for simulating random Poisson variates. Implemented algorithms are `ahrens_dieter` and `inverse`:

- `ahrens_dieter`, based on algorithm described in Ahrens, J.H. and Dieter, U. (1982) *Computer Generation of Poisson Deviates From Modified Normal Distributions*. ACM Trans. Math. Software, 8, 2, June,163-179.
- `inverse`, based on the general inverse method.

See also `rpoisson`.

rpoisson (*m*) Function
rpoisson (*m*,*n*) Function

Returns a Poisson random variate $Poi(m)$, with $m > 0$. Calling `rpoisson` with a second argument n, a random sample of size n will be simulated.

There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `rpoisson_algorithm`, which defaults to `ahrens_dieter`.

See also `rpoisson_algorithm`.

denbernoulli (*x*,*p*) Function

Returns the value at x of the probability function of a Bernoulli random variable $Ber(p)$, with $0 < p < 1$.

The $Ber(p)$ random variable is equivalent to the binomial $B(1, p)$, therefore when Maxima has not enough information to get the result, a nominal form based on the binomial probability function is returned.

```
(%i1) denbernoulli(1,p);
(%o1)              denbinomial(1, 1, p)
(%i2) assume(0<p,p<1)$ denbernoulli(1,p);
(%o3)                         p
```

disbernoulli (*x*,*p*) Function

Returns the value at x of the distribution function of a Bernoulli random variable $Ber(p)$, with $0 < p < 1$.

qbernoulli (*q*,*p*) Function

Returns the *q*-quantile of a Bernoulli random variable $Ber(p)$, with $0 < p < 1$; in other words, this is the inverse of `disbernoulli`. Argument q must be an element of $[0, 1]$.

meanbernoulli (*p*) Function

Returns the mean of a Bernoulli random variable $Ber(p)$, with $0 < p < 1$.

The $Ber(p)$ random variable is equivalent to the binomial $B(1, p)$, therefore when Maxima has not enough information to get the result, a nominal form based on the binomial mean is returned.

```
(%i1) meanbernoulli(p);
(%o1)                meanbinomial(1, p)
(%i2) assume(0<p,p<1)$ meanbernoulli(p);
(%o3)                          p
```

varbernoulli (*p*) Function

Returns the variance of a Bernoulli random variable $Ber(p)$, with $0 < p < 1$.

The $Ber(p)$ random variable is equivalent to the binomial $B(1, p)$, therefore when Maxima has not enough information to get the result, a nominal form based on the binomial variance is returned.

```
(%i1) varbernoulli(p);
(%o1)                varbinomial(1, p)
(%i2) assume(0<p,p<1)$ varbernoulli(p);
(%o3)                     (1 - p) p
```

stdbernoulli (*p*) Function

Returns the standard deviation of a Bernoulli random variable $Ber(p)$, with $0 < p < 1$.

The $Ber(p)$ random variable is equivalent to the binomial $B(1, p)$, therefore when Maxima has not enough information to get the result, a nominal form based on the binomial standard deviation is returned.

```
(%i1) stdbernoulli(p);
(%o1)               stdbinomial(1, p)
(%i2) assume(0<p,p<1)$ stdbernoulli(p);
(%o3)              sqrt(1 - p) sqrt(p)
```

skwbernoulli (*p*) Function

Returns the skewness coefficient of a Bernoulli random variable $Ber(p)$, with $0 < p < 1$.

The $Ber(p)$ random variable is equivalent to the binomial $B(1, p)$, therefore when Maxima has not enough information to get the result, a nominal form based on the binomial skewness coefficient is returned.

```
(%i1) skwbernoulli(p);
(%o1)               skwbinomial(1, p)
(%i2) assume(0<p,p<1)$ skwbernoulli(p);
                           1 - 2 p
(%o3)              --------------------
                   sqrt(1 - p) sqrt(p)
```

kurbernoulli (*p*) Function

Returns the kurtosis coefficient of a Bernoulli random variable $Ber(p)$, with $0 < p < 1$.

The $Ber(p)$ random variable is equivalent to the binomial $B(1, p)$, therefore when Maxima has not enough information to get the result, a nominal form based on the binomial kurtosis coefficient is returned.

```
(%i1) kurbernoulli(p);
(%o1)              kurbinomial(1, p)
(%i2) assume(0<p,p<1)$ kurbernoulli(p);
                    1 - 6 (1 - p) p
(%o3)               ---------------
                       (1 - p) p
```

rbernoulli (*p*) Function

rbernoulli (*p*,*n*) Function

Returns a Bernoulli random variate $Ber(p)$, with $0 < p < 1$. Calling `rbernoulli` with a second argument n, a random sample of size n will be simulated.

This is a direct application of the `random` built-in Maxima function.

See also `random`.

dengeo (*x*,*p*) Function

Returns the value at x of the probability function of a geometric random variable $Geo(p)$, with $0 < p < 1$.

disgeo (*x*,*p*) Function

Returns the value at x of the distribution function of a geometric random variable $Geo(p)$, with $0 < p < 1$.

qgeo (*q*,*p*) Function

Returns the q-quantile of a geometric random variable $Geo(p)$, with $0 < p < 1$; in other words, this is the inverse of `disgeo`. Argument q must be an element of $[0, 1]$.

meangeo (*p*) Function

Returns the mean of a geometric random variable $Geo(p)$, with $0 < p < 1$.

vargeo (*p*) Function

Returns the variance of a geometric random variable $Geo(p)$, with $0 < p < 1$.

stdgeo (*p*) Function

Returns the standard deviation of a geometric random variable $Geo(p)$, with $0 < p < 1$.

skwgeo (*p*) Function

Returns the skewness coefficient of a geometric random variable $Geo(p)$, with $0 < p < 1$.

kurgeo (*p*) Function

Returns the kurtosis coefficient of a geometric random variable $Geo(p)$, with $0 < p < 1$.

rgeo_algorithm Option variable

Default value: `bernoulli`

This is the selected algorithm for simulating random geometric variates. Implemented algorithms are `bernoulli`, `devroye` and `inverse`:

- `bernoulli`, based on simulation of Bernoulli trials.
- `devroye`, based on algorithm described in Devroye, L. (1986) *Non-Uniform Random Variate Generation.* Springer Verlag, p. 480.
- `inverse`, based on the general inverse method.

See also `rgeo`.

rgeo (*p*) Function
rgeo (*p,n*) Function

Returns a geometric random variate $Geo(p)$, with $0 < p < 1$. Calling `rgeo` with a second argument n, a random sample of size n will be simulated.

There are three algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `rgeo_algorithm`, which defaults to `bernoulli`.

See also `rgeo_algorithm`.

dendiscu (*x,n*) Function

Returns the value at x of the probability function of a discrete uniform random variable $dUnif(n)$, with n a strictly positive integer.

disdiscu (*x,n*) Function

Returns the value at x of the distribution function of a discrete uniform random variable $dUnif(n)$, with n a strictly positive integer.

qdiscu (*q,n*) Function

Returns the q-quantile of a discrete uniform random variable $dUnif(n)$, with n a strictly positive integer; in other words, this is the inverse of `disdiscu`. Argument q must be an element of $[0, 1]$.

meandiscu (*n*) Function

Returns the mean of a discrete uniform random variable $dUnif(n)$, with n a strictly positive integer.

vardiscu (*n*) Function

Returns the variance of a discrete uniform random variable $dUnif(n)$, with n a strictly positive integer.

stddiscu (*n*) Function
Returns the standard deviation of a discrete uniform random variable $dUnif(n)$, with n a strictly positive integer.

skwdiscu (*n*) Function
Returns the skewness coefficient of a discrete uniform random variable $dUnif(n)$, with n a strictly positive integer.

kurdiscu (*n*) Function
Returns the kurtosis coefficient of a discrete uniform random variable $dUnif(n)$, with n a strictly positive integer.

rdiscu (*n*) Function
rdiscu (*n,m*) Function
Returns a uniform random variable $dUnif(n)$, with n a strictly positive integer. Calling `rdiscu` with a second argument m, a random sample of size m will be simulated.

This is a direct application of the `random` built-in Maxima function.

See also `random`.

denhypergeo (*x,n1,n2,n*) Function
Returns the value at x of the probability function of a hypergeometric random variable $Hyp(n1, n2, n)$, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$.

dishypergeo (*x,n1,n2,n*) Function
Returns the value at x of the distribution function of a hypergeometric random variable $Hyp(n1, n2, n)$, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$.

qhypergeo (*q,n1,n2,n*) Function
Returns the q-quantile of a hypergeometric random variable $Hyp(n1, n2, n)$, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$; in other words, this is the inverse of `dishypergeo`. Argument q must be an element of $[0, 1]$.

meanhypergeo (*n1,n2,n*) Function
Returns the mean of a discrete uniform random variable $Hyp(n1, n2, n)$, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$.

varhypergeo (*n1,n2,n*) Function
Returns the variance of a hypergeometric random variable $Hyp(n1, n2, n)$, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$.

stdhypergeo (*n1,n2,n*) Function
Returns the standard deviation of a hypergeometric random variable $Hyp(n1, n2, n)$, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$.

skwhypergeo (*n1,n2,n*) Function
Returns the skewness coefficient of a hypergeometric random variable $Hyp(n1, n2, n)$, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$.

kurhypergeo (*n1,n2,n*) Function

Returns the kurtosis coefficient of a hypergeometric random variable $Hyp(n1, n2, n)$, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$.

rhypergeo_algorithm Option variable

Default value: `kachit`

This is the selected algorithm for simulating random hypergeometric variates. Implemented algorithms are `kachit` and `inverse`:

- `kachit`, based on algorithm described in Kachitvichyanukul, V., Schmeiser, B.W. (1985) *Computer generation of hypergeometric random variates*. Journal of Statistical Computation and Simulation 22, 127-145.
- `inverse`, based on the general inverse method.

See also `rhypergeo`.

rhypergeo (*n1,n2,n*) Function
rhypergeo (*n1,n2,n,m*) Function

Returns a hypergeometric random variate $Hyp(n1, n2, n)$, with *n1*, *n2* and n non negative integers and $n <= n1 + n2$. Calling `rhypergeo` with a fourth argument m, a random sample of size m will be simulated.

There are two algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `rhypergeo_algorithm`, which defaults to `kachit`.

See also `rhypergeo_algorithm`.

dennegbinom (*x,n,p*) Function

Returns the value at x of the probability function of a negative binomial random variable $NB(n, p)$, with $0 < p < 1$ and n a positive integer.

disnegbinom (*x,n,p*) Function

Returns the value at x of the distribution function of a negative binomial random variable $NB(n, p)$, with $0 < p < 1$ and n a positive integer.

This function is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) disnegbinom(3,4,1/8);
                                      1
(%o1)              disnegbinom(3, 4, -)
                                      8
(%i2) disnegbinom(3,4,1/8),numer;
(%o2)              .006238937377929698
```

qnegbinom (*q,n,p*) Function

Returns the q-quantile of a negative binomial random variable $NB(n, p)$, with $0 < p < 1$ and n a positive integer; in other words, this is the inverse of `disnegbinom`. Argument q must be an element of $[0, 1]$.

meannegbinom (*n*,*p*) Function

Returns the mean of a negative binomial random variable $NB(n,p)$, with $0 < p < 1$ and n a positive integer.

varnegbinom (*n*,*p*) Function

Returns the variance of a negative binomial random variable $NB(n,p)$, with $0 < p < 1$ and n a positive integer.

stdnegbinom (*n*,*p*) Function

Returns the standard deviation of a negative binomial random variable $NB(n,p)$, with $0 < p < 1$ and n a positive integer.

skwnegbinom (*n*,*p*) Function

Returns the skewness coefficient of a negative binomial random variable $NB(n,p)$, with $0 < p < 1$ and n a positive integer.

kurnegbinom (*n*,*p*) Function

Returns the kurtosis coefficient of a negative binomial random variable $NB(n,p)$, with $0 < p < 1$ and n a positive integer.

rnegbinom_algorithm Option variable

Default value: `bernoulli`

This is the selected algorithm for simulating random negative binomial variates. Implemented algorithms are `devroye`, `bernoulli` and `inverse`:

- `devroye`, based on algorithm described in Devroye, L. (1986) *Non-Uniform Random Variate Generation*. Springer Verlag, p. 480.
- `bernoulli`, based on simulation of Bernoulli trials.
- `inverse`, based on the general inverse method.

See also `rnegbinom`.

rnegbinoml (*n*,*p*) Function
rnegbinom (*n*,*p*,*m*) Function

Returns a negative binomial random variate $NB(n,p)$, with $0 < p < 1$ and n a positive integer. Calling `rnegbinoml` with a third argument m, a random sample of size m will be simulated.

There are three algorithms implemented for this function, the one to be used can be selected giving a certain value to the global variable `rnegbinom_algorithm`, which defaults to `bernoulli`.

See also `rnegbinom_algorithm`.

49 dynamics

49.1 Introduction to dynamics

The additional package `dynamics` includes several functions to create various graphical representations of discrete dynamical systems and fractals, and an implementation of the Runge-Kutta 4th-order numerical method for solving systems of differential equations.

To use the functions in this package you must first load it with `load("dynamics")`.

49.2 Definitions for dynamics

chaosgame ([[*x1, y1*]...[*xm, ym*]], [*x0, y0*], *b*, *n*, *...options...*); Function

Implements the so-called chaos game: the initial point (*x0*, *y0*) is plotted and then one of the *m* points `[`*x1*, *y1*`]`...`[`*xm*, *ym*`]` will be selected at random. The next point plotted will be in the segment from the previous point to the point chosen randomly, at a fraction *b* of the distance from the random point. The procedure is repeated *n* times.

evolution (*F*, *y0*, *n*,*...options...*); Function

Draws *n+1* points in a two-dimensional graph, where the horizontal coordinates of the points are the integers 0, 1, 2, ..., *n*, and the vertical coordinates are the corresponding values *y(n)* of the sequence defined by the recurrence relation

$$y_{n+1} = F(y_n)$$

With initial value *y(0)* equal to *y0*. *F* must be an expression that depends only on the variable *y* (and not on *n*), *y0* must be a real number and *n* must be a positive integer.

evolution2d ([*F*, *G*], [*x0, y0*], *n*, *...options...*); Function

Shows, in a two-dimensional plot, the first *n+1* points in the sequence of points defined by the two-dimensional discrete dynamical system with recurrence relations

$$\begin{cases} x_{n+1} = F(x_n, y_n) \\ y_{n+1} = G(x_n, y_n) \end{cases}$$

With initial values *x0* and *y0*. *F* and *G* must be two expressions that depend just on *x* and *y*.

ifs ([*r1,...,rm*],[*A1,...,Am*], [[*x1,y1*]...[*xm, ym*]], [*x0,y0*], *n*, *...options...*); Function

Implements the Iterated Function System method. This method is similar to the method described in the function `chaosgame`, but instead of shrinking the segment from the current point to the randomly chosen point, the 2 components of that segment will be multiplied by the 2 by 2 matrix *Ai* that corresponds to the point chosen randomly.

The random choice of one of the *m* attractive points can be made with a non-uniform probability distribution defined by the weights *r1,...,rm*. Those weights are given in cumulative form; for instance if there are 3 points with probabilities 0.2, 0.5 and 0.3, the weights *r1*, *r2* and *r3* could be 2, 7 and 10.

orbits (*F, y0, n1, n2,* [*x, x0, xf, xstep*], *...options...*); Function

Draws the orbits diagram for a family of one-dimensional discrete dynamical systems, with one parameter *x*; that kind of diagram is used to study the bifurcations of a one-dimensional discrete system.

The function *F(y)* defines a sequence with a starting value of *y0*, as in the case of the function `evolution`, but in this case that function will also depend on a parameter *x* that will take values in the interval from *x0* to *xf* with increments of *xstep*. Each value used for the parameter *x* is shown on the horizontal axis. The vertical axis will show the *n2* values of the sequence *y(n1+1)*,..., *y(n1+n2+1)* obtained after letting the sequence evolve *n1* iterations.

rk (*ODE, var, initial, domain*) Function

rk ([*ODE1,...,ODEm*], [*v1,...,vm*], [*init1,...,initm*], *domain*) Function

The first form solves numerically one first-order ordinary differential equation, and the second form solves a system of m of those equations, using the 4th order Runge-Kutta method. var represents the dependent variable. ODE must be an expression that depends only on the independent and dependent variables and represents the derivative of the dependent variable with respect to the independent variable.

The independent variable is specified with `domain`, which must be a list of four elements such as:

```
[t, 0, 10, 0.1]
```

the first element of the list identifies the independent variable, the second and third elements are the initial and final values for that variable, and the last element sets the increments that should be used within that interval.

If *m* equations are going to be solved, there should be *m* dependent variables *v1*, *v2*, ..., *vm*. The initial values for those variables will be *init1*, *init2*, ..., *initm*. There will still be just one independent variable defined by `domain`, as in the previous case. *ODE1*, ..., *ODEm* are the expressions for the derivatives of each dependent variable in terms of the independent variable. The only variables that may appear in those expressions are the independent variable and any of the dependent variables.

The result will be a list of lists with *m*+1 elements. Those *m*+1 elements will be the value of the independent variable, followed by the values of the dependent variables corresponding to that point in the interval of integration. If at some point one of the variables becomes too large, the list will stop there. Otherwise, the list will extend until the last value of the independent variable specified by `domain`.

staircase (*F, y0, n, ...options...*); Function

Draws a staircase diagram for the sequence defined by the recurrence relation

$$y_{n+1} = F(y_n)$$

The interpretation and allowed values of the input parameters is the same as for the function `evolution`. A staircase diagram consists of a plot of the function *F(y)*, together with the line *G(y)* `=` *y*. A vertical segment is drawn from the point (*y0*, *y0*) on that line until the point where it intersects the function *F*. From that point a horizontal segment is drawn until it reaches the point (*y1*, *y1*) on the line, and the procedure is repeated *n* times until the point (*yn*, *yn*) is reached.

Options

The options accepted by the functions that plot graphs are:

- Option: `domain` sets the minimum and maximum values for the plot of the function F shown by `staircase`.

  `[domain, -2, 3.5]`

- Option: `pointsize` defines the radius of each point plotted, in units of points.

  `[pointsize, 1.5]`

  The default value is 1.

- Option: `xaxislabel` is a label to put on the horizontal axis.

  `[xaxislabel, "time"]`

- Option: `xcenter` is the x coordinate of the point at the center of the plot. This option is not used by the function `orbits`.

  `[xcenter,3.45]`

- Option: `xradius` is half of the length of the range of values that will be shown in the x direction. This option is not used by the function `orbits`.

  `[xradius,12.5]`

- Option: `yaxislabel` is a label to put on the vertical axis.

  `[yaxislabel, "temperature"]`

- Option: `ycenter` is the y coordinate of the point at the center of the plot.

  `[ycenter,4.5]`

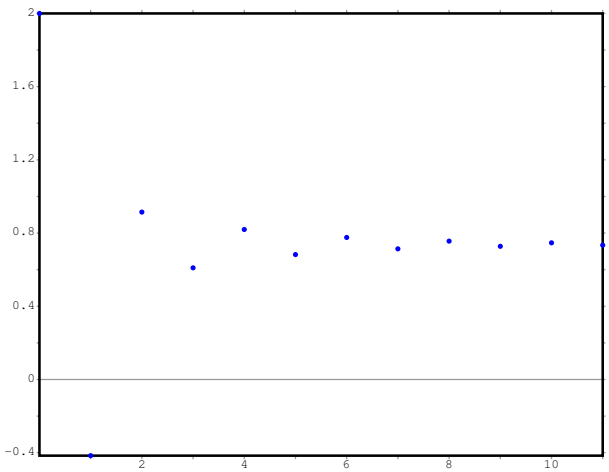
- Option: `yradius` is half of the length of the range of values that will be shown in the y direction.

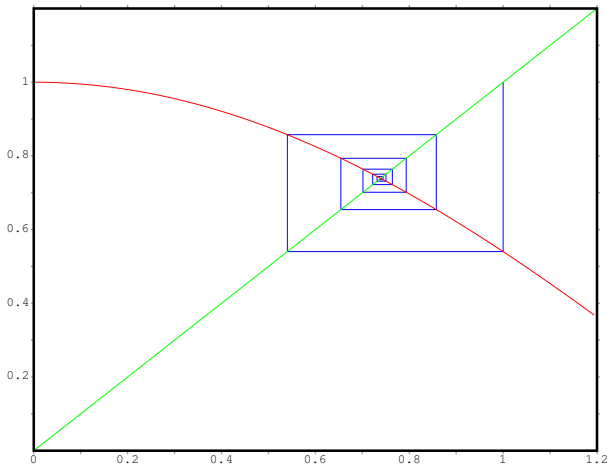
  `[yradius,15]`

Examples

Graphical representation and staircase diagram for the sequence: 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]);
```



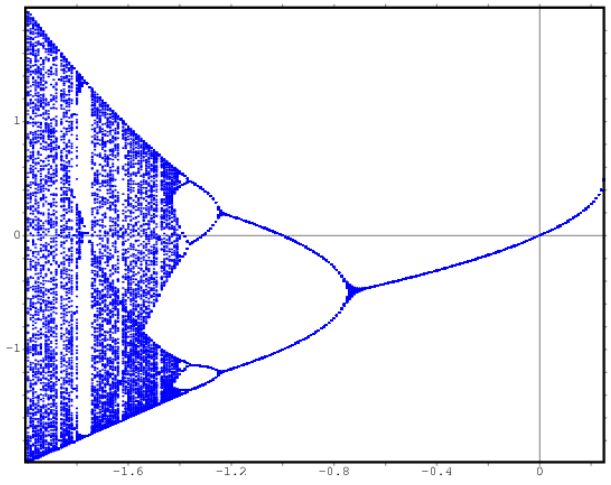


If your system is slow, you'll have to reduce the number of iterations in the following examples. And the pointsize that gives the best results depends on the monitor and the resolution being used.

Orbits diagram for the quadratic map

$$y_{n+1} = x + y_n^2$$

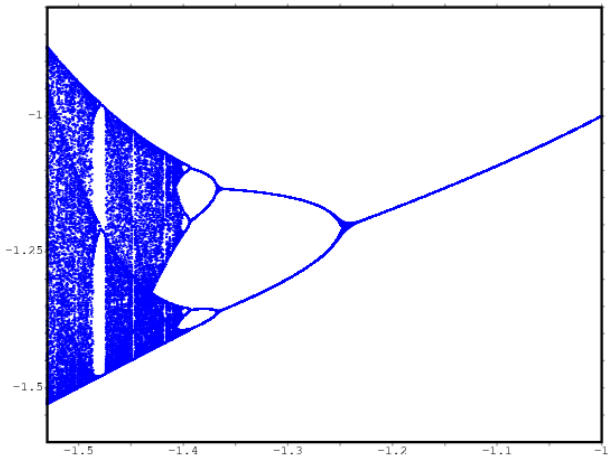
```
(%i4) orbits(y^2+x, 0, 50, 200, [x, -2, 0.25, 0.01], [pointsize, 0.9]);
```



To enlarge the region around the lower bifurcation near x = -1.25 use:

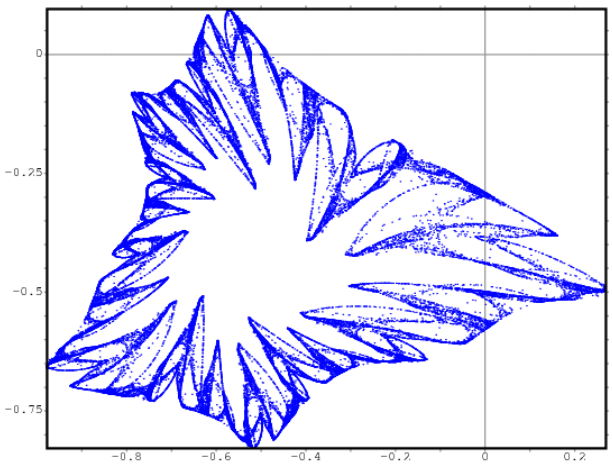
```
(%i5) orbits(x+y^2, 0, 100, 400, [x,-1,-1.53,-0.001], [pointsize,0.9],
```

```
[ycenter,-1.2], [yradius,0.4]);
```



Evolution of a two-dimensional system that leads to a fractal:

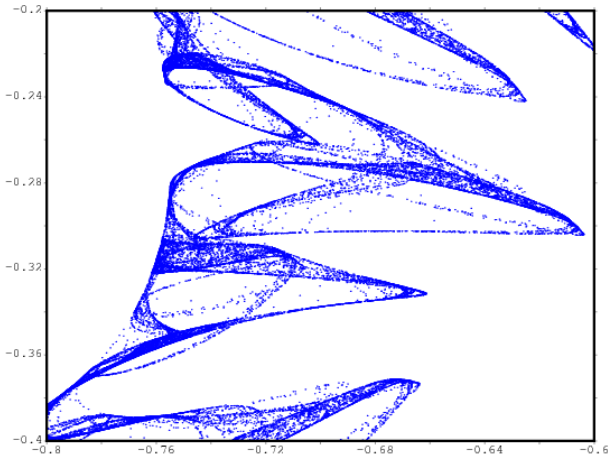
```
(%i6) f: 0.6*x*(1+2*x)+0.8*y*(x-1)-y^2-0.9$
(%i7) g: 0.1*x*(1-6*x+4*y)+0.1*y*(1+9*y)-0.4$
(%i8) evolution2d([f,g],[-0.5,0],50000,[pointsize,0.7]);
```



And an enlargement of a small region in that fractal:

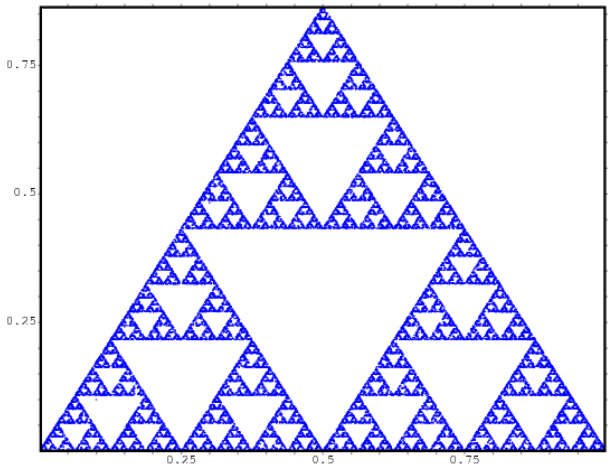
```
(%i9) evolution2d([f,g],[-0.5,0],300000,[pointsize,0.7], [xcenter,-0.7],
```

```
                    [ycenter,-0.3],[xradius,0.1],[yradius,0.1]);
```



A plot of Sierpinsky's triangle, obtained with the chaos game:

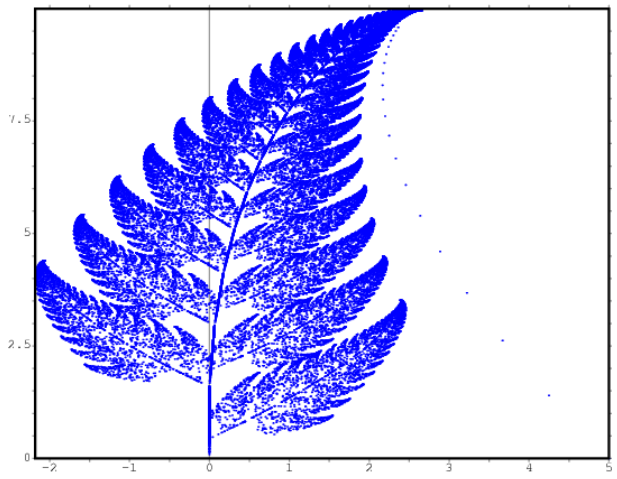
```
(%i9) chaosgame([[0, 0], [1, 0], [0.5, sqrt(3)/2]], [0.1, 0.1], 1/2,
                30000, [pointsize,0.7]);
```



Barnsley's fern, obtained with an Iterated Function System:

```
(%i10) a1: matrix([0.85,0.04],[-0.04,0.85])$
(%i11) a2: matrix([0.2,-0.26],[0.23,0.22])$
(%i12) a3: matrix([-0.15,0.28],[0.26,0.24])$
(%i13) a4: matrix([0,0],[0,0.16])$
(%i14) p1: [0,1.6]$
(%i15) p2: [0,1.6]$
(%i16) p3: [0,0.44]$
(%i17) p4: [0,0]$
(%i18) w: [85,92,99,100]$
```

```
(%i19) ifs(w,[a1,a2,a3,a4],[p1,p2,p3,p4],[5,0],50000,[pointsize,0.9]);
```



To solve numerically the differential equation

$$\frac{dx}{dt} = t - x^2$$

With initial value x(t=0) = 1, in the interval of t from 0 to 8 and with increments of 0.1 for t, use:

```
(%i20) results: rk(t-x^2,x,1,[t,0,8,0.1])$
```

the results will be saved in the list results.

To solve numerically the system:

$$\begin{cases} \frac{dx}{dt} = 4 - x^2 - 4y^2 \\ \frac{dy}{dt} = y^2 - x^2 + 1 \end{cases}$$

for t between 0 and 4, and with values of -1.25 and 0.75 for x and y at t=0

```
(%i21) sol: rk([4-x^2-4*y^2,y^2-x^2+1],[x,y],[-1.25,0.75],[t,0,4,0.02])$
```

50 eval_string

50.1 Definições para eval_string

eval_string (*str*) Função

Entrega a seqüência de caracteres do Maxima *str* como uma expressão do Maxima e a avalia. *str* é uma seqüência de caracteres do Maxima. Essa seqüência 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 seqüência 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 seqüência de caracteres do Maxima *str* como uma expressão do Maxima (sem fazer nenhuma avaliação dessa expressão). *str* é uma seqüência de caracteres do Maxima. Essa seqüência 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 seqüência 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`.

51 f90

51.1 Definições para f90

f90 (*expr*) Function

O comando f90 é uma atualização para o comando `fortran` original do maxima. A diferença primária é o caminho através do qual linhas longas são quebradas.

No exemplo seguinte, observe como o comando `fortran` para linhas dentro de símbolos. O comando `f90` jamais para linha dentro 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 implementação `f90` termina como um rápido reparo em `fortran`. Não é necessáriamente um bom exemplo sobre o qual se deva basear outros tradutores do Maxima para outras linguagens de programação.

Para usar essa função escreva primeiro `load("f90")`.

52 ggf

52.1 Definições para ggf

GGFINFINITY Variável de Opção

Valor padrão: 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* irá ser descartado e o convergente atual irá ser considerado com o valor exato da função geradora; na grande mioria dos casos o grau de todos os quocientes parciais serão 0 ou 1; se você usar um valor muito grande, então você poderá fornecer termos suficientes com o objetivo de fazer o cálculo preciso o suficiente.

Veja também `ggf`.

GGFCFMAX Variável de opção

Valor padrão: 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 irá ser considerada como não sendo uma fração de dois polinômios 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 polinômios) de uma seqüência, sendo dados seus primeiros termos. *l* é uma lista de números.

A solução é retornada como uma fração de dois polinômios. 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")`.

53 impdiff

53.1 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")`.

54 interpol

54.1 Introduction to interpol

Package `interpol` defines de Lagrangian, the linear and the cubic splines methods for polynomial interpolation.

For comments, bugs or suggestions, please contact me at *'mario AT edu DOT xunta DOT es'*.

54.2 Definitions for interpol

lagrange (*points*) Function
lagrange (*points, option*) Function

Computes the polynomial interpolation by the Lagrangian method. Argument *points* must be either:

- a two column matrix, `p:matrix([2,4],[5,6],[9,3])`,
- a list of pairs, `p: [[2,4],[5,6],[9,3]]`,
- a list of numbers, `p: [4,6,3]`, in which case the abscissas will be assigned automatically to 1, 2, 3, etc.

In the first two cases the pairs are ordered with respect to the first coordinate before making computations.

With the *option* argument it is possible to select the name for the independent variable, which is `'x` by default; to define another one, write something like `varname='z`.

Examples:

```
(%i1) load("interpol")$
(%i2) p:[[7,2],[8,2],[1,5],[3,2],[6,7]]$
(%i3) lagrange(p);
            4         3         2
       73 x  - 1402 x  + 8957 x  - 21152 x + 15624
(%o3)  ----------------------------------------
                          420
(%i4) f(x):=''%;
                  4         3         2
             73 x  - 1402 x  + 8957 x  - 21152 x + 15624
(%o4) f(x) := ----------------------------------------
                                420
(%i5) /* Evaluate the polynomial at some points */
      map(f,[2.3,5/7,%pi]);
                           919062
(%o5) [- 1.567535000000005, ------,
                           84035
              4           3           2
       73 %pi  - 1402 %pi  + 8957 %pi  - 21152 %pi + 15624
       ------------------------------------------------]
```

```
                                  420
(%i6) %,numer;
(%o6) [- 1.567535000000005, 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  - 1402 w  + 8957 w  - 21152 w + 15624
(%o8)  ----------------------------------------
                           420
```

charfun2 (*x, a, b*) Function

Returns `true` if number x belongs to the interval $[a, b)$, and `false` otherwise.

linearinterpol (*points*) Function
linearinterpol (*points, option*) Function

Computes the polynomial interpolation by the linear method. Argument *points* must be either:

- a two column matrix, `p:matrix([2,4],[5,6],[9,3])`,
- a list of pairs, `p: [[2,4],[5,6],[9,3]]`,
- a list of numbers, `p: [4,6,3]`, in which case the abscissas will be assigned automatically to 1, 2, 3, etc.

In the first two cases the pairs are ordered with respect to the first coordinate before making computations.

With the *option* argument it is possible to select the name for the independent variable, which is `'x` by default; to define another one, write something like `varname='z`.

Examples:

```
(%i1) load("interpol")$
(%i2) p: matrix([7,2],[8,3],[1,5],[3,2],[6,7])$
(%i3) linearinterpol(p);
(%o3) - ((9 x - 39) charfun2(x, minf, 3)
 + (30 - 6 x) charfun2(x, 7, inf)
 + (30 x - 222) charfun2(x, 6, 7)
 + (18 - 10 x) charfun2(x, 3, 6))/6
(%i4) f(x):=''%;
(%o4) f(x) := - ((9 x - 39) charfun2(x, minf, 3)
 + (30 - 6 x) charfun2(x, 7, inf)
 + (30 x - 222) charfun2(x, 6, 7)
 + (18 - 10 x) charfun2(x, 3, 6))/6
(%i5)  /* Evaluate the polynomial at some points */
       map(f,[7.3,25/7,%pi]);
                        62    18 - 10 %pi
```

```
(%o5)            [2.3, --, - -----------]
                        21         6
(%i6) %,numer;
(%o6)  [2.3, 2.952380952380953, 2.235987755982988]
(%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);
(%o8) - ((9 s - 39) charfun2(s, minf, 3)
 + (30 - 6 s) charfun2(s, 7, inf)
 + (30 s - 222) charfun2(s, 6, 7)
 + (18 - 10 s) charfun2(s, 3, 6))/6
```

cspline (*points*) Function
cspline (*points, option1, option2, ...*) Function

Computes the polynomial interpolation by the cubic splines method. Argument *points* must be either:

- a two column matrix, `p:matrix([2,4],[5,6],[9,3])`,
- a list of pairs, `p: [[2,4],[5,6],[9,3]]`,
- a list of numbers, `p: [4,6,3]`, in which case the abscissas will be assigned automatically to 1, 2, 3, etc.

In the first two cases the pairs are ordered with respect to the first coordinate before making computations.

There are three options to fit specific needs:

- `'d1`, default `'unknown`, is the first derivative at x_1; if it is `'unknown`, the second derivative at x_1 is made equal to 0 (natural cubic spline); if it is equal to a number, the second derivative is calculated based on this number.
- `'dn`, default `'unknown`, is the first derivative at x_n; if it is `'unknown`, the second derivative at x_n is made equal to 0 (natural cubic spline); if it is equal to a number, the second derivative is calculated based on this number.
- `'varname`, default `'x`, is the name of the independent variable.

Examples:

```
(%i1) load("interpol")$
(%i2) p:[[7,2],[8,2],[1,5],[3,2],[6,7]]$
(%i3) /* Unknown first derivatives at the extremes
         is equivalent to natural cubic splines */
      cspline(p);
              3          2
(%o3) ((3477 x  - 10431 x  - 18273 x + 74547)
                                    3           2
 charfun2(x, minf, 3) + (- 15522 x  + 372528 x  - 2964702 x
 + 7842816) charfun2(x, 7, inf)
           3           2
```

```
 + (28290 x  - 547524 x  + 3475662 x - 7184700)
                                3          2
 charfun2(x, 6, 7) + (- 6574 x  + 80028 x  - 289650 x
 + 345924) charfun2(x, 3, 6))/9864
(%i4) f(x):=''%$
(%i5) /* Some evaluations */
      map(f,[2.3,5/7,%pi]), numer;
(%o5) [1.991460766423358, 5.823200187269904,
                                          2.227405312429501]
(%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
(%o7) ((17541 x  - 102933 x  + 153243 x + 33669)
                                   3            2
 charfun2(x, minf, 3) + (- 55692 x  + 1280916 x  - 9801792 x
 + 24990624) charfun2(x, 7, inf)
          3            2
 + (65556 x  - 1265292 x  + 8021664 x - 16597440)
                                 3          2
 charfun2(x, 6, 7) + (- 15580 x  + 195156 x  - 741024 x
 + 927936) charfun2(x, 3, 6))/20304
(%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"]])$
```

55 lindstedt

55.1 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")`.

56 linearalgebra

56.1 Introduction to linearalgebra

`linearalgebra` is a collection of functions for linear algebra.

Example:

```
(%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]

56.2 Definitions for linearalgebra

addmatrices (*f, M_1, ..., M_n*) Function

Using the function *f* as the addition function, return the sum of the matrices *M_1*, ..., *M_n*. The function *f* must accept any number of arguments (a Maxima nary function).

Examples:

```
(%i1) m1 : matrix([1,2],[3,4])$
(%i2) m2 : matrix([7,8],[9,10])$
(%i3) addmatrices('max,m1,m2);
(%o3) matrix([7,8],[9,10])
(%i4) addmatrices('max,m1,m2,5*m1);
(%o4) matrix([7,10],[15,20])
```

blockmatrixp (*M*) Function

Return true if and only if *M* is a matrix and every entry of *M* is a matrix.

columnop (*M, i, j, theta*) Function

If *M* is a matrix, return the matrix that results from doing the column operation `C_i <- C_i - theta * C_j`. If *M* doesn't have a row *i* or *j*, signal an error.

columnswap (*M, i, j*) Function

If *M* is a matrix, swap columns *i* and *j*. If *M* doesn't have a column *i* or *j*, signal an error.

columnspace (*M*) Function

If *M* is a matrix, return `span (v_1, ..., v_n)`, where the set `{v_1, ..., v_n}` is a basis for the column space of *M*. The span of the empty set is `{0}`. Thus, when the column space has only one member, return `span ()`.

copy (*e*) Function

Return a copy of the Maxima expression *e*. Although *e* can be any Maxima expression, the copy function is the most useful when *e* is either a list or a matrix; consider: 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]]
```

Let's try the same experiment, but this time let *mm* be a copy of *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]]
```

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

cholesky (*M*) Function
cholesky (*M*, *field*) Function

Return the Cholesky factorization of the matrix selfadjoint (or hermitian) matrix *M*. The second argument defaults to 'generalring.' For a description of the possible values for *field*, see `lu_factor`.

ctranspose (*M*) Function

Return the complex conjugate transpose of the matrix *M*. The function `ctranspose` uses `matrix_element_transpose` to transpose each matrix element.

diag_matrix (*d_1, d_2,...,d_n*) Function

Return a diagonal matrix with diagonal entries *d_1*, *d_2*,...,*d_n*. When the diagonal entries are matrices, the zero entries of the returned matrix are zero matrices of the appropriate size; for example:

```
(%i1) 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*) Function

Return the dotproduct of vectors *u* and *v*. This is the same as `conjugate (transpose (u)) . v`. The arguments *u* and *v* must be column vectors.

get_lu_factors (*x*) Function

When *x* = `lu_factor (A)`, then `get_lu_factors` returns a list of the form `[P, L, U]`, where *P* is a permutation matrix, *L* is lower triangular with ones on the diagonal, and *U* is upper triangular, and *A* = *P L U*.

hankel (*col*) Function
hankel (*col, row*) Function
Return a Hankel matrix *H*. The first first column of *H* is *col*; except for the first entry, the last row of *H* is *row*. The default for *row* is the zero vector with the same length as *col*.

hessian (*f,vars*) Function
Return the hessian matrix of *f* with respect to the variables in the list *vars*. The *i,j* entry of the hessian matrix is *diff(f vars[i],1,vars[j],1)*.

hilbert_matrix (*n*) Function
Return the *n* by *n* Hilbert matrix. When *n* isn't a positive integer, signal an error.

identfor (*M*) Function
identfor (*M, fld*) Function
Return an identity matrix that has the same shape as the matrix *M*. The diagonal entries of the identity matrix are the multiplicative identity of the field *fld*; the default for *fld* is *generalring*.

The first argument *M* should be a square matrix or a non-matrix. When *M* is a matrix, each entry of *M* can be a square matrix – thus *M* can be a blocked Maxima matrix. The matrix can be blocked to any (finite) depth.

See also `zerofor`

invert_by_lu (*M*, (*rng generalring*)) Function
Invert a matrix *M* by using the LU factorization. The LU factorization is done using the ring *rng*.

kronecker_product (*A*, *B*) Function
Return the Kronecker product of the matrices *A* and *B*.

listp (*e*, *p*) Function
listp (*e*) Function
Given an optional argument *p*, return `true` if *e* is a Maxima list and *p* evaluates to `true` for every list element. When `listp` is not given the optional argument, return `true` if *e* is a Maxima list. In all other cases, return `false`.

locate_matrix_entry (*M, r_1, c_1, r_2, c_2, f, rel*) Function
The first argument must be a matrix; the arguments *r_1* through *c_2* determine a sub-matrix of *M* that consists of rows *r_1* through *r_2* and columns *c_1* through *c_2*.

Find a entry in the sub-matrix *M* that satisfies some property. Three cases:

(1) *rel* = `'bool` and *f* a predicate:

Scan the sub-matrix from left to right then top to bottom, and return the index of the first entry that satisfies the predicate *f*. If no matrix entry satisfies *f*, return false.

(2) *rel* = `'max` and *f* real-valued:

Scan the sub-matrix looking for an entry that maximizes *f*. Return the index of a maximizing entry.

(3) *rel* = `'min` and *f* real-valued:

Scan the sub-matrix looking for an entry that minimizes *f*. Return the index of a minimizing entry.

lu_backsub (*M, b*) Function

When *M* = `lu_factor (`*A*`,` *field*`)`, then `lu_backsub (`*M*`,` *b*`)` solves the linear system *A x* = *b*.

lu_factor (*M, field*) Function

Return a list of the form `[`*LU*`,` *perm*`,` *fld*`]`, or `[`*LU*`,` *perm*`,` *fld*`,` *lower-cnd upper-cnd*`]`, where

(1) The matrix *LU* contains the factorization of *M* in a packed form. Packed form means three things: First, the rows of *LU* are permuted according to the list *perm*. If, for example, *perm* is the list `[3,2,1]`, the actual first row of the *LU* factorization is the third row of the matrix *LU*. Second, the lower triangular factor of m is the lower triangular part of *LU* with the diagonal entries replaced by all ones. Third, the upper triangular factor of *M* is the upper triangular part of *LU*.

(2) When the field is either `floatfield` or `complexfield`, the numbers *lower-cnd* and *upper-cnd* are lower and upper bounds for the infinity norm condition number of *M*. For all fields, the condition number might not be estimated; for such fields, `lu_factor` returns a two item list. Both the lower and upper bounds can differ from their true values by arbitrarily large factors. (See also `mat_cond`.)

The argument *M* must be a square matrix.

The optional argument *fld* must be a symbol that determines a ring or field. The pre-defined fields and rings are:

(a) `generalring` – the ring of Maxima expressions, (b) `floatfield` – the field of floating point numbers of the type double, (c) `complexfield` – the field of complex floating point numbers of the type double, (d) `crering` – the ring of Maxima CRE expressions, (e) `rationalfield` – the field of rational numbers, (f) `runningerror` – track the all floating point rounding errors, (g) `noncommutingring` – the ring of Maxima expressions where multiplication is the non-commutative dot operator.

When the field is `floatfield`, `complexfield`, or `runningerror`, the algorithm uses partial pivoting; for all other fields, rows are switched only when needed to avoid a zero pivot.

Floating point addition arithmetic isn't associative, so the meaning of 'field' differs from the mathematical definition.

A member of the field `runningerror` is a two member Maxima list of the form `[x,n]`,where *x* is a floating point number and `n` is an integer. The relative difference between the 'true' value of `x` and `x` is approximately bounded by the machine epsilon times `n`. The running error bound drops some terms that of the order the square of the machine epsilon.

There is no user-interface for defining a new field. A user that is familiar with Common Lisp should be able to define a new field. To do this, a user must define functions for

the arithmetic operations and functions for converting from the field representation to Maxima and back. Additionally, for ordered fields (where partial pivoting will be used), a user must define functions for the magnitude and for comparing field members. After that all that remains is to define a Common Lisp structure `mring`. The file `mring` has many examples.

To compute the factorization, the first task is to convert each matrix entry to a member of the indicated field. When conversion isn't possible, the factorization halts with an error message. Members of the field needn't be Maxima expressions. Members of the `complexfield`, for example, are Common Lisp complex numbers. Thus after computing the factorization, the matrix entries must be converted to Maxima expressions.

See also `get_lu_factors`.

Examples:

```
(%i1) 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*) Function
mat_cond (*M*, *inf*) Function

Return the *p*-norm matrix condition number of the matrix *m*. The allowed values for *p* are 1 and *inf*. This function uses the LU factorization to invert the matrix *m*. Thus the running time for `mat_cond` is proportional to the cube of the matrix size; `lu_factor` determines lower and upper bounds for the infinity norm condition number in time proportional to the square of the matrix size.

mat_norm (*M*, *1*) Function
mat_norm (*M*, *inf*) Function
mat_norm (*M*, *frobenius*) Function

Return the matrix *p*-norm of the matrix *M*. The allowed values for *p* are 1, `inf`, and `frobenius` (the Frobenius matrix norm). The matrix *M* should be an unblocked matrix.

matrixp (*e*, *p*) Function
matrixp (*e*) Function

Given an optional argument *p*, return `true` if *e* is a matrix and *p* evaluates to `true` for every matrix element. When `matrixp` is not given an optional argument, return `true` if `e` is a matrix. In all other cases, return `false`.

See also `blockmatrixp`

matrix_size (*M*) Function

Return a two member list that gives the number of rows and columns, respectively of the matrix *M*.

mat_fullunblocker (*M*) Function

If *M* is a block matrix, unblock the matrix to all levels. If *M* is a matrix, return *M*; otherwise, signal an error.

mat_trace (*M*) Function

Return the trace of the matrix *M* If *M* isn't a matrix, return a noun form. When *M* is a block matrix, `mat_trace(M)` returns the same value as does `mat_trace(mat_unblocker(m))`.

mat_unblocker (*M*) Function

If *M* is a block matrix, unblock *M* one level. If *M* is a matrix, `mat_unblocker (M)` returns *M*; otherwise, signal an error.

Thus if each entry of *M* is matrix, `mat_unblocker (M)` returns an unblocked matrix, but if each entry of *M* is a block matrix, `mat_unblocker (M)` returns a block matrix with on less level of blocking.

If you use block matrices, most likely you'll want to set `matrix_element_mult` to `"."` and `matrix_element_transpose` to `'transpose`. See also `mat_fullunblocker`.

Example:

```
(%i1) 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*) Function

Return `true` if and only if *n* `>= 0` and *n* is an integer.

nullspace (*M*) Function

If *M* is a matrix, return `span (v_1, ..., v_n)`, where the set `{v_1, ..., v_n}` is a basis for the nullspace of *M*. The span of the empty set is `{0}`. Thus, when the nullspace has only one member, return `span ()`.

nullity (*M*) Function

If *M* is a matrix, return the dimension of the nullspace of *M*.

orthogonal_complement (*v_1, ..., v_n*) Function

Return `span (u_1, ..., u_m)`, where the set `{u_1, ..., u_m}` is a basis for the orthogonal complement of the set `(v_1, ..., v_n)`.

Each vector *v_1* through *v_n* must be a column vector.

polynomialp (*p, L, coeffp, exponp*) Function
polynomialp (*p, L, coeffp*) Function
polynomialp (*p, L*) Function

Return true if *p* is a polynomial in the variables in the list *L*, The predicate *coeffp* must evaluate to `true` for each coefficient, and the predicate *exponp* must evaluate to `true` for all exponents of the variables in *L*. If you want to use a non-default value

for *exponp*, you must supply *coeffp* with a value even if you want to use the default for *coeffp*.

`polynomialp` (*p*, *L*, *coeffp*) is equivalent to `polynomialp` (*p*, *L*, *coeffp*, `'nonnegintegerp`).

`polynomialp` (*p*, *L*) is equivalent to `polynomialp` (*p*, L, `'constantp`, `'nonnegintegerp`).

The polynomial needn't be expanded:

```
(%i1) 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
```

An example using non-default values for coeffp and 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
```

Polynomials with two variables:

```
(%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*) Function

If *p* is a polynomial in *x*, return the companion matrix of *p*. For a monic polynomial *p* of degree *n*, we have *p* = `(-1)^`*n* `charpoly (polytocompanion (`*p*, *x*`))`.

When *p* isn't a polynomial in *x*, signal an error.

ptriangularize (*M*, *v*) Function

If *M* is a matrix with each entry a polynomial in *v*, return a matrix *M2* such that

(1) *M2* is upper triangular,

(2) *M2* = *E_n* ... *E_1* *M*, where *E_1* through *E_n* are elementary matrices whose entries are polynomials in *v*,

(3) `|det (`*M*`)| = |det (`*M2*`)|`,

Note: This function doesn't check that every entry is a polynomial in *v*.

rowop (*M, i, j, theta*) Function

If *M* is a matrix, return the matrix that results from doing the row operation `R_i <- R_i - theta * R_j`. If *M* doesn't have a row *i* or *j*, signal an error.

rank (*M*) Function

Return the rank of that matrix *M*. The rank is the dimension of the column space. Example:

```
(%i1) load(linearalgebra)$
(%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*) Function

If *M* is a matrix, swap rows *i* and *j*. If *M* doesn't have a row *i* or *j*, signal an error.

toeplitz (*col*) Function
toeplitz (*col, row*) Function

Return a Toeplitz matrix *T*. The first first column of *T* is *col*; except for the first entry, the first row of *T* is *row*. The default for *row* is complex conjugate of *col*. Example:

```
(%i1) 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*]) Function

Return a *n* by *n* matrix whose *i*-th row is `[1, `*x_i*`, `*x_i*`^2, ... `*x_i*`^(`*n*`-1)]`.

zerofor (*M*) Function
zerofor (*M*, *fld*) Function

Return a zero matrix that has the same shape as the matrix *M*. Every entry of the zero matrix matrix is the additive identity of the field *fld*; the default for *fld* is *generalring*.

The first argument *M* should be a square matrix or a non-matrix. When *M* is a matrix, each entry of *M* can be a square matrix – thus *M* can be a blocked Maxima matrix. The matrix can be blocked to any (finite) depth.

See also `identfor`

zeromatrixp (*M*) Function

If *M* is not a block matrix, return `true` if `is (equal (`*e*`, 0))` is true for each element *e* of the matrix *M*. If *M* is a block matrix, return `true` if `zeromatrixp` evaluates to true for each element of *e*.

57 lsquares

57.1 Definitions for lsquares

DETCOEF Global variable

This variable is used by functions `lsquares` and `plsquares` to store the Coefficient of Determination which measures the goodness of fit. It ranges from 0 (no correlation) to 1 (exact correlation).

When `plsquares` is called with a list of dependent variables, *DETCOEF* is set to a list of Coefficients of Determination. See `plsquares` for details.

See also `lsquares`.

lsquares (*Mat,VarList,equation,ParamList*) Function
lsquares (*Mat,VarList,equation,ParamList,GuessList*) Function

Multiple nonlinear equation adjustment of a data table by the "least squares" method. *Mat* is a matrix containing the data, *VarList* is a list of variable names (one for each *Mat* column), *equation* is the equation to adjust (it must be in the form: `depvar=f(indepvari,..., paramj,...)`, `g(depvar)=f(indepvari,..., paramj,...)` or `g(depvar, paramk,...)=f(indepvari,..., paramj,...)`), *ParamList* is the list of the parameters to obtain, and *GuessList* is an optional list of initial approximations to the parameters; when this last argument is present, `mnewton` is used instead of `solve` in order to get the parameters.

The equation may be fully nonlinear with respect to the independent variables and to the dependent variable. In order to use `solve()`, the equations must be linear or polynomial with respect to the parameters. Equations like `y=a*b^x+c` may be adjusted for `[a,b,c]` with `solve` if the `x` values are little positive integers and there are few data (see the example in lsquares.dem). `mnewton` allows to adjust a nonlinear equation with respect to the parameters, but a good set of initial approximations must be provided.

If possible, the adjusted equation is returned. If there exists more than one solution, a list of equations is returned. The Coefficient of Determination is displayed in order to inform about the goodness of fit, from 0 (no correlation) to 1 (exact correlation). This value is also stored in the global variable *DETCOEF*.

Examples using `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
```

Examples using `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
```

To use this function write first `load("lsquares")`. See also `DETCOEF` and `mnewton`.

plsquares (*Mat,VarList,depvars*) Function
plsquares (*Mat,VarList,depvars,maxexpon*) Function
plsquares (*Mat,VarList,depvars,maxexpon,maxdegree*) Function

Multivariable polynomial adjustment of a data table by the "least squares" method. *Mat* is a matrix containing the data, *VarList* is a list of variable names (one for each Mat column, but use `"-"` instead of varnames to ignore Mat columns), *depvars* is the name of a dependent variable or a list with one or more names of dependent variables (which names should be in *VarList*), *maxexpon* is the optional maximum exponent for each independent variable (1 by default), and *maxdegree* is the optional maximum polynomial degree (*maxexpon* by default); note that the sum of exponents of each term must be equal or smaller than *maxdegree*, and if `maxdgree = 0` then no limit is applied.

If *depvars* is the name of a dependent variable (not in a list), `plsquares` returns the adjusted polynomial. If *depvars* is a list of one or more dependent variables, `plsquares` returns a list with the adjusted polynomial(s). The Coefficients of Determination are displayed in order to inform about the goodness of fit, which ranges from 0 (no correlation) to 1 (exact correlation). These values are also stored in the global variable *DETCOEF* (a list if *depvars* is a list).

A simple example of multivariable linear adjustment:

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

(%i2) plsquares(matrix([1,2,0],[3,5,4],[4,7,9],[5,8,10]),
                [x,y,z],z);
     Determination Coefficient for z = .9897039897039897
                        11 y - 9 x - 14
(%o2)                z = ---------------
                                3
```

The same example without degree restrictions:

```
(%i3) plsquares(matrix([1,2,0],[3,5,4],[4,7,9],[5,8,10]),
                [x,y,z],z,1,0);
     Determination Coefficient for z = 1.0
                     x y + 23 y - 29 x - 19
(%o3)              z = ----------------------
                                6
```

How many diagonals does a N-sides polygon have? What polynomial degree should be used?

```
(%i4) plsquares(matrix([3,0],[4,2],[5,5],[6,9],[7,14],[8,20]),
                [N,diagonals],diagonals,5);
     Determination Coefficient for diagonals = 1.0
                                 2
                                N  - 3 N
(%o4)                diagonals = --------
                                   2
(%i5) ev(%, N=9);   /* Testing for a 9 sides polygon */
(%o5)                  diagonals = 27
```

How many ways do we have to put two queens without they are threatened into a n x n chessboard?

```
(%i6) plsquares(matrix([0,0],[1,0],[2,0],[3,8],[4,44]),
                [n,positions],[positions],4);
     Determination Coefficient for [positions] = [1.0]
                         4       3      2
                      3 n  - 10 n  + 9 n  - 2 n
(%o6)     [positions = ------------------------]
                                 6
(%i7) ev(%[1], n=8); /* Testing for a (8 x 8) chessboard */
(%o7)                 positions = 1288
```

An example with six dependent variables:

```
(%i8) mtrx:matrix([0,0,0,0,0,1,1,1],[0,1,0,1,1,1,0,0],
```

```
                    [1,0,0,1,1,1,0,0],[1,1,1,1,0,0,0,1])$
(%i8) plsquares(mtrx,[a,b,_And,_Or,_Xor,_Nand,_Nor,_Nxor],
                      [_And,_Or,_Xor,_Nand,_Nor,_Nxor],1,0);
      Determination Coefficient for
[_And, _Or, _Xor, _Nand, _Nor, _Nxor] =
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
(%o2) [_And = a b, _Or = - a b + b + a,
_Xor = - 2 a b + b + a, _Nand = 1 - a b,
_Nor = a b - b - a + 1, _Nxor = 2 a b - b - a + 1]
```

To use this function write first `load("lsquares")`.

58 makeOrders

58.1 Definições para makeOrders

makeOrders (*indvarlist,orderlist*) Função

Retorna uma lista de todos os expoentes para um polinômio 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")`.

59 mnewton

59.1 Definições para mnewton

newtonepsilon Variável de opção

Valor padrão: `10.0^(-fpprec/2)`

Precisão para determinar quando a função `mnewton` convergiu em direção à solução.

Veja também `mnewton`.

newtonmaxiter Variável de opção

Valor padrão: `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` and `newtonmaxiter`.

60 numericalio

60.1 Introduction to numericalio

`numericalio` is a collection of functions to read and write data files. The entire file is read to construct one object; partial reads are not supported.

It is assumed that each item to read or write is atomic: an integer, float, bigfloat, string, or symbol, and not a rational or complex number or any other kind of nonatomic expression. These functions may attempt to do something sensible faced with nonatomic expressions, but the results are not specified here and subject to change.

Atoms in both input and output files have the same format as in Maxima batch files or the interactive console. In particular, strings are enclosed in double quotes, backslash `\` prevents any special interpretation of the next character, and the question mark `?` is recognized at the beginning of a symbol to mean a Lisp symbol (as opposed to a Maxima symbol). No continuation character (to join broken lines) is recognized.

separator_flag tells which character separates elements. It is an optional argument for all read and write functions.

For input, these values of *separator_flag* are recognized: `comma` for comma separated values, `pipe` for values separated by the vertical bar character `|`, `semicolon` for values separated by semicolon `;`, and `space` for values separated by space or tab characters. If the file name ends in `.csv` and *separator_flag* is not specified, `comma` is assumed. If the file name ends in something other than `.csv` and `separator_flag` is not specified, `space` is assumed.

For output, the same four flags are recognized as for input, and also `tab`, for values separated by the tab character.

In input, multiple successive space and tab characters count as a single separator. However, multiple comma, pipe, or semicolon characters are significant. Successive comma, pipe, or semicolon characters (with or without intervening spaces or tabs) are considered to have `false` between the separators. For example, `1234,,Foo` is treated the same as `1234,false,Foo`. In output, `false` atoms are written as such; a list `[1234, false, Foo]` is written `1234,false,Foo`, and there is no attempt to collapse the output to `1234,,Foo`.

60.2 Definitions for numericalio

read_matrix (*file_name*) Function
read_matrix (*file_name, separator_flag*) Function

Reads the file *file_name* and returns its entire content as a matrix. If *separator_flag* is not specified, the file is assumed space-delimited.

`read_matrix` infers the size of the matrix from the input data. Each line of the file becomes one row of the matrix. If some lines have different lengths, `read_matrix` complains.

read_lisp_array (*file_name, A*) Function
read_lisp_array (*file_name, A, separator_flag*) Function

`read_lisp_array` requires that the array be declared by `make_array` before calling the read function. (This obviates the need to infer the array dimensions, which could be a hassle for arrays with multiple dimensions.)

`read_lisp_array` does not check to see that the input file conforms in some way to the array dimensions; the input is read as a flat list, then the array is filled using `fillarray`.

read_maxima_array (*file_name, A*) Function
read_maxima_array (*file_name, A, separator_flag*) Function

`read_maxima_array` requires that the array be declared by `array` before calling the read function. (This obviates the need to infer the array dimensions, which could be a hassle for arrays with multiple dimensions.)

`read_maxima_array` does not check to see that the input file conforms in some way to the array dimensions; the input is read as a flat list, then the array is filled using `fillarray`.

read_hashed_array (*file_name, A*) Function
read_hashed_array (*file_name, A, separator_flag*) Function

`read_hashed_array` treats the first item on a line as a hash key, and associates the remainder of the line (as a list) with the key. For example, the line `567 12 17 32 55` is equivalent to `A[567]: [12, 17, 32, 55]$`. Lines need not have the same numbers of elements.

read_nested_list (*file_name*) Function
read_nested_list (*file_name, separator_flag*) Function

`read_nested_list` returns a list which has a sublist for each line of input. Lines need not have the same numbers of elements. Empty lines are *not* ignored: an empty line yields an empty sublist.

read_list (*file_name*) Function
read_list (*file_name, separator_flag*) Function

`read_list` reads all input into a flat list. `read_list` ignores end-of-line characters.

write_data (*X, file_name*) Function
write_data (*object, file_name, separator_flag*) Function

`write_data` writes the object *X* to the file *file_name*.

`write_data` writes matrices in row-major form, with one line per row.

`write_data` writes Lisp and Maxima declared arrays in row-major form, with a new line at the end of every slab. Higher-dimensional slabs are separated by additional new lines.

`write_data` writes hashed arrays with a key followed by the associated list on each line.

`write_data` writes a nested list with each sublist on one line.

`write_data` writes a flat list all on one line.

Whether `write_data` appends or truncates its output file is governed by the global variable `file_output_append`.

61 opsubst

61.1 Definitions for opsubst

opsubst (*f,g,e*) Function
opsubst (*g=f,e*) Function
opsubst ([*g1=f1,g2=f2,..., gn=fn*],*e*) Function

The function `opsubst` is similar to the function `subst`, except that `opsubst` only makes substitutions for the operators in an expression. In general, When *f* is an operator in the expression *e*, substitute *g* for *f* in the expression *e*.

To determine the operator, `opsubst` sets `inflag` to true. This means `opsubst` substitutes for the internal, not the displayed, operator in the expression.

Examples:

```
(%i1) load (opsubst)$

(%i2) opsubst(f,g,g(g(x)));
(%o2)                      f(f(x))
(%i3) opsubst(f,g,g(g));
(%o3)                       f(g)
(%i4) opsubst(f,g[x],g[x](z));
(%o4)                       f(z)
(%i5) opsubst(g[x],f, f(z));
(%o5)                      g (z)
                            x
(%i6) opsubst(tan, sin, sin(sin));
(%o6)                     tan(sin)
(%i7) opsubst([f=g,g=h],f(x));
(%o7)                       h(x)
```

Internally, Maxima does not use the unary negation, division, or the subtraction operators; thus:

```
(%i8) opsubst("+","-",a-b);
(%o8)                      a - b
(%i9) opsubst("f","-",-a);
(%o9)                       - a
(%i10) opsubst("^^","//",a/b);
                             a
(%o10)                       -
                             b
```

The internal representation of -a*b is *(-1,a,b); thus

```
(%i11) opsubst("[","*", -a*b);
(%o11)                   [- 1, a, b]
```

When either operator isn't a Maxima symbol, generally some other function will signal an error:

```
(%i12) opsubst(a+b,f, f(x));

Improper name or value in functional position:
b + a
 -- an error.  Quitting.  To debug this try debugmode(true);
```

However, subscripted operators are allowed:

```
(%i13) opsubst(g[5],f, f(x));
(%o13)                       g (x)
                              5
```

To use this function write first `load("opsubst")`.

62 orthopoly

62.1 Introduction to orthogonal polynomials

`orthopoly` is a package for symbolic and numerical evaluation of several kinds of orthogonal polynomials, including Chebyshev, Laguerre, Hermite, Jacobi, Legendre, and ultraspherical (Gegenbauer) polynomials. Additionally, `orthopoly` includes support for the spherical Bessel, spherical Hankel, and spherical harmonic functions.

For the most part, `orthopoly` follows the conventions of Abramowitz and Stegun *Handbook of Mathematical Functions*, Chapter 22 (10th printing, December 1972); additionally, we use Gradshteyn and Ryzhik, *Table of Integrals, Series, and Products* (1980 corrected and enlarged edition), and Eugen Merzbacher *Quantum Mechanics* (2nd edition, 1970).

Barton Willis of the University of Nebraska at Kearney (UNK) wrote the `orthopoly` package and its documentation. The package is released under the GNU General Public License (GPL).

62.1.1 Getting Started with orthopoly

`load (orthopoly)` loads the `orthopoly` package.

To find the third-order Legendre polynomial,

```
(%i1) legendre_p (3, x);
                        3             2
             5 (1 - x)    15 (1 - x)
(%o1)      - ---------- + ---------- - 6 (1 - x) + 1
                  2             2
```

To express this as a sum of powers of x, apply *ratsimp* or *rat* to the result.

```
(%i2) [ratsimp (%), rat (%)];
                          3           3
                       5 x  - 3 x  5 x  - 3 x
(%o2)/R/              [----------, ----------]
                           2           2
```

Alternatively, make the second argument to `legendre_p` (its "main" variable) a canonical rational expression (CRE).

```
(%i1) legendre_p (3, rat (x));
                                3
                             5 x  - 3 x
(%o1)/R/                     ----------
                                2
```

For floating point evaluation, `orthopoly` uses a running error analysis to estimate an upper bound for the error. For example,

```
(%i1) jacobi_p (150, 2, 3, 0.2);
(%o1) interval(- 0.062017037936715, 1.533267919277521E-11)
```

Intervals have the form `interval (c, r)`, where c is the center and r is the radius of the interval. Since Maxima does not support arithmetic on intervals, in some situations, such

as graphics, you want to suppress the error and output only the center of the interval. To do this, set the option variable `orthopoly_returns_intervals` to `false`.

```
(%i1) orthopoly_returns_intervals : false;
(%o1)                          false
(%i2) jacobi_p (150, 2, 3, 0.2);
(%o2)                  - 0.062017037936715
```

Refer to the section veja [Floating point Evaluation], página 631 for more information.

Most functions in `orthopoly` have a `gradef` property; thus

```
(%i1) diff (hermite (n, x), x);
(%o1)                      2 n H     (x)
                               n - 1
(%i2) diff (gen_laguerre (n, a, x), x);
                  (a)                (a)
            n L   (x) - (n + a) L     (x) unit_step(n)
               n                  n - 1
(%o2)       ----------------------------------------
                              x
```

The unit step function in the second example prevents an error that would otherwise arise by evaluating with n equal to 0.

```
(%i3) ev (%, n = 0);
(%o3)                           0
```

The gradef property only applies to the "main" variable; derivatives with respect other arguments usually result in an error message; for example

```
(%i1) diff (hermite (n, x), x);
(%o1)                      2 n H     (x)
                               n - 1
(%i2) diff (hermite (n, x), n);

Maxima doesn't know the derivative of hermite with respect the first argument
 -- an error.  Quitting.  To debug this try debugmode(true);
```

Generally, functions in `orthopoly` map over lists and matrices. For the mapping to fully evaluate, the option variables `doallmxops` and `listarith` must both be `true` (the defaults). To illustrate the mapping over matrices, consider

```
(%i1) hermite (2, x);
                                    2
(%o1)                     - 2 (1 - 2 x )
(%i2) m : matrix ([0, x], [y, 0]);
                              [ 0  x ]
(%o2)                         [      ]
                              [ y  0 ]
(%i3) hermite (2, m);
                [                                2  ]
                [     - 2        - 2 (1 - 2 x )   ]
(%o3)           [                                  ]
                [             2                    ]
                [ - 2 (1 - 2 y )       - 2        ]
```

In the second example, the `i, j` element of the value is `hermite (2, m[i,j])`; this is not the same as computing `-2 + 4 m . m`, as seen in the next example.

```
(%i4) -2 * matrix ([1, 0], [0, 1]) + 4 * m . m;
                     [ 4 x y - 2      0     ]
(%o4)                [                      ]
                     [     0      4 x y - 2 ]
```

If you evaluate a function at a point outside its domain, generally `orthopoly` returns the function unevaluated. For example,

```
(%i1) legendre_p (2/3, x);
                             P   (x)
(%o1)                         2/3
```

`orthopoly` supports translation into TeX; it also does two-dimensional output on a terminal.

```
(%i1) spherical_harmonic (l, m, theta, phi);
                              m
(%o1)                        Y (theta, phi)
                              l
(%i2) tex (%);
$$Y_{l}^{m}\left(\vartheta,\varphi\right)$$
(%o2)                            false
(%i3) jacobi_p (n, a, a - b, x/2);
                              (a, a - b) x
(%o3)                        P           (-)
                              n           2
(%i4) tex (%);
$$P_{n}^{\left(a,a-b\right)}\left({{x}\over{2}}\right)$$
(%o4)                            false
```

62.1.2 Limitations

When an expression involves several orthogonal polynomials with symbolic orders, it's possible that the expression actually vanishes, yet Maxima is unable to simplify it to zero. If you divide by such a quantity, you'll be in trouble. For example, the following expression vanishes for integers n greater than 1, yet Maxima is unable to simplify it to zero.

```
(%i1) (2*n - 1) * legendre_p (n - 1, x) * x - n * legendre_p (n, x) + (1 - n) * leg
(%o1)  (2 n - 1) P     (x) x - n P (x) + (1 - n) P     (x)
                  n - 1           n               n - 2
```

For a specific n, we can reduce the expression to zero.

```
(%i2) ev (% ,n = 10, ratsimp);
(%o2)                             0
```

Generally, the polynomial form of an orthogonal polynomial is ill-suited for floating point evaluation. Here's an example.

```
(%i1) p : jacobi_p (100, 2, 3, x)$

(%i2) subst (0.2, x, p);
(%o2)                3.4442767023833592E+35
```

```
(%i3) jacobi_p (100, 2, 3, 0.2);
(%o3)  interval(0.18413609135169, 6.8990300925815987E-12)
(%i4) float(jacobi_p (100, 2, 3, 2/10));
(%o4)                   0.18413609135169
```

The true value is about 0.184; this calculation suffers from extreme subtractive cancellation error. Expanding the polynomial and then evaluating, gives a better result.

```
(%i5) p : expand(p)$
(%i6) subst (0.2, x, p);
(%o6) 0.18413609766122982
```

This isn't a general rule; expanding the polynomial does not always result in an expression that is better suited for numerical evaluation. By far, the best way to do numerical evaluation is to make one or more of the function arguments floating point numbers. By doing that, specialized floating point algorithms are used for evaluation.

Maxima's `float` function is somewhat indiscriminant; if you apply `float` to an an expression involving an orthogonal polynomial with a symbolic degree or order parameter, these parameters may be converted into floats; after that, the expression will not evaluate fully. Consider

```
(%i1) assoc_legendre_p (n, 1, x);
                                1
(%o1)                          P (x)
                                n
(%i2) float (%);
                               1.0
(%o2)                         P   (x)
                               n
(%i3) ev (%, n=2, x=0.9);
                              1.0
(%o3)                        P   (0.9)
                              2
```

The expression in (%o3) will not evaluate to a float; orthopoly doesn't recognize floating point values where it requires an integer. Similarly, numerical evaluation of the `pochhammer` function for orders that exceed `pochhammer_max_index` can be troublesome; consider

```
(%i1) x :  pochhammer (1, 10), pochhammer_max_index : 5;
(%o1)                          (1)
                                  10
```

Applying `float` doesn't evaluate x to a float

```
(%i2) float (x);
(%o2)                       (1.0)
                                 10.0
```

To evaluate x to a float, you'll need to bind `pochhammer_max_index` to 11 or greater and apply `float` to x.

```
(%i3) float (x), pochhammer_max_index : 11;
(%o3)                      3628800.0
```

The default value of `pochhammer_max_index` is 100; change its value after loading `orthopoly`.

Finally, be aware that reference books vary on the definitions of the orthogonal polynomials; we've generally used the conventions of conventions of Abramowitz and Stegun.

Before you suspect a bug in orthopoly, check some special cases to determine if your definitions match those used by orthonormal. Definitions often differ by a normalization; occasionally, authors use "shifted" versions of the functions that makes the family orthogonal on an interval other than $(-1, 1)$. To define, for example, a Legendre polynomial that is orthogonal on $(0, 1)$, define

```
(%i1) shifted_legendre_p (n, x) := legendre_p (n, 2*x - 1)$

(%i2) shifted_legendre_p (2, rat (x));
                              2
(%o2)/R/                   6 x  - 6 x + 1
(%i3) legendre_p (2, rat (x));
                                 2
                              3 x  - 1
(%o3)/R/                      --------
                                 2
```

62.1.3 Floating point Evaluation

Most functions in orthopoly use a running error analysis to estimate the error in floating point evaluation; the exceptions are the spherical Bessel functions and the associated Legendre polynomials of the second kind. For numerical evaluation, the spherical Bessel functions call SLATEC functions. No specialized method is used for numerical evaluation of the associated Legendre polynomials of the second kind.

The running error analysis ignores errors that are second or higher order in the machine epsilon (also known as unit roundoff). It also ignores a few other errors. It's possible (although unlikely) that the actual error exceeds the estimate.

Intervals have the form `interval (`c`, `r`)`, where c is the center of the interval and r is its radius. The center of an interval can be a complex number, and the radius is always a positive real number.

Here is an an example.

=

```
(%i1) fpprec : 50$

(%i2) y0 : jacobi_p (100, 2, 3, 0.2);
(%o2) interval(0.1841360913516871, 6.8990300925815987E-12)
(%i3) y1 : bfloat (jacobi_p (100, 2, 3, 1/5));
(%o3) 1.8413609135168563091370224958913493690868904463668b-1
```

Let's test that the actual error is smaller than the error estimate

```
(%i4) is (abs (part (y0, 1) - y1) < part (y0, 2));
(%o4)                          true
```

Indeed, for this example the error estimate is an upper bound for the true error.

Maxima does not support arithmetic on intervals.

```
(%i1) legendre_p (7, 0.1) + legendre_p (8, 0.1);
(%o1) interval(0.18032072148437508, 3.1477135311021797E-15)
        + interval(- 0.19949294375000004, 3.3769353084291579E-15)
```

A user could define arithmetic operators that do interval math. To define interval addition, we can define

```
(%i1) infix ("@+")$

(%i2) "@+"(x,y) := interval (part (x, 1) + part (y, 1), part (x, 2) + part (y, 2))$

(%i3) legendre_p (7, 0.1) @+ legendre_p (8, 0.1);
(%o3) interval(- 0.019172222265624955, 6.5246488395313372E-15)
```

The special floating point routines get called when the arguments are complex. For example,

```
(%i1) legendre_p (10, 2 + 3.0*%i);
(%o1) interval(- 3.876378825E+7 %i - 6.0787748E+7,
                                          1.2089173052721777E-6)
```

Let's compare this to the true value.

```
(%i1) float (expand (legendre_p (10, 2 + 3*%i)));
(%o1)          - 3.876378825E+7 %i - 6.0787748E+7
```

Additionally, when the arguments are big floats, the special floating point routines get called; however, the big floats are converted into double floats and the final result is a double.

```
(%i1) ultraspherical (150, 0.5b0, 0.9b0);
(%o1) interval(- 0.043009481257265, 3.3750051301228864E-14)
```

62.1.4 Graphics and orthopoly

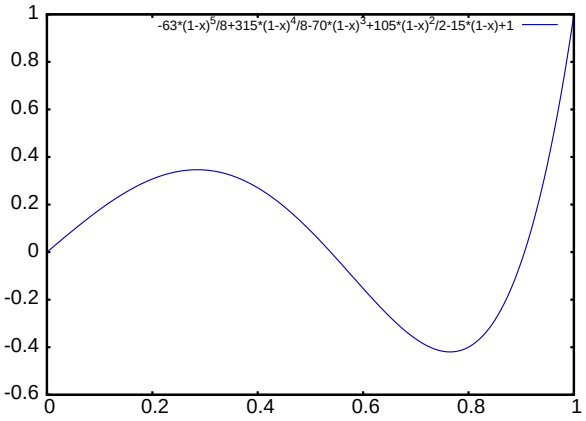
To plot expressions that involve the orthogonal polynomials, you must do two things:

1. Set the option variable `orthopoly_returns_intervals` to `false`,
2. Quote any calls to `orthopoly` functions.

If function calls aren't quoted, Maxima evaluates them to polynomials before plotting; consequently, the specialized floating point code doesn't get called. Here is an example of how to plot an expression that involves a Legendre polynomial.

```
(%i1) plot2d ('(legendre_p (5, x)), [x, 0, 1]), orthopoly_returns_intervals : false
```

```
(%o1)
```



The *entire* expression `legendre_p (5, x)` is quoted; this is different than just quoting the function name using `'legendre_p (5,` *x*`)`.

62.1.5 Miscellaneous Functions

The `orthopoly` package defines the Pochhammer symbol and a unit step function. `orthopoly` uses the Kronecker delta function and the unit step function in gradef statements.

To convert Pochhammer symbols into quotients of gamma functions, use `makegamma`.

```
(%i1) makegamma (pochhammer (x, n));
                          gamma(x + n)
(%o1)                     ------------
                            gamma(x)
(%i2) makegamma (pochhammer (1/2, 1/2));
                                1
(%o2)                           ---------
                                sqrt(%pi)
```

Derivatives of the pochhammer symbol are given in terms of the `psi` function.

```
(%i1) diff (pochhammer (x, n), x);
(%o1)             (x)  (psi (x + n) - psi (x))
                     n     0             0
(%i2) diff (pochhammer (x, n), n);
(%o2)                  (x)  psi (x + n)
                          n    0
```

You need to be careful with the expression in (%o1); the difference of the `psi` functions has poles when *x* `= -1, -2, .., -`*n*. These poles cancel with factors in `pochhammer (`*x*`,` *n*`)` making the derivative a degree *n* `- 1` polynomial when *n* is a positive integer.

The Pochhammer symbol is defined for negative orders through its representation as a quotient of gamma functions. Consider

```
(%i1) q : makegamma (pochhammer (x, n));
                          gamma(x + n)
```

```
(%o1)                        ------------
                               gamma(x)
(%i2) sublis ([x=11/3, n= -6], q);
                                  729
(%o2)                           - ----
                                  2240
```

Alternatively, we can get this result directly.

```
(%i1) pochhammer (11/3, -6);
                                  729
(%o1)                           - ----
                                  2240
```

The unit step function is left-continuous; thus

```
(%i1) [unit_step (-1/10), unit_step (0), unit_step (1/10)];
(%o1)                          [0, 0, 1]
```

If you need a unit step function that is neither left or right continuous at zero, define your own using signum; for example,

```
(%i1) xunit_step (x) := (1 + signum (x))/2$

(%i2) [xunit_step (-1/10), xunit_step (0), xunit_step (1/10)];
                                    1
(%o2)                          [0, -, 1]
                                    2
```

Do not redefine `unit_step` itself; some code in `orthopoly` requires that the unit step function be left-continuous.

62.1.6 Algorithms

Generally, `orthopoly` does symbolic evaluation by using a hypergeometic representation of the various orthogonal polynomials. The hypergeometic functions are evaluated using the (undocumented) functions `hypergeo11` and `hypergeo21`. The exceptions are the half-integer Bessel functions and the associated Legendre function of the second kind. The Bessel functions are evaluated using an explicit representation, while the associated Legendre function of the second kind is evaluated using recursion.

For floating point evaluation, we again convert most functions into a hypergeometic form; we evaluate the hypergeometic functions using forward recursion. Again, the exceptions are the half-integer Bessel functions and the associated Legendre function of the second kind. Numerically, the half-integer Bessel functions are evaluated using the SLATEC code, and the associated Legendre functions of the second kind is numerically evaluated using the same algorithm as its symbolic evaluation uses.

62.2 Definitions for orthogonal polynomials

assoc_legendre_p (*n*, *m*, *x*) Function

The associated Legendre function of the first kind.

Reference: Abramowitz and Stegun, equations 22.5.37, page 779, 8.6.6 (second equation), page 334, and 8.2.5, page 333.

assoc_legendre_q (*n*, *m*, *x*) Function

The associated Legendre function of the second kind.

Reference: Abramowitz and Stegun, equation 8.5.3 and 8.1.8.

chebyshev_t (*n*, *x*) Function

The Chebyshev function of the first kind.

Reference: Abramowitz and Stegun, equation 22.5.47, page 779.

chebyshev_u (*n*, *x*) Function

The Chebyshev function of the second kind.

Reference: Abramowitz and Stegun, equation 22.5.48, page 779.

gen_laguerre (*n*, *a*, *x*) Function

The generalized Laguerre polynomial.

Reference: Abramowitz and Stegun, equation 22.5.54, page 780.

hermite (*n*, *x*) Function

The Hermite polynomial.

Reference: Abramowitz and Stegun, equation 22.5.55, page 780.

intervalp (*e*) Function

Return true if the input is an interval and return false if it isn't.

jacobi_p (*n*, *a*, *b*, *x*) Function

The Jacobi polynomial.

The Jacobi polynomials are actually defined for all *a* and *b*; however, the Jacobi polynomial weight `(1 - x)^a (1 + x)^b` isn't integrable for *a* `<= -1` or *b* `<= -1`.

Reference: Abramowitz and Stegun, equation 22.5.42, page 779.

laguerre (*n*, *x*) Function

The Laguerre polynomial.

Reference: Abramowitz and Stegun, equations 22.5.16 and 22.5.54, page 780.

legendre_p (*n*, *x*) Function

The Legendre polynomial of the first kind.

Reference: Abramowitz and Stegun, equations 22.5.50 and 22.5.51, page 779.

legendre_q (*n*, *x*) Function

The Legendre polynomial of the first kind.

Reference: Abramowitz and Stegun, equations 8.5.3 and 8.1.8.

orthopoly_recur (*f, args*) Function

Returns a recursion relation for the orthogonal function family *f* with arguments *args*. The recursion is with respect to the polynomial degree.

```
(%i1) orthopoly_recur (legendre_p, [n, x]);
                (2 n - 1) P     (x) x + (1 - n) P     (x)
                           n - 1                 n - 2
(%o1)   P (x) = ----------------------------------------
         n                          n
```

The second argument to `orthopoly_recur` must be a list with the correct number of arguments for the function *f*; if it isn't, Maxima signals an error.

```
(%i1) orthopoly_recur (jacobi_p, [n, x]);

Function jacobi_p needs 4 arguments, instead it received 2
 -- an error.  Quitting.  To debug this try debugmode(true);
```

Additionally, when *f* isn't the name of one of the families of orthogonal polynomials, an error is signalled.

```
(%i1) orthopoly_recur (foo, [n, x]);

A recursion relation for foo isn't known to Maxima
 -- an error.  Quitting.  To debug this try debugmode(true);
```

orthopoly_returns_intervals Variable

Default value: `true`

When `orthopoly_returns_intervals` is true, floating point results are returned in the form `interval (`*c*`,` *r*`)`, where *c* is the center of an interval and *r* is its radius. The center can be a complex number; in that case, the interval is a disk in the complex plane.

orthopoly_weight (*f, args*) Function

Returns a three element list; the first element is the formula of the weight for the orthogonal polynomial family *f* with arguments given by the list *args*; the second and third elements give the lower and upper endpoints of the interval of orthogonality. For example,

```
(%i1) w : orthopoly_weight (hermite, [n, x]);
                              2
                           - x
(%o1)                  [%e    , - inf, inf]
(%i2) integrate (w[1] * hermite (3, x) * hermite (2, x), x, w[2], w[3]);
(%o2)                          0
```

The main variable of *f* must be a symbol; if it isn't, Maxima signals an error.

pochhammer (*n, x*) Function

The Pochhammer symbol. For nonnegative integers *n* with *n* `<= pochhammer_max_index`, the expression `pochhammer (`*x*`,` *n*`)` evaluates to the product *x* `(`*x* `+ 1) (`*x* `+ 2) ... (`*x* `+ n - 1)` when *n* `> 0` and to 1 when *n* `= 0`. For negative *n*, `pochhammer (`*x*`,` *n*`)` is defined as `(-1)^`*n* `/ pochhammer (1 -` *x*`, -`*n*`)`. Thus

```
(%i1) pochhammer (x, 3);
(%o1)                    x (x + 1) (x + 2)
(%i2) pochhammer (x, -3);
                                  1
(%o2)                - ------------------------
                       (1 - x) (2 - x) (3 - x)
```

To convert a Pochhammer symbol into a quotient of gamma functions, (see Abramowitz and Stegun, equation 6.1.22) use `makegamma`; for example

```
(%i1) makegamma (pochhammer (x, n));
                            gamma(x + n)
(%o1)                       ------------
                              gamma(x)
```

When n exceeds `pochhammer_max_index` or when n is symbolic, `pochhammer` returns a noun form.

```
(%i1) pochhammer (x, n);
(%o1)                           (x)
                                   n
```

pochhammer_max_index Variable

Default value: 100

`pochhammer (`n`, `x`)` expands to a product if and only if n `<= pochhammer_max_index`.

Examples:

```
(%i1) pochhammer (x, 3), pochhammer_max_index : 3;
(%o1)                    x (x + 1) (x + 2)
(%i2) pochhammer (x, 4), pochhammer_max_index : 3;
(%o2)                           (x)
                                   4
```

Reference: Abramowitz and Stegun, equation 6.1.16, page 256.

spherical_bessel_j (n, x) Function

The spherical Bessel function of the first kind.

Reference: Abramowitz and Stegun, equations 10.1.8, page 437 and 10.1.15, page 439.

spherical_bessel_y (n, x) Function

The spherical Bessel function of the second kind.

Reference: Abramowitz and Stegun, equations 10.1.9, page 437 and 10.1.15, page 439.

spherical_hankel1 (n, x) Function

The spherical hankel function of the first kind.

Reference: Abramowitz and Stegun, equation 10.1.36, page 439.

spherical_hankel2 (n, x) Function

The spherical hankel function of the second kind.

Reference: Abramowitz and Stegun, equation 10.1.17, page 439.

spherical_harmonic (*n*, *m*, *x*, *y*) Function

The spherical harmonic function.

Reference: Merzbacher 9.64.

unit_step (*x*) Function

The left-continuous unit step function; thus `unit_step (x)` vanishes for *x* `<= 0` and equals 1 for *x* `> 0`.

If you want a unit step function that takes on the value 1/2 at zero, use `(1 + signum (x))/2`.

ultraspherical (*n*, *a*, *x*) Function

The ultraspherical polynomial (also known the Gegenbauer polynomial).

Reference: Abramowitz and Stegun, equation 22.5.46, page 779.

63 plotdf

63.1 Introduction to plotdf

The function `plotdf` creates a plot of the direction field of a first-order Ordinary Differential Equation (ODE) or a system of two autonomous first-order ODE's.

To plot the direction field of a single ODE, the ODE must be written in the form:

$$\frac{dy}{dx} = F(x, y)$$

and the function F should be given as the argument for `plotdf`. The independent variable is always identified as x, and the dependent variable as y. Those two variables should not have any values assigned to them.

To plot the direction field of a set of two autonomous ODE's, they must be written in the form

$$\frac{dx}{dt} = G(x, y) \qquad \frac{dy}{dt} = F(x, y)$$

and the argument for `plotdf` should be a list with the two functions F and G, in any order.

If only one ODE is given, `plotdf` will implicitly admit `x=t`, and `G(x,y)=1`, transforming the non-autonomous equation into a system of two autonomous equations.

63.2 Definitions for plotdf

plotdf (*dydx,...options...*) Function
plotdf ([*dxdt,dydt*],*...options...*) Function

Displays a direction field in two dimensions x and y.

dydx, *dxdt* and *dydt* are expressions that depend on x and y. In addition to those two variables, the expressions can also depend on a set of parameters, with numerical values given with the `parameters` option (the option syntax is given below), or with an range of allowed values specified by a *sliders* option.

Several other options can be given within the command, or entered into the menu that will appear when the upper-left corner of the plot window is clicked. Integral curves can be obtained by clicking on the plot, or with the option `trajectory_at`. The direction of the integration can be controlled with the `direction` option, which can have values of `"forward"`, `"backward"` or `"both"`. The number of integration steps is given by `nsteps` and the time interval between them is set up with the `tstep` option. The Adams Moulton method is used for the integration; it is also possible to switch to an adaptive Runge-Kutta 4th order method.

Plot window menu:

The menu in the plot window has the following options: `"Zoom"`, will change the behavior of the mouse so that it will allow you to zoom in on a region of the plot by clicking with the left button. Each click near a point magnifies the plot, keeping the

center at the point where you clicked. Holding the SHIFT key while clicking, zooms out to the previous magnification. To resume computing trajectories when you click on a point, select "Integrate" from the menu.

The option "Config" in the menu can be used to change the ODE(s) in use and various other settings. After configuration changes are made, the menu option "Replot" should be selected, to activate the new settings. If a pair of coordinates are entered in the field "Trajectory at" in the "Config" dialog menu, and the "enter" key is pressed, a new integral curve will be shown, in addition to the ones already shown. When "Replot" is selected, only the last integral curve entered will be shown.

Holding the right mouse button down while the cursor is moved, can be used to drag the plot sideways or up and down. Additional parameters such as the number of steps, the initial value of t and the x and y centers and radii, may be set in the Config menu.

A copy of the plot can be printed to a Postscript printer, or saved as a postscript file, using the menu option "Save". To switch between printing and saving to a Postscript file, "Print Options" should be selected in the dialog window of "Config". After the settings in the "Save" dialog window are entered, "Save" must be selected in the first menu, to create the file or print the plot.

Plot options:

The `plotdf` command may include several commands, each command is a list of two or more items. The first item is the name of the option, and the remainder comprises the value or values assigned to the option.

The options which are recognized by `plotdf` are the following:

- Option: `tstep` defines the length of the increments on the independent variable t, used to compute an integral curve. If only one expression $dydx$ is given to `plotdf`, the x variable will be directly proportional to t: `x - xinitial = t - tinitial`.

  ```
  [tstep,0.01]
  ```

  The default value is 0.1.

- Option: `nsteps` defines the number of steps of length `tstep` that will be used for the independent variable, to compute an integral curve.

  ```
  [nsteps,500]
  ```

  The default value is 100.

- Option: `direction` defines the direction of the independent variable that will be followed to compute an integral curve. Possible values are `forward`, to make the independent variable increase `nsteps` times, with increments `tstep`, `backward`, to make the independent variable decrease, or `both` that will lead to an integral curve that extends `nsteps` forward, and `nsteps` backward. The keywords `right` and `left` can be used as synonyms for `forward` and `backward`.

  ```
  [direction,forward]
  ```

  The defaul value is `both`.

- Option: `tinitial` defines the initial value of variable t used to compute integral curves. Since the differential equations are autonomous, that setting will only appear in the plot of the curves as functions of t.

```
[tinitial,6.7]
```

The default value is 0.

- Option: `versus_t` is used to create a second plot window, with a plot of an integral curve, as two functions x, y, of the independent variable t. If `versus_t` is given any value different from 0, the second plot window will be displayed. The second plot window includes another menu, similar to the menu of the main plot window.

  ```
  [versus_t,1]
  ```

  The default value is 0.

- Option: `trajectory_at` defines the coordinates *xinitial* and *yinitial* for the starting point of an integral curve.

  ```
  [trajectory_at,0.1,3.2]
  ```

  The option is empty by default.

- Option `parameters` defines a list of parameters, and their numerical values, used in the definition of the differential equations. The name and values of the parameters must be given in a string with a comma-separated sequence of pairs `name=value`.

  ```
  [parameters,"k=1.1,m=2.5"]
  ```

- Option: `sliders` defines a list o parameters that will be changed interactively using slider buttons, and the range of variation of those parameters. The names and ranges of the parameters must be given in a string with a comma-separated sequence of elements `name=min:max`

  ```
  [sliders,"k=0:4,m=1:3"]
  ```

- Option: `xfun` defines a string with semi-colon-separated sequence of functions of x to be displayed, on top of the direction field. Those functions will be parsed by Tcl and not by Maxima.

  ```
  [xfun,"x^2;sin(x);exp(x)"]
  ```

- Option: `xradius` is half of the length of the range of values that will be shown in the x direction.

  ```
  [xradius,12.5]
  ```

  the default value is 10.

- Option: `yradius` is half of the length of the range of values that will be shown in the y direction.

  ```
  [yradius,15]
  ```

  the default value is 10.

- Option: `xcenter` is the x coordinate of the point at the center of the plot.

  ```
  [xcenter,3.45]
  ```

  The default value is 0.

- Option: `ycenter` is the y coordinate of the point at the center of the plot.

  ```
  [ycenter,4.5]
  ```

  The default value is 0.

- Option: `width` defines the width of the plot window, in pixels.

```
[width,800]
```

The default value is 500.

- Option: `height` defines the height of the plot window, in pixels.

  ```
  [width,600]
  ```

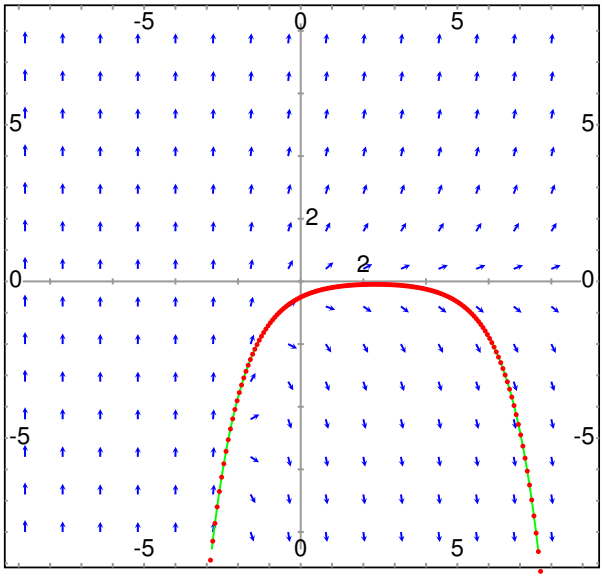
  The default value is 500.

Examples:

NOTE: Due to a bug in `openmath`, all commands that use it, in particular `plotdf`, must end with a semicolon and not with a dollar sign. The dollar sign might work in some of the graphical interfaces to Maxima, but to avoid problems we will use a semicolon in all the examples below.

- To show the direction field of the differential equation $y' = exp(-x) + y$ and the solution that goes through $(2, -0.1)$:

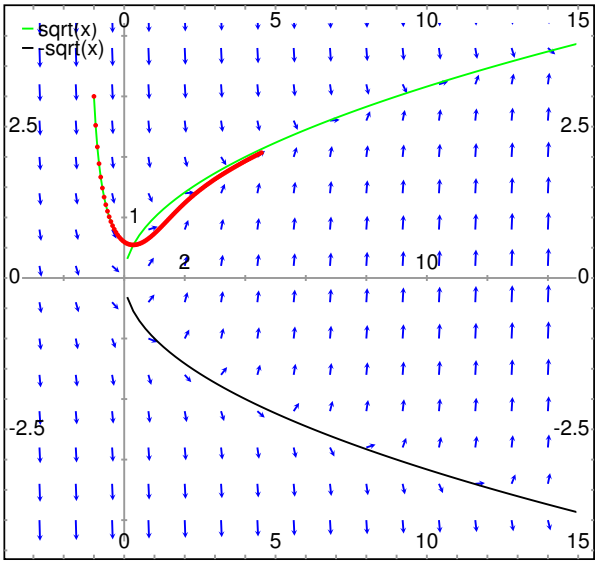
```
(%i1) load("plotdf")$

(%i2) plotdf(exp(-x)+y,[trajectory_at,2,-0.1]);
```



- To obtain the direction field for the equation $diff(y, x) = x - y^2$ and the solution with initial condition $y(-1) = 3$, we can use the command:

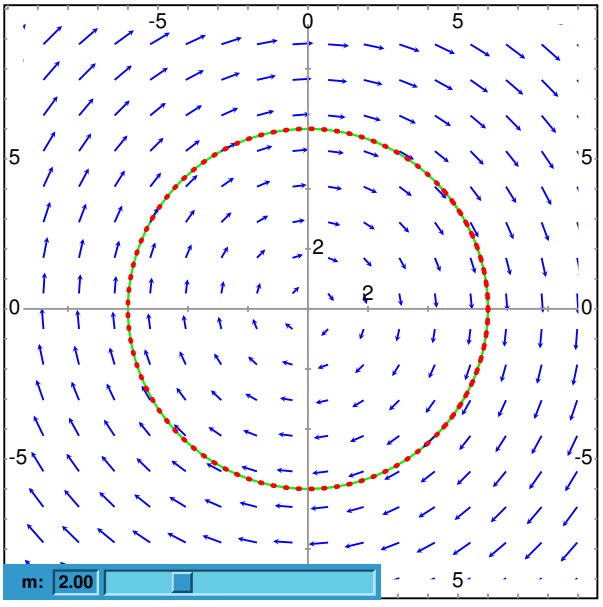
```
(%i3) plotdf(x-y^2,[xfun,"sqrt(x);-sqrt(x)"],
          [trajectory_at,-1,3], [direction,forward],
          [yradius,5],[xcenter,6]);
```

The graph also shows the function $y = sqrt(x)$.



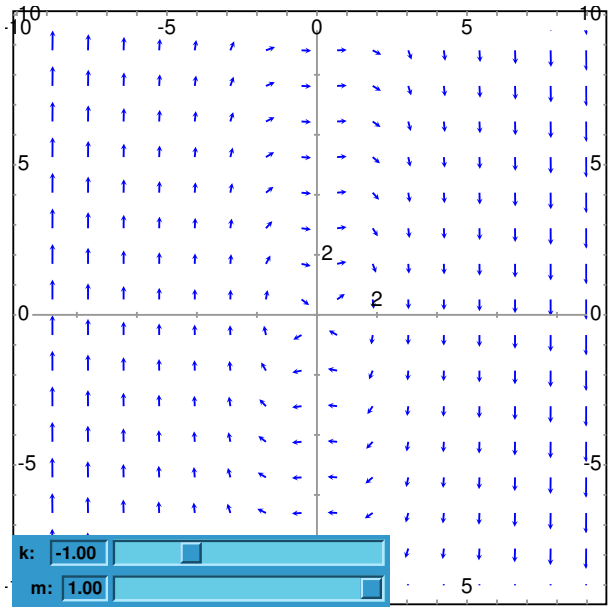
- The following example shows the direction field of a harmonic oscillator, defined by the two equations $dx/dt = y$ and $dy/dt = -k * x/m$, and the integral curve through $(x, y) = (6, 0)$, with a slider that will allow you to change the value of m interactively (k is fixed at 2):

```
(%i4) plotdf([y,-k*x/m],[parameters,"m=2,k=2"],
              [sliders,"m=1:5"], [trajectory_at,6,0]);
```



- To plot the direction field of the Duffing equation, $m*x''+c*x'+k*x+b*x^3 = 0$, we introduce the variable $y = x'$ and use:

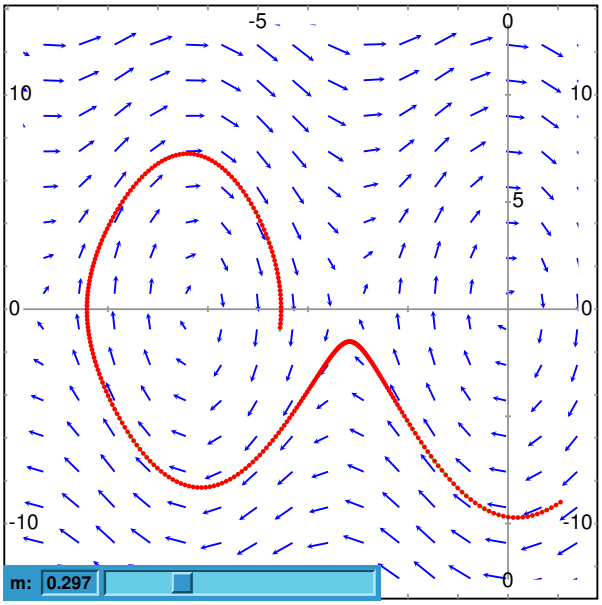
```
(%i5) plotdf([y,-(k*x + c*y + b*x^3)/m],
             [parameters,"k=-1,m=1.0,c=0,b=1"],
             [sliders,"k=-2:2,m=-1:1"],[tstep,0.1]);
```

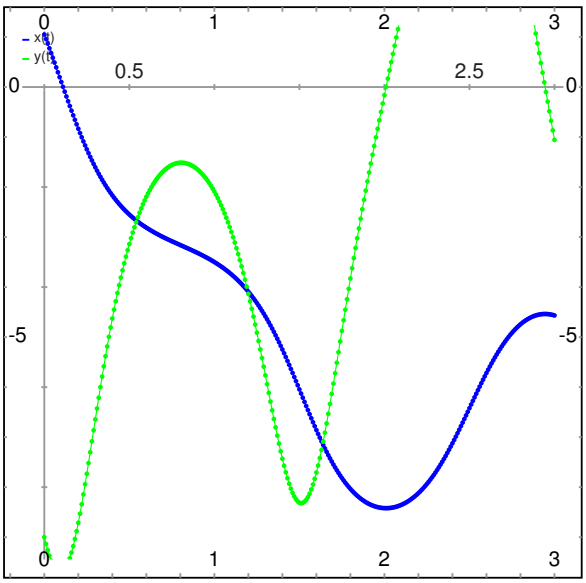


- The direction field for a damped pendulum, including the solution for the given initial conditions, with a slider that can be used to change the value of the mass m, and with a plot of the two state variables as a function of time:

```
(%i6) plotdf([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]);
```





To use this function write first `load("plotdf")`.

64 simplex

64.1 Introduction to simplex

`simplex` is a package for linear optimization using the simplex algorithm.

Example:

```
(%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 Definitions for simplex

epsilon_sx Option variable

Default value: `10^-8`

Epsilon used for numerical computations in `linear_program`.

See also: `linear_program`.

linear_program (*A*, *b*, *c*) Function

`linear_program` is an implementation of the simplex algorithm. `linear_program(A, b, c)` computes a vector *x* for which `c.x` is minimum possible among vectors for which `A.x = b` and `x >= 0`. Argument *A* is a matrix and arguments *b* and *c* are lists.

`linear_program` returns a list which contains the minimizing vector *x* and the minimum value `c.x`. If the problem is not bounded, it returns `"Problem not bounded!"` and if the problem is not feasible, it returns `"Problem not feasible!"`.

To use this function first load the `simplex` package with `load(simplex);`.

Example:

```
(%i2) A: matrix([1,1,-1,0], [2,-3,0,-1], [4,-5,0,0])$
(%i3) b: [1,1,6]$
(%i4) c: [1,-2,0,0]$
(%i5) linear_program(A, b, c);
                    13     19        3
(%o5)            [[--, 4, --, 0], - -]
                    2      2         2
```

See also: `minimize_sx`, `scale_sx`, and `epsilon_sx`.

maximize_sx (*obj*, *cond*, [*pos*]) Function

Maximizes linear objective function *obj* subject to some linear constraints *cond*. See `minimize_sx` for detailed description of arguments and return value.

See also: `minimize_sx`.

minimize_sx (*obj*, *cond*, [*pos*]) Function

Minimizes a linear objective function *obj* subject to some linear constraints *cond*. *cond* a list of linear equations or inequalities. In strict inequalities `>` is replaced by `>=` and `<` by `<=`. The optional argument *pos* is a list of decision variables which are assumed to be positive.

If the minimum exists, `minimize_sx` returns a list which contains the minimum value of the objective function and a list of decision variable values for which the minimum is attained. If the problem is not bounded, `minimize_sx` returns "Problem not bounded!" and if the problem is not feasible, it returns "Ploblem not feasible!".

The decision variables are not assumed to be nonegative by default. If all decision variables are nonegative, set `nonegative_sx` to `true`. If only some of decision variables are positive, list them in the optional argument *pos* (note that this is more efficient than adding constraints).

`minimize_sx` uses the simplex algorithm which is implemented in maxima `linear_program` function.

To use this function first load the `simplex` package with `load(simplex);`.

Examples:

```
(%i1) minimize_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!
```

See also: `maximize_sx`, `nonegative_sx`, `epsilon_sx`.

nonegative_sx Option variable

Default value: `false`

If `nonegative_sx` is true all decision variables to `minimize_sx` and `maximize_sx` are assumed to be positive.

See also: `minimize_sx`.

65 simplification

65.1 Introduction to simplification

The directory `maxima/share/simplification` contains several scripts which implement simplification rules and functions, and also some functions not related to simplification.

65.2 Definitions for simplification

65.2.1 Package absimp

The `absimp` package contains pattern-matching rules that extend the built-in simplification rules for the `abs` and `signum` functions. `absimp` respects relations established with the built-in `assume` function and by declarations such as `modedeclare (m, even, n, odd)` for even or odd integers.

`absimp` defines `unitramp` and `unitstep` functions in terms of `abs` and `signum`.

`load (absimp)` loads this package. `demo (absimp)` shows a demonstration of this package.

Examples:

```
(%i1) load (absimp)$
(%i2) (abs (x))^2;
                                      2
(%o2)                                 x
(%i3) diff (abs (x), x);
                                      x
(%o3)                                 ------
                                      abs(x)
(%i4) cosh (abs (x));
(%o4)                                 cosh(x)
```

65.2.2 Package facexp

The `facexp` package contains several related functions that provide the user with the ability to structure expressions by controlled expansion. This capability is especially useful when the expression contains variables that have physical meaning, because it is often true that the most economical form of such an expression can be obtained by fully expanding the expression with respect to those variables, and then factoring their coefficients. While it is true that this procedure is not difficult to carry out using standard Maxima functions, additional fine-tuning may also be desirable, and these finishing touches can be more difficult to apply.

The function `facsum` and its related forms provide a convenient means for controlling the structure of expressions in this way. Another function, `collectterms`, can be used to add two or more expressions that have already been simplified to this form, without resimplifying the whole expression again. This function may be useful when the expressions are very large.

`load (facexp)` loads this package. `demo (facexp)` shows a demonstration of this package.

facsum (*expr, arg_1, ..., arg_n*) Function

Returns a form of *expr* which depends on the arguments *arg_1*, ..., *arg_n*. The arguments can be any form suitable for `ratvars`, or they can be lists of such forms. If the arguments are not lists, then the form returned is fully expanded with respect to the arguments, and the coefficients of the arguments are factored. These coefficients are free of the arguments, except perhaps in a non-rational sense.

If any of the arguments are lists, then all such lists are combined into a single list, and instead of calling `factor` on the coefficients of the arguments, `facsum` calls itself on these coefficients, using this newly constructed single list as the new argument list for this recursive call. This process can be repeated to arbitrary depth by nesting the desired elements in lists.

It is possible that one may wish to `facsum` with respect to more complicated subexpressions, such as `log (x + y)`. Such arguments are also permissible. With no variable specification, for example `facsum (`*expr*`)`, the result returned is the same as that returned by `ratsimp (`*expr*`)`.

Occasionally the user may wish to obtain any of the above forms for expressions which are specified only by their leading operators. For example, one may wish to `facsum` with respect to all `log`'s. In this situation, one may include among the arguments either the specific `log`'s which are to be treated in this way, or alternatively, either the expression `operator (log)` or `'operator (log)`. If one wished to `facsum` the expression *expr* with respect to the operators *op_1*, ..., *op_n*, one would evaluate `facsum (`*expr*`, operator (`*op_1*`, ..., `*op_n*`))`. The `operator` form may also appear inside list arguments.

In addition, the setting of the switches `facsum_combine` and `nextlayerfactor` may affect the result of `facsum`.

nextlayerfactor Global variable

Default value: `false`

When `nextlayerfactor` is `true`, recursive calls of `facsum` are applied to the factors of the factored form of the coefficients of the arguments.

When `false`, `facsum` is applied to each coefficient as a whole whenever recusive calls to `facsum` occur.

Inclusion of the atom `nextlayerfactor` in the argument list of `facsum` has the effect of `nextlayerfactor: true`, but for the next level of the expression *only*. Since `nextlayerfactor` is always bound to either `true` or `false`, it must be presented single-quoted whenever it appears in the argument list of `facsum`.

facsum_combine Global variable

Default value: `true`

`facsum_combine` controls the form of the final result returned by `facsum` when its argument is a quotient of polynomials. If `facsum_combine` is `false` then the form will

be returned as a fully expanded sum as described above, but if `true`, then the expression returned is a ratio of polynomials, with each polynomial in the form described above.

The `true` setting of this switch is useful when one wants to `facsum` both the numerator and denominator of a rational expression, but does not want the denominator to be multiplied through the terms of the numerator.

factorfacsum (*expr, arg_1, ... arg_n*) Function

Returns a form of *expr* which is obtained by calling `facsum` on the factors of *expr* with *arg_1*, ... *arg_n* as arguments. If any of the factors of *expr* is raised to a power, both the factor and the exponent will be processed in this way.

collectterms (*arg_1, ..., arg_n*) Function

If several expressions have been simplified with `facsum`, `factorfacsum`, `factenexpand`, `facexpten` or `factorfacexpten`, and they are to be added together, it may be desirable to combine them using the function `collecterms`. `collecterms` can take as arguments all of the arguments that can be given to these other associated functions with the exception of `nextlayerfactor`, which has no effect on `collecterms`. The advantage of `collecterms` is that it returns a form similar to `facsum`, but since it is adding forms that have already been processed by `facsum`, it does not need to repeat that effort. This capability is especially useful when the expressions to be summed are very large.

65.2.3 Package functs

rempart (*expr, n*) Function

Removes part *n* from the expression *expr*.

If *n* is a list of the form `[`*l*`,` *m*`]` then parts *l* thru *m* are removed.

To use this function write first `load(functs)`.

wronskian ([*f_1, ..., f_n*], *x*) Function

Returns the Wronskian matrix of the functions *f_1*, ..., *f_n* in the variable *x*.

f_1, ..., *f_n* may be the names of user-defined functions, or expressions in the variable *x*.

The determinant of the Wronskian matrix is the Wronskian determinant of the set of functions. The functions are linearly dependent if this determinant is zero.

To use this function write first `load(functs)`.

tracematrix (*M*) Function

Returns the trace (sum of the diagonal elements) of matrix *M*.

To use this function write first `load(functs)`.

rational (z) Function

Multiplies numerator and denominator of *z* by the complex conjugate of denominator, thus rationalizing the denominator. Returns canonical rational expression (CRE) form if given one, else returns general form.

To use this function write first `load(functs)`.

logand (x,y) Function

Returns logical (bit-wise) "and" of arguments x and y.

To use this function write first `load(functs)`.

logor (x,y) Function

Returns logical (bit-wise) "or" of arguments x and y.

To use this function write first `load(functs)`.

logxor (x,y) Function

Returns logical (bit-wise) exclusive-or of arguments x and y.

To use this function write first `load(functs)`.

nonzeroandfreeof (*x*, *expr*) Function

Returns `true` if *expr* is nonzero and `freeof (`*x*`,` *expr*`)` returns `true`. Returns `false` otherwise.

To use this function write first `load(functs)`.

linear (*expr*, *x*) Function

When *expr* is an expression linear in variable *x*, `linear` returns *a*`*`*x* `+` *b* where *a* is nonzero, and *a* and *b* are free of *x*. Otherwise, `linear` returns *expr*.

To use this function write first `load(functs)`.

gcdivide (*p*, *q*) Function

When `takegcd` is `true`, `gcdivide` divides the polynomials *p* and *q* by their greatest common divisor and returns the ratio of the results.

When `takegcd` is `false`, `gcdivide` returns the ratio *p*/*q*.

To use this function write first `load(functs)`.

arithmetic (*a*, *d*, *n*) Function

Returns the *n*-th term of the arithmetic series *a*`,` *a* `+` *d*`,` *a* `+ 2*`*d*`, ...,` *a* `+ (`*n* `- 1)*`*d*.

To use this function write first `load(functs)`.

geometric (*a*, *r*, *n*) Function

Returns the *n*-th term of the geometric series *a*`,` *a*`*`*r*`,` *a*`*`*r*`^2, ...,` *a*`*`*r*`^(`*n* `- 1)`.

To use this function write first `load(functs)`.

harmonic (*a*, *b*, *c*, *n*) Function

Returns the *n*-th term of the harmonic series *a*/*b*, *a*/(*b* + *c*), *a*/(*b* + 2**c*), ..., *a*/(*b* + (*n* - 1)**c*).

To use this function write first `load(functs)`.

arithsum (*a*, *d*, *n*) Function

Returns the sum of the arithmetic series from 1 to *n*.

To use this function write first `load(functs)`.

geosum (*a*, *r*, *n*) Function

Returns the sum of the geometric series from 1 to *n*. If *n* is infinity (`inf`) then a sum is finite only if the absolute value of *r* is less than 1.

To use this function write first `load(functs)`.

gaussprob (*x*) Function

Returns the Gaussian probability function `%e^(-`*x*`^2/2) / sqrt(2*%pi)`.

To use this function write first `load(functs)`.

gd (*x*) Function

Returns the Gudermannian function `2 * atan(%e^`*x*` - %pi/2)`.

To use this function write first `load(functs)`.

agd (*x*) Function

Returns the inverse Gudermannian function `log (tan (%pi/4 + x/2)))`.

To use this function write first `load(functs)`.

vers (*x*) Function

Returns the versed sine `1 - cos (x)`.

To use this function write first `load(functs)`.

covers (*x*) Function

Returns the coversed sine `1 - sin (`*x*`)`.

To use this function write first `load(functs)`.

exsec (*x*) Function

Returns the exsecant `sec (`*x*`) - 1`.

To use this function write first `load(functs)`.

hav (*x*) Function

Returns the haversine `(1 - cos(x))/2`.

To use this function write first `load(functs)`.

combination (*n*, *r*) Function

Returns the number of combinations of *n* objects taken *r* at a time.

To use this function write first `load(functs)`.

permutation (*n*, *r*) Function

Returns the number of permutations of *r* objects selected from a set of *n* objects.

To use this function write first `load(functs)`.

65.2.4 Package ineq

The `ineq` package contains simplification rules for inequalities.

Example session:

```
(%i1) load(ineq)$
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
(%i2) a>=4;  /* a sample inequality */
(%o2)                                a >= 4
(%i3) (b>c)+%; /* add a second, strict inequality */
(%o3)                            b + a > c + 4
(%i4) 7*(x<y); /* multiply by a positive number */
(%o4)                                7 x < 7 y
(%i5) -2*(x>=3*z); /* multiply by a negative number */
(%o5)                            - 2 x <= - 6 z
(%i6) (1+a^2)*(1/(1+a^2)<=1); /* Maxima knows that 1+a^2 > 0 */
                                        2
(%o6)                              1 <= a  + 1
(%i7) assume(x>0)$ x*(2<3); /* assuming x>0 */
(%o7)                                2 x < 3 x
(%i8) a>=b; /* another inequality */
(%o8)                                a >= b
(%i9) 3+%; /* add something */
(%o9)                            a + 3 >= b + 3
(%i10) %-3; /* subtract it out */
(%o10)                               a >= b
(%i11) a>=c-b; /* yet another inequality */
(%o11)                             a >= c - b
(%i12) b+%; /* add b to both sides */
(%o12)                             b + a >= c
(%i13) %-c; /* subtract c from both sides */
(%o13)                          - c + b + a >= 0
(%i14) -%;  /* multiply by -1 */
(%o14)                            c - b - a <= 0
(%i15) (z-1)^2>-2*z; /* determining truth of assertion */
                                      2
(%o15)                          (z - 1)  > - 2 z
(%i16) expand(%)+2*z; /* expand this and add 2*z to both sides */
                                    2
```

```
(%o16)                              z  + 1 > 0
(%i17) %,pred;
(%o17)                                true
```

Be careful about using parentheses around the inequalities: when the user types in `(A > B) + (C = 5)` the result is `A + C > B + 5`, but `A > B + C = 5` is a syntax error, and `(A > B + C) = 5` is something else entirely.

Do `disprule (all)` to see a complete listing of the rule definitions.

The user will be queried if Maxima is unable to decide the sign of a quantity multiplying an inequality.

The most common mis-feature is illustrated by:

```
eq: a > b;
2*eq;
% - eq;
```

Another problem is 0 times an inequality; the default to have this turn into 0 has been left alone. However, if you type `X*`*some_inequality* and Maxima asks about the sign of `X` and you respond `zero` (or `z`), the program returns `X*`*some_inequality* and not use the information that `X` is 0. You should do `ev (%, x: 0)` in such a case, as the database will only be used for comparison purposes in decisions, and not for the purpose of evaluating `X`.

The user may note a slower response when this package is loaded, as the simplifier is forced to examine more rules than without the package, so you might wish to remove the rules after making use of them. Do `kill (rules)` to eliminate all of the rules (including any that you might have defined); or you may be more selective by killing only some of them; or use `remrule` on a specific rule.

Note that if you load this package after defining your own rules you will clobber your rules that have the same name. The rules in this package are: `*rule1`, ..., `*rule8`, `+rule1`, ..., `+rule18`, and you must enclose the rulename in quotes to refer to it, as in `remrule ("+", "+rule1")` to specifically remove the first rule on `"+"` or `disprule ("*rule2")` to display the definition of the second multiplicative rule.

65.2.5 Package rducon

reduce_consts (*expr*) Function

Replaces constant subexpressions of *expr* with constructed constant atoms, saving the definition of all these constructed constants in the list of equations `const_eqns`, and returning the modified *expr*. Those parts of *expr* are constant which return `true` when operated on by the function `constantp`. Hence, before invoking `reduce_consts`, one should do

```
declare ([objects to be given the constant property], constant)$
```

to set up a database of the constant quantities occurring in your expressions.

If you are planning to generate Fortran output after these symbolic calculations, one of the first code sections should be the calculation of all constants. To generate this code segment, do

```
map ('fortran, const_eqns)$
```

Variables besides `const_eqns` which affect `reduce_consts` are:

`const_prefix` (default value: `xx`) is the string of characters used to prefix all symbols generated by `reduce_consts` to represent constant subexpressions.

`const_counter` (default value: 1) is the integer index used to generate unique symbols to represent each constant subexpression found by `reduce_consts`.

`load (rducon)` loads this function. `demo (rducon)` shows a demonstration of this function.

65.2.6 Package scifac

gcfac (*expr*) Function

`gcfac` is a factoring function that attempts to apply the same heuristics which scientists apply in trying to make expressions simpler. `gcfac` is limited to monomial-type factoring. For a sum, `gcfac` does the following:

1. Factors over the integers.
2. Factors out the largest powers of terms occurring as coefficients, regardless of the complexity of the terms.
3. Uses (1) and (2) in factoring adjacent pairs of terms.
4. Repeatedly and recursively applies these techniques until the expression no longer changes.

Item (3) does not necessarily do an optimal job of pairwise factoring because of the combinatorially-difficult nature of finding which of all possible rearrangements of the pairs yields the most compact pair-factored result.

`load (scifac)` loads this function. `demo (scifac)` shows a demonstration of this function.

65.2.7 Package sqdnst

sqrtdenest (*expr*) Function

Denests `sqrt` of simple, numerical, binomial surds, where possible. E.g.

```
(%i1) load (sqdnst)$
(%i2) sqrt(sqrt(3)/2+1)/sqrt(11*sqrt(2)-12);
                                      sqrt(3)
                                  sqrt(------- + 1)
                                         2
(%o2)                           ---------------------
                                sqrt(11 sqrt(2) - 12)
(%i3) sqrtdenest(%);
                                     sqrt(3)   1
                                     ------- + -
                                        2      2
(%o3)                               -------------
                                       1/4    3/4
                                    3 2    - 2
```

Sometimes it helps to apply `sqrtdenest` more than once, on such as `(19601-13860 sqrt(2))^(7/4)`.

`load (sqdnst)` loads this function.

66 solve_rec

66.1 Introduction to solve_rec

`solve_rec` is a package for solving linear recurrences with polynomial coefficients.

A demo is available with `demo(solve_rec);`.

Example:

```
(%i1) load("solve_rec")$
(%i2) solve_rec((n+4)*s[n+2] + s[n+1] - (n+1)*s[n], s[n]);
                                    n
                  %k  (2 n + 3) (- 1)          %k
                    1                            2
(%o2)       s  = -------------------- + ---------------
             n     (n + 1) (n + 2)      (n + 1) (n + 2)
```

66.2 Definitions for solve_rec

closed_form (*expr*) Function

Tries to simplify all sums appearing in *expr* to a closed form.

`closed_form` uses Gosper and Zeilberger algorithms to simplify sums.

To use this function first load the `closed_form` package with `load(closed_form)`.

Example:

```
(%i1) load("closed_form")$
(%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) closed_form(%);
                                 n
                                4     n
(%o3)                           -- + 2
                                2
```

reduce_order (*rec, sol, var*) Function

Reduces the order of linear recurrence *rec* when a particular solution *sol* is known. The reduced reccurence can be used to get other solutions.

Example:

```
(%i3) rec: x[n+2] = x[n+1] + x[n]/n;
                                          x
                                           n
```

```
(%o3)                x      = x      + --
                      n + 2    n + 1   n
(%i4) solve_rec(rec, x[n]);
WARNING: found some hypergeometrical solutions!
(%o4)                      x  = %k  n
                            n     1
(%i5) reduce_order(rec, n, x[n]);
(%t5)                      x  = n %z
                            n       n

                              n - 1
                              ====
                              \
(%t6)                  %z  =   >     %u
                         n    /        %j
                              ====
                              %j = 0

(%o6)              (- n - 2) %u     - %u
                                n + 1     n
(%i6) solve_rec((n+2)*%u[n+1] + %u[n], %u[n]);
                                          n
                               %k  (- 1)
                                 1
(%o6)                   %u  = ----------
                          n    (n + 1)!
```

So the general solution is

$$\sum_{\%k_2=0}^{n-1} \frac{(-1)^n}{(n+1)!} + \%k_1^n$$

simplify_products Option variable

Default value: `true`

If `simplify_products` is `true`, `solve_rec` will try to simplify products in result.

See also: `solve_rec`.

solve_rec (*eqn, var,* [*init*]) Function

Solves for hypergeometrical solutions to linear recurrence *eqn* with polynomials coefficient in variable *var*. Optional arguments *init* are initial conditions.

`solve_rec` can solve linear recurrences with constant coefficients, finds hypergeometrical solutions to homogeneous linear recurrences with polynomial coefficients, rational

solutions to linear recurrences with polynomial coefficients and can solve Ricatti type recurrences.

Note that the running time of the algorithm used to find hypergeometrical solutions is exponential in the degree of the leading and trailing coefficient.

To use this function first load the `solve_rec` package with `load(solve_rec);`.

Example of linear recurrence with constant coefficients:

```
(%i2) solve_rec(a[n]=a[n-1]+a[n-2]+n/2^n, a[n]);
                          n          n
              (sqrt(5) - 1)  %k  (- 1)
                             1           n
(%o2) a  = ------------------------- - ----
       n                n                  n
                       2                5 2
                                                 n
                                    (sqrt(5) + 1)  %k
                                                   2    2
                                 + ------------------ - ----
                                             n             n
                                            2            5 2
```

Example of linear recurrence with polynomial coefficients:

```
(%i7) 2*x*(x+1)*y[x] - (x^2+3*x-2)*y[x+1] + (x-1)*y[x+2];
                            2
(%o7) (x - 1) y      - (x  + 3 x - 2) y      + 2 x (x + 1) y
                x + 2                    x + 1                 x
(%i8) solve_rec(%, y[x], y[1]=1, y[3]=3);
                              x
                           3 2    x!
(%o9)                 y  = ---- - --
                        x    4     2
```

Example of Ricatti type recurrence:

```
(%i2) x*y[x+1]*y[x] - y[x+1]/(x+2) + y[x]/(x-1) = 0;
                              y         y
                               x + 1     x
(%o2)         x y  y      - ------ + ----- = 0
                  x  x + 1   x + 2    x - 1
(%i3) solve_rec(%, y[x], y[3]=5)$
(%i4) ratsimp(minfactorial(factcomb(%)));
                                  3
                               30 x  - 30 x
(%o4) y  = - ------------------------------------------------
       x        6      5       4       3       2
              5 x  - 3 x  - 25 x  + 15 x  + 20 x  - 12 x - 1584
```

See also: `solve_rec_rat`, `simplify_products`, and `product_use_gamma`.

solve_rec_rat (*eqn, var,* [*init*]) Function

Solves for rational solutions to linear recurrences. See solve_rec for description of arguments.

To use this function first load the `solve_rec` package with `load(solve_rec);`.

Example:

```
(%i1) (x+4)*a[x+3] + (x+3)*a[x+2] - x*a[x+1] + (x^2-1)*a[x];
(%o1)  (x + 4) a      + (x + 3) a      - x a
                x + 3            x + 2      x + 1
                                                  2
                                             + (x  - 1) a
                                                        x
(%i2) solve_rec_rat(% = (x+2)/(x+1), a[x]);
                          1
(%o2)       a  = ---------------
             x   (x - 1) (x + 1)
```

See also: `solve_rec`.

product_use_gamma Option variable

Default value: `true`

When simplifying products, `solve_rec` introduces gamma function into the expression if `product_use_gamma` is `true`.

See also: `simplify_products`, `solve_rec`.

summand_to_rec (*summand, k, n*) Function
summand_to_rec (*summand*, [*k, lo, hi*], *n*) Function

Returns the recurrence sattisfied by the sum

```
 hi
====
\
 >     summand
/
====
k = lo
```

where summand is hypergeometrical in *k* and *n*. If *lo* and *hi* are omited, they are assumed to be `lo = -inf` and `hi = inf`.

To use this function first load the `closed_form` package with `load(closed_form)`.

Example:

```
(%i1) load("closed_form")$
(%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 stirling

67.1 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")`.

68 stringproc

68.1 Introduction to string processing

`stringproc.lisp` enlarges Maximas capabilities of working with strings.

Please note that for Maxima Version 5.9.1 you need a different file. For questions and bugs mail to van.nek at arcor.de .

Load `stringproc.lisp` by typing `load("stringproc");`.

In Maxima a string is easily constructed by typing "text". Note that Maxima-strings are no Lisp-strings and vice versa. Tests can be done with `stringp` respectively `lstringp`. If for some reasons you have a value, that is a Lisp-string, maybe when using Maxima-function sconcat, you can convert 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
```

All functions in `stringproc.lisp`, that return strings, return Maxima-strings.

Characters are introduced as Maxima-strings of length 1. Of course, these are no Lisp-characters. Tests can be done with `charp` (respectively `lcharp` and conversion from Lisp to Maxima with `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
```

Again, all functions in `stringproc.lisp`, that return characters, return Maxima-characters. Due to the fact, that the introduced characters are strings of length 1, you can use a lot of string functions also for characters. As seen, `supcase` is one example.

It is important to know, that the first character in a Maxima-string is at position 1. This is designed due to the fact that the first element in a Maxima-list is at position 1 too. See definitions of charat and charlist for examples.

In applications string-functions are often used when working with files. You will find some useful stream- and print-functions in `stringproc.lisp`. The following example shows some of the here introduced functions at work.

Example:

Let file contain Maxima console I/O, saved with 'Save Console to File' or with copy and paste. `extracti` then extracts the values of all input labels to a batchable file, which path is the return value. The batch process can directly be started with `batch(%)`. Note that `extracti` fails if at least one label is damaged, maybe due to erasing the `)`. Or if there are input lines from a batch process. In this case terminators are missing. It fails too, if there are some characters behind the terminators, maybe due to comment.

```
extracti(file):= block(
   [ s1: openr(file), ifile: sconc(file,".in"), line, nl: false ],
   s2: openw(ifile),

   while ( stringp(line: readline(s1)) ) do (
      if ssearch( sconc("(",inchar),line ) = 1 then (
         line: strim(" ",substring( line,ssearch(")",line)+1 )),
         printf( s2,"~a~%",line ),
         checklast(line) )
      else if nl then (
         line: strimr(" ",line),
         printf( s2,"~a~%",line ),
         checklast(line) )),

   close(s1), close(s2),
   ifile)$

checklast(line):= block(
   [ last: charat( line,slength(line) ) ],
   if cequal(last,";") or cequal(last,"$") then
      nl:false else nl:true )$
```

File 'C:\home\maxima\test.out':

```
(%i1) f(x):= sin(x)$
(%i2) diff(f(x),x);
(%o2)                                cos(x)
(%i3) df(x):= ''%;
(%o3)                           df(x) := cos(x)
(%i4) df(0);
(%o4)                                 1
```

Maxima:

```
(%i11) extracti("C:\\home\\maxima\\test.out");
(%o11)             C:\home\maxima\test.out.in
(%i12) batch(%);

batching #pC:/home/maxima/test.out.in
(%i13)                          f(x) := sin(x)
(%i14)                           diff(f(x), x)
(%o14)                              cos(x)
(%i15)                          df(x) := cos(x)
(%o15)                          df(x) := cos(x)
(%i16)                               df(0)
```

```
(%o16)                                1
```

68.2 Definitions for input and output

Example:

```
(%i1) s: openw("C:\\home\\file.txt");
(%o1)                 #<output stream C:\home\file.txt>
(%i2) control: "~2tAn atom: ~20t~a~%~2tand a list: ~20t~{~r ~}~%~2tand an integer:
(%i3) printf( s,control, 'true,[1,2,3],42 )$
(%o3)                                 false
(%i4) close(s);
(%o4)                                 true
(%i5) s: openr("C:\\home\\file.txt");
(%o5)                 #<input stream C:\home\file.txt>
(%i6) while stringp( tmp:readline(s) ) do print(tmp)$
  An atom:          true
  and a list:       one two three
  and an integer:   42
(%i7) close(s)$
```

close (*stream*) Function

Closes *stream* and returns `true` if *stream* had been open.

flength (*stream*) Function

Returns the number of elements in *stream*.

fposition (*stream*) Function
fposition (*stream*, *pos*) Function

Returns the current position in *stream*, if *pos* is not used. If *pos* is used, `fposition` sets the position in *stream*. *pos* has to be a positive number, the first element in *stream* is in position 1.

freshline () Function
freshline (*stream*) Function

Writes a new line to *stream*, if the position is not at the beginning of a line. `freshline` does not work properly with the streams `true` and `false`.

newline () Function
newline (*stream*) Function

Writes a new line to *stream*. `newline` does not work properly with the streams `true` and `false`. See `sprint` for an example of using `newline`.

opena (*file*) Function

Returns an output stream to *file*. If an existing file is opened, `opena` appends elements at the end of file.

openr (*file*) Function

Returns an input stream to *file*. If *file* does not exist, it will be created.

openw (*file*) Function

Returns an output stream to *file*. If *file* does not exist, it will be created. If an existing file is opened, `openw` destructively modifies *file*.

printf (*dest, string*) Function

printf (*dest, string, expr_1, ..., expr_n*) Function

`printf` is like FORMAT in Common Lisp. (From gcl.info: "format produces formatted output by outputting the characters of control-string string and observing that a tilde introduces a directive. The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies what kind of formatting is desired. Most directives use one or more elements of args to create their output.")

The following description and the examples may give an idea of using `printf`. See Lisp reference for more information. Note that there are some directives, which do not work in Maxima. For example, `~:[` fails. `printf` is designed with the intention, that `~s` is read as `~a`. Also note that the selection directive `~[` is zero-indexed.

```
~%       new line
~&       fresh line
~t       tab
~$       monetary
~d       decimal integer
~b       binary integer
~o       octal integer
~x       hexadecimal integer
~br      base-b integer
~r       spell an integer
~p       plural
~f       floating point
~e       scientific notation
~g       ~f or ~e, depending upon magnitude
~a       as printed by Maxima function print
~s       like ~a
~~       ~
~<       justification, ~> terminates
~(       case conversion, ~) terminates
~[       selection, ~] terminates
~{       iteration, ~} terminates
```

```
(%i1) printf( false, "~s ~a ~4f ~a ~@r",
"String",sym,bound,sqrt(8),144), bound = 1.234;
(%o1)                  String sym 1.23 2*sqrt(2) CXLIV
(%i2) printf( false,"~{~a ~}",["one",2,"THREE"] );
(%o2)                            one 2 THREE
(%i3) printf( true,"~{~{~9,1f ~}~%~}",mat ),
mat = args( matrix([1.1,2,3.33],[4,5,6],[7,8.88,9]) )$
      1.1       2.0       3.3
      4.0       5.0       6.0
```

```
      7.0       8.9       9.0
(%i4) control: "~:(~r~) bird~p ~[is~;are~] singing."$
(%i5) printf( false,control, n,n,if n=1 then 0 else 1 ), n=2;
(%o5)                    Two birds are singing.
```

If *dest* is a stream or `true`, then `printf` returns `false`. Otherwise, `printf` returns a string containing the output.

readline (*stream*) Function

Returns a string containing the characters from the current position in *stream* up to the end of the line or *false* if the end of the file is encountered.

sprint (*expr_1*, ..., *expr_n*) Function

Evaluates and displays its arguments one after the other 'on a line' starting at the leftmost position. The numbers are printed with the '-' right next to the number, and it disregards line length.

```
(%i1) for n:0 thru 16 do sprint( fib(n) )$
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

In xmaxima you might wish to add `,newline()`, if you prefer line breaking prior to printing. See `ascii` for an example.

68.3 Definitions for characters

alphacharp (*char*) Function

Returns `true` if *char* is an alphabetic character.

alphanumericp (*char*) Function

Returns `true` if *char* is an alphabetic character or a digit.

ascii (*int*) Function

Returns the character corresponding to the ASCII number *int*. (-1 < int < 256)

```
(%i1) for n from 0 thru 255 do ( tmp: ascii(n),
if alphacharp(tmp) then sprint(tmp) ), newline()$
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g
h i j k l m n o p q r s t u v w x y z
```

cequal (*char_1*, *char_2*) Function

Returns `true` if *char_1* and *char_2* are the same.

cequalignore (*char_1*, *char_2*) Function

Like `cequal` but ignores case.

cgreaterp (*char_1*, *char_2*) Function

Returns `true` if the ASCII number of *char_1* is greater than the number of *char_2*.

cgreaterpignore (*char_1*, *char_2*) Function
Like `cgreaterp` but ignores case.

charp (*obj*) Function
Returns `true` if *obj* is a Maxima-character. See introduction for example.

cint (*char*) Function
Returns the ASCII number of *char*.

clessp (*char_1*, *char_2*) Function
Returns `true` if the ASCII number of *char_1* is less than the number of *char_2*.

clesspignore (*char_1*, *char_2*) Function
Like `clessp` but ignores case.

constituent (*char*) Function
Returns `true` if *char* is a graphic character and not the space character. A graphic character is a character one can see, plus the space character. (`constituent` is defined by Paul Graham, ANSI Common Lisp, 1996, page 67.)

```
(%i1) for n from 0 thru 255 do ( tmp: ascii(n),
if constituent(tmp) then sprint(tmp) ), newline()$
! " # % ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B
C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _ ` a b c
d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

cunlisp (*lisp_char*) Function
Converts a Lisp-character into a Maxima-character. (You wont need it.)

digitcharp (*char*) Function
Returns `true` if *char* is a digit.

lcharp (*obj*) Function
Returns `true` if *obj* is a Lisp-character. (You wont need it.)

lowercasep (*char*) Function
Returns `true` if *char* is a lowercase character.

newline Variable
The character newline.

space Variable
The character space.

tab Variable
The character tab.

uppercasep (*char*) Function
Returns `true` if *char* is an uppercase character.

68.4 Definitions for strings

sunlisp (*lisp_string*) Function

Converts a Lisp-string into a Maxima-string. (In general you wont need it.)

lstringp (*obj*) Function

Returns `true` if *obj* is a Lisp-string. (In general you wont need it.)

stringp (*obj*) Function

Returns `true` if *obj* is a Maxima-string. See introduction for example.

charat (*string, n*) Function

Returns the *n*-th character of *string*. The first character in *string* is returned with *n* = 1.

```
(%i1) load("stringproc")$
(%i2) charat("Lisp",1);
(%o2)                           L
```

charlist (*string*) Function

Returns the list of all characters in *string*.

```
(%i1) load("stringproc")$
(%i2) charlist("Lisp");
(%o2)                      [L, i, s, p]
(%i3) %[1];
(%o3)                           L
```

parsetoken (*string*) Function

`parsetoken` converts the first token in *string* to the corresponding number or returns `false` if the number cannot be determined . The delimiter set for tokenizing is `{space, comma, semicolon, tab, newline}`.

```
(%i1) load("stringproc")$
(%i2) 2*parsetoken("1.234 5.678");
(%o2)                         2.468
```

For parsing you can also use function parse_string. See description in file 'share\contrib\eval_string.lisp'.

sconc (*expr_1, ..., expr_n*) Function

Evaluates its arguments and concatenates them into a string. `sconc` is like `sconcat` but returns a Maxima string.

```
(%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 (*string*) Function
Returns a copy of *string* as a new string.

sdowncase (*string*) Function
sdowncase (*string, start*) Function
sdowncase (*string, start, end*) Function
Like `supcase`, but uppercase characters are converted to lowercase.

sequal (*string_1, string_2*) Function
Returns `true` if *string_1* and *string_2* are the same length and contain the same characters.

sequalignore (*string_1, string_2*) Function
Like `sequal` but ignores case.

sexplode (*string*) Function
`sexplode` is an alias for function `charlist`.

simplode (*list*) Function
simplode (*list, delim*) Function
`simplode` takes a list of expressions and concatenates them into a string. If no delimiter *delim* is used, `simplode` is like `sconc` and uses no delimiter. *delim* can be any string.

```
(%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, string, pos*) Function
Returns a string that is a concatenation of `substring (`*string*`, 1, `*pos*` - 1)`, the string *seq* and `substring (`*string*`, `*pos*`)`. Note that the first character in *string* is in position 1.

```
(%i1) 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 (*string*) Function
sinvertcase (*string, start*) Function
sinvertcase (*string, start, end*) Function
Returns *string* except that each character from position *start* to *end* is inverted. If *end* is not given, all characters from start to the *end* of *string* are replaced.

```
(%i1) load("stringproc")$
(%i2) sinvertcase("sInvertCase");
(%o2)                      SiNVERTcASE
```

slength (*string*) Function

Returns the number of characters in *string*.

smake (*num*, *char*) Function

Returns a new string with a number of *num* characters *char*.

```
(%i1) load("stringproc")$
(%i2) smake(3,"w");
(%o2)                           www
```

smismatch (*string_1*, *string_2*) Function
smismatch (*string_1*, *string_2*, *test*) Function

Returns the position of the first character of *string_1* at which *string_1* and *string_2* differ or `false`. Default test function for matching is `sequal`. If `smismatch` should ignore case, use `sequalignore` as test.

```
(%i1) load("stringproc")$
(%i2) smismatch("seven","seventh");
(%o2)                            6
```

split (*string*) Function
split (*string*, *delim*) Function
split (*string*, *delim*, *multiple*) Function

Returns the list of all tokens in *string*. Each token is an unparsed string. `split` uses *delim* as delimiter. If *delim* is not given, the space character is the default delimiter. *multiple* is a boolean variable with `true` by default. Multiple delimiters are read as one. This is useful if tabs are saved as multiple space characters. If *multiple* is set to `false`, each delimiter is noted.

```
(%i1) 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 (*char*, *string*) Function

Returns the position of the first character in *string* which matches *char*. The first character in *string* is in position 1. For matching characters ignoring case see `ssearch`.

sremove (*seq*, *string*) Function
sremove (*seq*, *string*, *test*) Function
sremove (*seq*, *string*, *test*, *start*) Function
sremove (*seq*, *string*, *test*, *start*, *end*) Function

Returns a string like *string* but without all substrings matching *seq*. Default test function for matching is `sequal`. If `sremove` should ignore case while searching for *seq*, use `sequalignore` as test. Use *start* and *end* to limit searching. Note that the first character in *string* is in position 1.

```
(%i1) 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*, *string*) Function
sremovefirst (*seq*, *string*, *test*) Function
sremovefirst (*seq*, *string*, *test*, *start*) Function
sremovefirst (*seq*, *string*, *test*, *start*, *end*) Function

Like `sremove` except that only the first substring that matches `seq` is removed.

sreverse (*string*) Function

Returns a string with all the characters of *string* in reverse order.

ssearch (*seq*, *string*) Function
ssearch (*seq*, *string*, *test*) Function
ssearch (*seq*, *string*, *test*, *start*) Function
ssearch (*seq*, *string*, *test*, *start*, *end*) Function

Returns the position of the first substring of *string* that matches the string *seq*. Default test function for matching is `sequal`. If `ssearch` should ignore case, use `sequalignore` as test. Use *start* and *end* to limit searching. Note that the first character in *string* is in position 1.

```
(%i1) ssearch("~s","~{~S ~}~%",'sequalignore);
(%o1)                                  4
```

ssort (*string*) Function
ssort (*string*, *test*) Function

Returns a string that contains all characters from *string* in an order such there are no two successive characters *c* and *d* such that `test (`*c*`, `*d*`)` is `false` and `test (`*d*`, `*c*`)` is `true`. Default test function for sorting is *clessp*. The set of test functions is `{clessp, clesspignore, cgreaterp, cgreaterpignore, cequal, cequalignore}`.

```
(%i1) load("stringproc")$
(%i2) ssort("I don't like Mondays.");
(%o2)                     '.IMaddeiklnnoosty
(%i3) ssort("I don't like Mondays.",'cgreaterpignore);
(%o3)                  ytsoonnMlkIiedda.'
```

ssubst (*new*, *old*, *string*) Function
ssubst (*new*, *old*, *string*, *test*) Function
ssubst (*new*, *old*, *string*, *test*, *start*) Function
ssubst (*new*, *old*, *string*, *test*, *start*, *end*) Function

Returns a string like *string* except that all substrings matching *old* are replaced by *new*. *old* and *new* need not to be of the same length. Default test function for matching is `sequal`. If `ssubst` should ignore case while searching for old, use `sequalignore` as test. Use *start* and *end* to limit searching. Note that the first character in *string* is in position 1.

```
(%i1) 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 (*new*, *old*, *string*) Function
ssubstfirst (*new*, *old*, *string*, *test*) Function
ssubstfirst (*new*, *old*, *string*, *test*, *start*) Function
ssubstfirst (*new*, *old*, *string*, *test*, *start*, *end*) Function

Like `subst` except that only the first substring that matches *old* is replaced.

strim (*seq*,*string*) Function

Returns a string like *string*, but with all characters that appear in *seq* removed from both ends.

```
(%i1) load("stringproc")$
(%i2) "/* comment */"$
(%i3) strim(" /*",%);
(%o3)                          comment
(%i4) slength(%);
(%o4)                            7
```

striml (*seq*, *string*) Function

Like `strim` except that only the left end of *string* is trimmed.

strimr (*seq*, *string*) Function

Like `strim` except that only the right end of string is trimmed.

substring (*string*, *start*) Function
substring (*string*, *start*, *end*) Function

Returns the substring of *string* beginning at position *start* and ending at position *end*. The character at position *end* is not included. If *end* is not given, the substring contains the rest of the string. Note that the first character in *string* is in position 1.

```
(%i1) load("stringproc")$
(%i2) substring("substring",4);
(%o2)                          string
(%i3) substring(%,4,6);
(%o3)                            in
```

supcase (*string*) Function
supcase (*string*, *start*) Function
supcase (*string*, *start*, *end*) Function

Returns *string* except that lowercase characters from position *start* to *end* are replaced by the corresponding uppercase ones. If *end* is not given, all lowercase characters from *start* to the end of *string* are replaced.

```
(%i1) load("stringproc")$
(%i2) supcase("english",1,2);
(%o2)                          English
```

tokens (*string*) Function
tokens (*string*, *test*) Function

Returns a list of tokens, which have been extracted from *string*. The tokens are substrings whose characters satisfy a certain test function. If test is not given, *constituent* is used as the default test. `{constituent, alphacharp, digitcharp, lowercasep, uppercasep, charp, characterp, alphanumericp}` is the set of test functions. (The Lisp-version of `tokens` is written by Paul Graham. ANSI Common Lisp, 1996, page 67.)

```
(%i1) 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]
```

69 unit

69.1 Introduction to Units

The *unit* package enables the user to convert between arbitrary units and work with dimensions in equations. The functioning of this package is radically different from the original Maxima units package - whereas the original was a basic list of definitions, this package uses rulesets to allow the user to chose, on a per dimension basis, what unit final answers should be rendered in. It will separate units instead of intermixing them in the display, allowing the user to readily identify the units associated with a particular answer. It will allow a user to simplify an expression to its fundamental Base Units, as well as providing fine control over simplifying to derived units. Dimensional analysis is possible, and a variety of tools are available to manage conversion and simplification options. In addition to customizable automatic conversion, *units* also provides a traditional manual conversion option.

Note - when unit conversions are inexact Maxima will make approximations resulting in fractions. This is a consequence of the techniques used to simplify units. The messages warning of this type of substitution are disabled by default in the case of units (normally they are on) since this situation occurs frequently and the warnings clutter the output. (The existing state of ratprint is restored after unit conversions, so user changes to that setting will be preserved otherwise.) If the user needs this information for units, they can set *unitverbose:on* to reactivate the printing of warnings from the unit conversion process.

unit is included in Maxima in the share/contrib/unit directory. It obeys normal Maxima package loading conventions:

```
(%i1) load("unit")$
*****************************************************************
*                       Units version 0.50                      *
*          Definitions based on the NIST Reference on           *
*              Constants, Units, and Uncertainty                *
*       Conversion factors from various sources including       *
*                   NIST and the GNU units package              *
*****************************************************************

Redefining necessary functions...
WARNING: DEFUN/DEFMACRO: redefining function TOPLEVEL-MACSYMA-EVAL ...
WARNING: DEFUN/DEFMACRO: redefining function MSETCHK ...
WARNING: DEFUN/DEFMACRO: redefining function KILL1 ...
WARNING: DEFUN/DEFMACRO: redefining function NFORMAT ...
Initializing unit arrays...
Done.
```

The WARNING messages are expected and not a cause for concern - they indicate the *unit* package is redefining functions already defined in Maxima proper. This is necessary in order to properly handle units. The user should be aware that if other changes have been made to these functions by other packages those changes will be overwritten by this loading process.

The *unit.mac* file also loads a lisp file *unit-functions.lisp* which contains the lisp functions needed for the package.

Clifford Yapp is the primary author. He has received valuable assistance from Barton Willis of the University of Nebraska at Kearney (UNK), Robert Dodier, and other intrepid folk of the Maxima mailing list.

There are probably lots of bugs. Let me know. `float` and `numer` don't do what is expected.

TODO : dimension functionality, handling of temperature, showabbr and friends. Show examples with addition of quantities containing units.

69.2 Definitions for Units

setunits (*list*) Function

By default, the *unit* package does not use any derived dimensions, but will convert all units to the seven fundamental dimensions using MKS units.

```
(%i2) N;
                                    kg m
(%o2)                               ----
                                      2
                                     s
(%i3) dyn;
                                  1      kg m
(%o3)                           (------) (----)
                                 100000     2
                                           s
(%i4) g;
                                    1
(%o4)                              (----) (kg)
                                    1000
(%i5) centigram*inch/minutes^2;
                                   127        kg m
(%o5)                        (-------------) (----)
                              1800000000000     2
                                               s
```

In some cases this is the desired behavior. If the user wishes to use other units, this is achieved with the `setunits` command:

```
(%i6) setunits([centigram,inch,minute]);
(%o6)                                done
(%i7) N;
                              1800000000000   %in cg
(%o7)                        (-------------) (------)
                                   127            2
                                                %min
(%i8) dyn;
                                 18000000   %in cg
(%o8)                           (--------) (------)
```

```
                                      127          2
                                               %min
(%i9) g;
(%o9)                                (100) (cg)
(%i10) centigram*inch/minutes^2;
                                       %in cg
(%o10)                                 ------
                                           2
                                       %min
```

The setting of units is quite flexible. For example, if we want to get back to kilograms, meters, and seconds as defaults for those dimensions we can do:

```
(%i11) setunits([kg,m,s]);
(%o11)                                  done
(%i12) centigram*inch/minutes^2;
                                     127        kg m
(%o12)                         (-------------) (----)
                                1800000000000     2
                                                  s
```

Derived units are also handled by this command:

```
(%i17) setunits(N);
(%o17)                                  done
(%i18) N;
(%o18)                                    N
(%i19) dyn;
                                       1
(%o19)                               (------) (N)
                                     100000
(%i20) kg*m/s^2;
(%o20)                                    N
(%i21) centigram*inch/minutes^2;
                                      127
(%o21)                          (-------------) (N)
                                 1800000000000
```

Notice that the *unit* package recognized the non MKS combination of mass, length, and inverse time squared as a force, and converted it to Newtons. This is how Maxima works in general. If, for example, we prefer dyne to Newtons, we simply do the following:

```
(%i22) setunits(dyn);
(%o22)                                  done
(%i23) kg*m/s^2;
(%o23)                             (100000) (dyn)
(%i24) centigram*inch/minutes^2;
                                      127
(%o24)                            (--------) (dyn)
                                  18000000
```

To discontinue simplifying to any force, we use the uforget command:

```
(%i26) uforget(dyn);
```

```
(%o26)                                false
(%i27) kg*m/s^2;
                                      kg m
(%o27)                                ----
                                        2
                                       s
(%i28) centigram*inch/minutes^2;
                                  127        kg m
(%o28)                       (-------------) (----)
                              1800000000000     2
                                               s
```

This would have worked equally well with `uforget(N)` or `uforget(%force)`.

See also `uforget`. To use this function write first `load("unit")`.

uforget (*list*) Function

By default, the *unit* package converts all units to the seven fundamental dimensions using MKS units. This behavior can be changed with the `setunits` command. After that, the user can restore the default behavior for a particular dimension by means of the `uforget` command:

```
(%i13) setunits([centigram,inch,minute]);
(%o13)                                done
(%i14) centigram*inch/minutes^2;
                                     %in cg
(%o14)                               ------
                                        2
                                     %min
(%i15) uforget([cg,%in,%min]);
(%o15)                  [false, false, false]
(%i16) centigram*inch/minutes^2;
                                  127        kg m
(%o16)                       (-------------) (----)
                              1800000000000     2
                                               s
```

`uforget` operates on dimensions, not units, so any unit of a particular dimension will work. The dimension itself is also a legal argument.

See also `setunits`. To use this function write first `load("unit")`.

convert (*expr, list*) Function

When resetting the global environment is overkill, there is the `convert` command, which allows one time conversions. It can accept either a single argument or a list of units to use in conversion. When a convert operation is done, the normal global evaluation system is bypassed, in order to avoid the desired result being converted again. As a consequence, for inexact calculations "rat" warnings will be visible if the global environment controlling this behavior (`ratprint`) is true. This is also useful for spot-checking the accuracy of a global conversion. Another feature is `convert` will allow a user to do Base Dimension conversions even if the global environment is set to simplify to a Derived Dimension.

```
(%i2) kg*m/s^2;
                                      kg m
(%o2)                                 ----
                                        2
                                       s
(%i3) convert(kg*m/s^2,[g,km,s]);
                                      g km
(%o3)                                 ----
                                        2
                                       s
(%i4) convert(kg*m/s^2,[g,inch,minute]);

'rat' replaced 39.37007874015748 by 5000//127 = 39.37007874015748
                              18000000000   %in g
(%o4)                        (-----------) (-----)
                                  127           2
                                             %min
(%i5) convert(kg*m/s^2,[N]);
(%o5)                                  N
(%i6) convert(kg*m^2/s^2,[N]);
(%o6)                                 m N
(%i7) setunits([N,J]);
(%o7)                                done
(%i8) convert(kg*m^2/s^2,[N]);
(%o8)                                 m N
(%i9) convert(kg*m^2/s^2,[N,inch]);

'rat' replaced 39.37007874015748 by 5000//127 = 39.37007874015748
                                5000
(%o9)                          (----) (%in N)
                                127
(%i10) convert(kg*m^2/s^2,[J]);
(%o10)                                 J
(%i11) kg*m^2/s^2;
(%o11)                                 J
(%i12) setunits([g,inch,s]);
(%o12)                               done
(%i13) kg*m/s^2;
(%o13)                                 N
(%i14) uforget(N);
(%o14)                               false
(%i15) kg*m/s^2;
                               5000000   %in g
(%o15)                        (-------) (-----)
                                 127       2
                                          s
(%i16) convert(kg*m/s^2,[g,inch,s]);

'rat' replaced 39.37007874015748 by 5000//127 = 39.37007874015748
```

```
                                      5000000   %in g
(%o16)                               (-------) (-----)
                                        127       2
                                                 s
```

See also `setunits` and `uforget`. To use this function write first `load("unit")`.

usersetunits Optional variable

Default value: none

If a user wishes to have a default unit behavior other than that described, they can make use of *maxima-init.mac* and the *usersetunits* variable. The *unit* package will check on startup to see if this variable has been assigned a list. If it has, it will use setunits on that list and take the units from that list to be defaults. `uforget` will revert to the behavior defined by usersetunits over its own defaults. For example, if we have a *maxima-init.mac* file containing:

```
usersetunits : [N,J];
```

we would see the following behavior:

```
(%i1) load("unit")$
*************************************************************
*                      Units version 0.50                   *
*          Definitions based on the NIST Reference on       *
*              Constants, Units, and Uncertainty            *
*       Conversion factors from various sources including   *
*                   NIST and the GNU units package          *
*************************************************************

Redefining necessary functions...
WARNING: DEFUN/DEFMACRO: redefining function TOPLEVEL-MACSYMA-EVAL ...
WARNING: DEFUN/DEFMACRO: redefining function MSETCHK ...
WARNING: DEFUN/DEFMACRO: redefining function KILL1 ...
WARNING: DEFUN/DEFMACRO: redefining function NFORMAT ...
Initializing unit arrays...
Done.
User defaults found...
User defaults initialized.
(%i2) kg*m/s^2;
(%o2)                                  N
(%i3) kg*m^2/s^2;
(%o3)                                  J
(%i4) kg*m^3/s^2;
(%o4)                                J m
(%i5) kg*m*km/s^2;
(%o5)                            (1000) (J)
(%i6) setunits([dyn,eV]);
(%o6)                               done
(%i7) kg*m/s^2;
(%o7)                          (100000) (dyn)
(%i8) kg*m^2/s^2;
(%o8)                  (6241509596477042688) (eV)
```

```
(%i9) kg*m^3/s^2;
(%o9)                    (6241509596477042688) (eV m)
(%i10) kg*m*km/s^2;
(%o10)                   (62415095964770426880000) (eV)
(%i11) uforget([dyn,eV]);
(%o11)                             [false, false]
(%i12) kg*m/s^2;
(%o12)                                   N
(%i13) kg*m^2/s^2;
(%o13)                                   J
(%i14) kg*m^3/s^2;
(%o14)                                  J m
(%i15) kg*m*km/s^2;
(%o15)                               (1000) (J)
```

Without `usersetunits`, the initial inputs would have been converted to MKS, and uforget would have resulted in a return to MKS rules. Instead, the user preferences are respected in both cases. Notice these can still be overridden if desired. To completely eliminate this simplification - i.e. to have the user defaults reset to factory defaults - the `dontusedimension` command can be used. `uforget` can restore user settings again, but only if `usedimension` frees it for use. Alternately, `kill(usersetunits)` will completely remove all knowledge of the user defaults from the session. Here are some examples of how these various options work.

```
(%i2) kg*m/s^2;
(%o2)                                    N
(%i3) kg*m^2/s^2;
(%o3)                                    J
(%i4) setunits([dyn,eV]);
(%o4)                                 done
(%i5) kg*m/s^2;
(%o5)                              (100000) (dyn)
(%i6) kg*m^2/s^2;
(%o6)                         (6241509596477042688) (eV)
(%i7) uforget([dyn,eV]);
(%o7)                               [false, false]
(%i8) kg*m/s^2;
(%o8)                                    N
(%i9) kg*m^2/s^2;
(%o9)                                    J
(%i10) dontusedimension(N);
(%o10)                                [%force]
(%i11) dontusedimension(J);
(%o11)                              [%energy, %force]
(%i12) kg*m/s^2;
                                       kg m
(%o12)                                 ----
                                         2
                                        s
(%i13) kg*m^2/s^2;
```

```
                                        2
                                      kg m
(%o13)                                -----
                                        2
                                       s
(%i14) setunits([dyn,eV]);
(%o14)                                done
(%i15) kg*m/s^2;
                                      kg m
(%o15)                                ----
                                        2
                                       s
(%i16) kg*m^2/s^2;
                                         2
                                      kg m
(%o16)                                -----
                                        2
                                       s
(%i17) uforget([dyn,eV]);
(%o17)                          [false, false]
(%i18) kg*m/s^2;
                                      kg m
(%o18)                                ----
                                        2
                                       s
(%i19) kg*m^2/s^2;
                                         2
                                      kg m
(%o19)                                -----
                                        2
                                       s
(%i20) usedimension(N);
Done.  To have Maxima simplify to this dimension, use setunits([unit])
to select a unit.
(%o20)                                true
(%i21) usedimension(J);
Done.  To have Maxima simplify to this dimension, use setunits([unit])
to select a unit.
(%o21)                                true
(%i22) kg*m/s^2;
                                      kg m
(%o22)                                ----
                                        2
                                       s
(%i23) kg*m^2/s^2;
                                         2
                                      kg m
(%o23)                                -----
                                        2
```

```
                                          s
(%i24) setunits([dyn,eV]);
(%o24)                                done
(%i25) kg*m/s^2;
(%o25)                           (100000) (dyn)
(%i26) kg*m^2/s^2;
(%o26)                  (6241509596477042688) (eV)
(%i27) uforget([dyn,eV]);
(%o27)                            [false, false]
(%i28) kg*m/s^2;
(%o28)                                 N
(%i29) kg*m^2/s^2;
(%o29)                                 J
(%i30) kill(usersetunits);
(%o30)                                done
(%i31) uforget([dyn,eV]);
(%o31)                           [false, false]
(%i32) kg*m/s^2;
                                      kg m
(%o32)                                ----
                                        2
                                       s
(%i33) kg*m^2/s^2;
                                          2
                                      kg m
(%o33)                                -----
                                        2
                                       s
```

Unfortunately this wide variety of options is a little confusing at first, but once the user grows used to them they should find they have very full control over their working environment.

metricexpandall (x) Function

Rebuilds global unit lists automatically creating all desired metric units. x is a numerical argument which is used to specify how many metric prefixes the user wishes defined. The arguments are as follows, with each higher number defining all lower numbers' units:

```
            0 - none. Only base units
            1 - kilo, centi, milli
(default)   2 - giga, mega, kilo, hecto, deka, deci, centi, milli,
                micro, nano
            3 - peta, tera, giga, mega, kilo, hecto, deka, deci,
                centi, milli, micro, nano, pico, femto
            4 - all
```

Normally, Maxima will not define the full expansion since this results in a very large number of units, but `metricexpandall` can be used to rebuild the list in a more or less complete fashion. The relevant variable in the *unit.mac* file is *%unitexpand*.

%unitexpand Variable

Default value: `2`

This is the value supplied to `metricexpandall` during the initial loading of *unit*.

70 zeilberger

70.1 Introduction to zeilberger

`zeilberger` is a implementation of Zeilberger's algorithm for definite hypergeometric summation, and also Gosper's algorithm for indefinite hypergeometric summation.

`zeilberger` makes use of the "filtering" optimization method developed by Axel Riese.

`zeilberger` was developed by Fabrizio Caruso.

`load (zeilberger)` loads this package.

70.1.0.1 The indefinite summation problem

`zeilberger` implements Gosper's algorithm for indefinite hypergeometric summation. Given a hypergeometric term F_k in k we want to find its hypergeometric anti-difference, that is, a hypergeometric term f_k such that $F_k = f_(k+1) - f_k$.

70.1.0.2 The definite summation problem

`zeilberger` implements Zeilberger's algorithm for definite hypergeometric summation. Given a proper hypergeometric term (in n and k) $F_(n,k)$ and a positive integer d we want to find a d-th order linear recurrence with polynomial coefficients (in n) for $F_(n,k)$ and a rational function R in n and k such that

$a_0 F_(n,k) + ... + a_d F_(n+d), k = Delta_K(R(n,k)F_(n,k))$

where $Delta_k$ is the k-forward difference operator, i.e., $Delta_k(t_k) := t_(k+1) - t_k$.

70.1.1 Verbosity levels

There are also verbose versions of the commands which are called by adding one of the following prefixes:

`Summary` Just a summary at the end is shown

`Verbose` Some information in the intermidiate steps

`VeryVerbose`
More information

`Extra` Even more information including information on the linear system in Zeilberger's algorithm

For example: `GosperVerbose`, `parGosperVeryVerbose`, `ZeilbergerExtra`, `AntiDifferenceSummary`.

70.2 Definitions for zeilberger

AntiDifference (*F_k, k*) Function

Returns the hypergeometric anti-difference of *F_k*, if it exists. Otherwise `AntiDifference` returns `no_hyp_antidifference`.

Gosper (*F_k, k*) Function

Returns the rational certificate *R(k)* for *F_k*, that is, a rational function such that

$F_k = R(k+1)F_(k+1) - R(k)F_k$

if it exists. Otherwise, `Gosper` returns `no_hyp_sol`.

GosperSum (*F_k, k, a, b*) Function

Returns the summmation of *F_k* from $k = a$ to $k = b$ if *F_k* has a hypergeometric anti-difference. Otherwise, `GosperSum` returns `nongosper_summable`.

Examples:

```
(%i1) load (zeilberger);
(%o1)  /usr/share/maxima/share/contrib/Zeilberger/zeilberger.mac
(%i2) GosperSum ((-1)^k*k / (4*k^2 - 1), k, 1, n);

Dependent equations eliminated:  (1)
                            3       n + 1
                       (n + -) (- 1)
                            2               1
(%o2)                - ------------------ - -
                                  2         4
                       2 (4 (n + 1)  - 1)
(%i3) GosperSum (1 / (4*k^2 - 1), k, 1, n);
                                3
                            - n - -
                                2       1
(%o3)                    ------------ + -
                                2       2
                         4 (n + 1)  - 1
(%i4) GosperSum (x^k, k, 1, n);
                           n + 1
                          x          x
(%o4)                     ------ - -----
                          x - 1    x - 1
(%i5) GosperSum ((-1)^k*a! / (k!*(a - k)!), k, 1, n);
                                n + 1
                 a! (n + 1) (- 1)              a!
(%o5)       - ------------------------ - ----------
              a (- n + a - 1)! (n + 1)!   a (a - 1)!
(%i6) GosperSum (k*k!, k, 1, n);

Dependent equations eliminated:  (1)
(%o6)                      (n + 1)! - 1
```

```
(%i7) GosperSum ((k + 1)*k! / (k + 1)!, k, 1, n);
                    (n + 1) (n + 2) (n + 1)!
(%o7)               ------------------------ - 1
                              (n + 2)!
(%i8) GosperSum (1 / ((a - k)!*k!), k, 1, n);
(%o8)                    nonGosper_summable
```

parGosper (*F_{n,k}*, *k*, *n*, *d*) Function

Attempts to find a a *d*-th order recurrence for *F_{n,k}*.

The algorithm yields a sequence $[s_1, s_2, ..., s_m]$ of solutions. Each solution has the form

$[R(n,k), [a_0, a_1, ..., a_d]]$

`parGosper` returns `[]` if it fails to find a recurrence.

Zeilberger (*F_{n,k}*, *k*, *n*) Function

Attempts to compute the indefinite hypergeometric summation of *F_{n,k}*.

`Zeilberger` first invokes `Gosper`, and if that fails to find a solution, then invokes `parGosper` with order 1, 2, 3, ..., up to `MAX_ORD`. If Zeilberger finds a solution before reaching `MAX_ORD`, it stops and returns the solution.

The algorithms yields a sequence $[s_1, s_2, ..., s_m]$ of solutions. Each solution has the form

$[R(n,k), [a_0, a_1, ..., a_d]]$

`Zeilberger` returns `[]` if it fails to find a solution.

`Zeilberger` invokes `Gosper` only if `gosper_in_zeilberger` is `true`.

70.3 General global variables

MAX_ORD Global variable

Default value: 5

`MAX_ORD` is the maximum recurrence order attempted by `Zeilberger`.

simplified_output Global variable

Default value: `false`

When `simplified_output` is `true`, functions in the `zeilberger` package attempt further simplification of the solution.

linear_solver Global variable

Default value: `linsolve`

`linear_solver` names the solver which is used to solve the system of equations in Zeilberger's algorithm.

warnings Global variable

Default value: `true`

When `warnings` is `true`, functions in the `zeilberger` package print warning messages during execution.

gosper_in_zeilberger Global variable

Default value: `true`

When `gosper_in_zeilberger` is `true`, the `Zeilberger` function calls `Gosper` before calling `parGosper`. Otherwise, `Zeilberger` goes immediately to `parGosper`.

trivial_solutions Global variable

Default value: `true`

When `trivial_solutions` is `true`, `Zeilberger` returns solutions which have certificate equal to zero, or all coefficients equal to zero.

70.4 Variables related to the modular test

mod_test Global variable

Default value: `false`

When `mod_test` is `true`, `parGosper` executes a modular test for discarding systems with no solutions.

modular_linear_solver Global variable

Default value: `linsolve`

`modular_linear_solver` names the linear solver used by the modular test in `parGosper`.

ev_point Global variable

Default value: `big_primes[10]`

`ev_point` is the value at which the variable *n* is evaluated when executing the modular test in `parGosper`.

mod_big_prime Global variable

Default value: `big_primes[1]`

`mod_big_prime` is the modulus used by the modular test in `parGosper`.

mod_threshold Global variable

Default value: 4

`mod_threshold` is the greatest order for which the modular test in `parGosper` is attempted.

71 Índice de Funções e Variáveis

Apêndice A Índice de Funções e Variáveis

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z