Manual de Maxima

Versión 5.47post

Traducción española

Maxima es un sistema de cálculo simbólico escrito en Lisp.

Maxima desciende del sistema Macsyma, desarrollado en el MIT (Massachusetts Institute of Technology) entre los años 1968 y 1982 como parte del proyecto MAC. El MIT pasó una copia del código fuente al DOE (Department of Energy) en 1982, en una versión conocida como DOE-Macsyma. Una de estas copias fue mantenida por el Profesor William F. Schelter de la Universidad de Texas desde el año 1982 hasta su fallecimiento en 2001. En 1998 Schelter había obtenido del Departamento de Energía permiso para distribuir el código fuente de DOE-Macsyma bajo licencia GNU-GPL, iniciando en el año 2000 el proyecto Maxima en SourceForge con el fin de mantener y seguir desarrollando DOE-Macsyma, ahora con el nombre de Maxima.

Los usuarios de habla hispana disponen de una lista de correos en la que podrán participar para recabar información, proponer ideas y comentar sus experiencias con Maxima:
`https://lists.sourceforge.net/lists/listinfo/maxima-lang-es`

Nota de la traducción: Este manual es una traducción de la versión original en inglés. En la versión 5.25 se han introducido cambios sustanciales que afectan, fundamentalmente, a una reordenación de los contenidos, a la desaparición de algunas secciones y a la aparición de otras nuevas. Además, se han añadido introducciones a algunas secciones provenientes de la versión alemana; me ha parecido conveniente incorporarlas a la versión española por ser clarificadoras de algunos conceptos básicos de Maxima, aunque todavía no forman parte de la versión inglesa de referencia.

Mario Rodríguez Riotorto. (`mario ARROBA edu PUNTO xunta PUNTO es`)

Resumen del Contenido

Índice General

1 Introducción a Maxima

Se puede iniciar Maxima con el comando "maxima". Maxima desplegará alguna información importante acerca de la versión que se está usando y un prompt. Cada comando que vaya a ser ejecutado por Maxima debe terminar con un punto y coma. Para finalizar una sesión en Maxima se emplea el comando "quit();". A continuación se presenta un breve ejemplo de sesión:

```
[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 puede hacer consultas en la documentación. La instrucción `describe` mostrará información sobre una función o la lista de todas las funciones y variables que contengan el texto pasado a `describe`. La interrogación `?` (búsqueda exacta) y la doble interrogación `??` (búsqueda aproximada) son abreviaturas de la instrucción `describe`:

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

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

```
'false'.

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

Examples:

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

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

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

(%o1)                                true
```

Para usar posteriormente un resultado, se puede asignar dicho resultado a una variable o referirse a él por medio de la etiqueta asociada (`%i*` o `%o*`). Además, se puede utilizar `%` para referirse al último resultado obtenido.

```
(%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 manipula sin ningún problema números complejos y constantes numéricas:

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

Maxima puede hacer derivadas e integrales:

```
(%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 puede resolver sistemas de ecuaciones lineales y cúbicas:

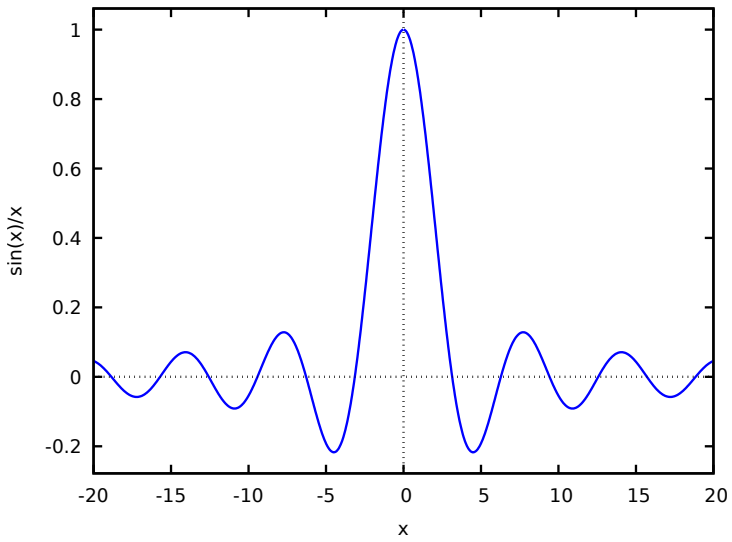
```
(%i1) linsolve ([3*x + 4*y = 7, 2*x + a*y = 13], [x, y]);
                        7 a - 52        25
(%o1)              [x = --------, y = -------]
                        3 a - 8       3 a - 8
(%i2) solve (x^3 - 3*x^2 + 5*x = 15, x);
(%o2)       [x = - sqrt(5) %i, x = sqrt(5) %i, x = 3]
```

Maxima puede resolver sistemas de ecuaciones no lineales. Tenga en cuenta que si usted no desea que el resultado sea impreso, puede finalizar el comando con *$* en vez de *;*.

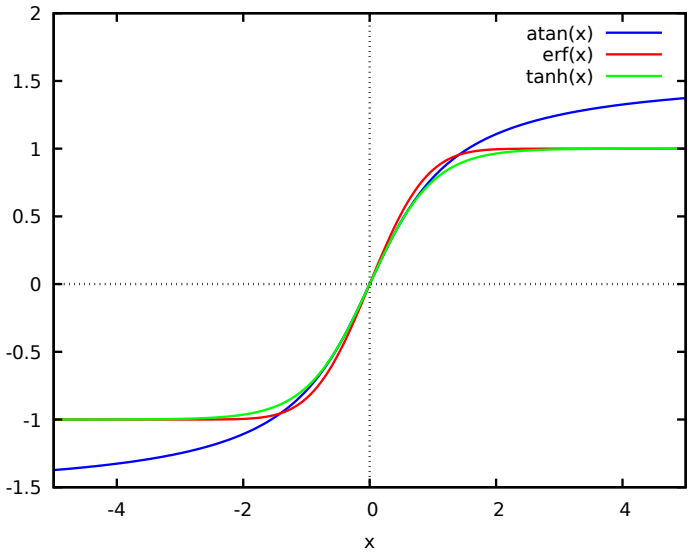
```
(%i1) eq_1: x^2 + 3*x*y + y^2 = 0$
(%i2) eq_2: 3*x + y = 1$
(%i3) solve ([eq_1, eq_2]);
              3 sqrt(5) + 7      sqrt(5) + 3
(%o3) [[y = - -------------, x = -----------],
                    2                2

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

Maxima puede generar gráficos de una o más funciones:

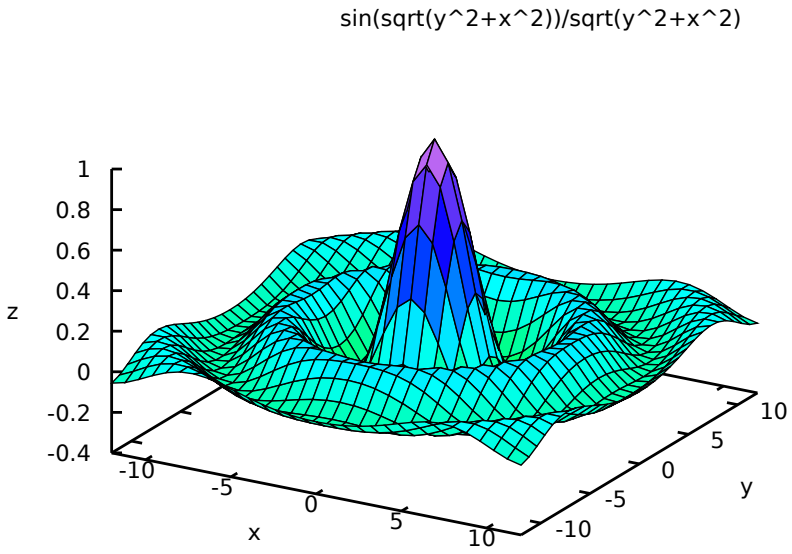
```
(%i1) plot2d (sin(x)/x, [x, -20, 20])$
```




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




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


sin(sqrt(y^2+x^2))/sqrt(y^2+x^2)
z
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
x
-10
-5
0
5
10
y
-10
-5
0
5
10

2 Detección e informe de fallos

2.1 Funciones y variables para la detección e informe de fallos

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

Ejecuta el conjunto de pruebas de Maxima. Los tests que producen las respuestas deseadas son considerados como "pruebas superadas", como los tests que no producen las respuestas deseadas, son marcados como fallos conocidos.

`run_testsuite` admite las siguientes opciones:

`display_all`
: Muestra todas las pruebas. Normalmente no se muestran las pruebas, a menos que produzcan fallos. (Su valor por defecto es `false`).

`display_known_bugs`
: Muestra las pruebas marcadas como fallos ya conocidos. (Su valor por defecto es `false`).

`tests`
: Esta es la lista de las pruebas que se deben ejecutar. Cada prueba se puede especificar, tanto mediante una cadena de texto como por un símbolo. Por defecto, todas las pruebas se ejecutan. El conjunto completo de pruebas está especificado en `testsuite_files`.

`time`
: Muestra información sobre tiempos de ejecución. Si vale `true`, se muestran los tiempos de los ficheros de prueba. Si vale `all`, se muestran los tiempos de cada prueba individual si `display_all` vale `true`. El valor por defecto es `false`, por lo que no se muestra información temporal alguna.

Por ejemplo, `run_testsuite(display_known_bugs = true, tests=[rtest5])` ejecuta la prueba `rtest5` y muestra si está marcada como fallo conocido.

`run_testsuite(display_all = true, tests=["rtest1", rtest1a])` ejecutará las pruebas `rtest1` y `rtest2`, mostrando cada una de ellas.

`run_testsuite` cambia el entorno de Maxima. Típicamente un script de test ejecuta `kill` para establecer un entorno conocido (llámese uno sin funciones ni variables definidas por el usuario) y entonces define una serie de funciones y variables apropiadas para el test.

`run_testsuite` retorna `done`.

`testsuite_files` [Variable opcional]

`testsuite_files` es el conjunto de tests a ejecutar por `run_testsuite`. Se trata de una lista con los nombres de los ficheros que contienen los tests a ejecutar. Si se sabe que alguno de los tests de un fichero falla, entonces en lugar de listar el nombre del fichero, se utiliza una lista que contiene el nombre del fichero y los números de los tests que fallan.

Por ejemplo, esta es una parte de los tests por defecto:

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

Con esto se especifica que el conjunto de tests está formado por los ficheros "rtest13s" y "rtest14", pero que "rtest14" contiene dos tests que se sabe que causan fallos, el 57 y el 63.

`bug_report` () [Función]

Imprime las versiones de Maxima y de Lisp y proporciona un enlace a la página web sobre informe de fallos del proyecto Maxima. La información respecto a las versiones es la misma que reporta la función `build_info`.

Cuando se informa sobre un fallo, es de gran ayuda que se copie la información relacionada con la versión de Maxima y de Lisp usada, dentro del propio informe.

`bug_report` retorna una cadena vacía `""`.

`build_info` () [Función]

Devuelve un resumen de los parÃ¡metros con los que se ha compilado Maxima en formato de estructura `defstruct`. Los campos de la estructura son: `version`, `timestamp`, `host`, `lisp_name` y `lisp_version`. Cuando `display2d` toma el valor `true`, la estructura se muestra como una pequeña tabla.

Véase también `bug_report`.

Ejemplos:

```
(%i1) build_info ();
(%o1)
Maxima version: "5.26.0_16_gb72c64c_dirty"
Maxima build date: "2012-01-29 12:29:04"
Host type: "i686-pc-linux-gnu"
Lisp implementation type: "CMU Common Lisp"
Lisp implementation version: "CVS release-19a 19a-release-20040728 + minimal debi
(%i2) x : build_info ()$
(%i3) x@version;
(%o3)               5.26.0_16_gb72c64c_dirty
(%i4) x@timestamp;
(%o4)                  2012-01-29 12:29:04
(%i5) x@host;
(%o5)                   i686-pc-linux-gnu
(%i6) x@lisp_name;
(%o6)                    CMU Common Lisp
(%i7) x@lisp_version;
(%o7)
    CVS release-19a 19a-release-20040728 + minimal debian patches
(%i8) x;
(%o8)
Maxima version: "5.26.0_16_gb72c64c_dirty"
Maxima build date: "2012-01-29 12:29:04"
Host type: "i686-pc-linux-gnu"
Lisp implementation type: "CMU Common Lisp"
Lisp implementation version: "CVS release-19a 19a-release-20040728 + minimal debi
```

3 Ayuda

3.1 Documentación

El manual en línea del usuario de Maxima puede ser visto en diferentes formas. Desde el prompt interactivo de Maxima, el manual de usuario es visto como texto plano por medio del comando `?` (i.e., la función `describe`). El manual de usuario también puede ser visto como hipertexto tipo `info` por medio del programa `info` y como una página web a través de cualquier navegador.

El comando `example` muestra ejemplos para muchas funciones Maxima. Por ejemplo:

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

produce:

```
(%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
```

y salidas adicionales.

3.2 Funciones y variables para la ayuda

`apropos` (*string*) [Función]

Busca los símbolos de Maxima en los cuales aparezca *cadena* en cualquier lugar dentro de su nombre. Así, `apropos (exp)` devuelve una lista con todas las variables y funciones que tengan `exp` formando parte de sus nombres, como `expand`, `exp` y `exponentialize`. De esta forma, si el usuario tan solo recuerda parte del nombre de algo, puede utilizar este comando para encontrar el resto del nombre. De manera semejante, también se puede hacer `apropos (tr_)` para encontrar una lista de muchas de las variables relacionadas con el traductor, buena parte de las cuales comienzan con `tr_`.

`apropos("")` devuelve una lista con todos los nombres de Maxima.

En caso de no encontrar información relevante, `apropos` devuelve la lista vacía `[]`.

Ejemplo:

Devuelve todos los símbolos de Maxima que contienen la subcadena `"gamma"` en su nombre:

```
(%i1) apropos("gamma");
(%o1) [%gamma, gamma, gammalim, gamma_expand, gamma_incomplete_lower,
gamma_incomplete, gamma_incomplete_generalized,
gamma_incomplete_regularized, Gamma, log_gamma, makegamma,
```

```
prefer_gamma_incomplete,
gamma_incomplete_generalized_regularized]
```

demo (*archivo*) [Función]

Evalua las expresiones Maxima contenidas en *archivo* y muestra los resultados. `demo` hace pausas después de evaluar cada expresión y continua después de que el usuario ingrese un retorno de carro. (Si se ejecuta en Xmaxima, `demo` puede que necesite un punto y coma `;` a continuación del retorno de carro.)

`demo` busca la lista de directorios `file_search_demo` para encontrar `archivo`. Si el archivo tiene el sufijo `dem`, el sufijo puede ser omitido. Ver también `file_search`.

`demo` evalua su argumento. `demo` retorna el nombre del archivo demostración.

Ejemplo:

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

La sentencia `describe(`*`string`*`)` equivale a `describe(`*`string`*`, exact)`.

La sentencia `describe(`*`string`*`, exact)` encuentra el elemento, si existe, cuyo título coincide exactamente con *string* (ignorando la diferencia entre mayúsculas y minúsculas).

La sentencia `describe(`*`string`*`, inexact)` encuentra todos los elementos documentados que contengan *string* en sus títulos.

Si hay más de una opción, Maxima preguntará al usuario para que seleccione las opciones que desee consultar.

La sentencia `? foo` (con espacio entre `?` y `foo`) equivale a `describe("foo", exact)`, mientras que `?? foo` equivale a `describe("foo", inexact)`.

`describe ("", inexact)` produce una lista de todos los temas documentados en el manual en línea.

`describe` no evalúa su argumento. La función `describe` devuelve `true` si encuentra la documentación solicitada y `false` en caso contrario.

Véase también `Documentación`.

Ejemplo:

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

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

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

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

En este ejemplo fueron seleccionadas las opciones 7 y 8 (la salida ha sido recortada, tal como indica `[...]`). Todas o ninguna de las opciones pueden ser seleccionadas escribiendo `all` o `none`, las cuales pueden ser abreviadas por `a` o `n`, respectivamente.

`example` (*topic*) [Función]
`example` () [Función]

`example` (*topic*) muestra algunos ejemplos sobre *topic*, el cual debe ser un símbolo o cadena de texto. Para ver ejemplos sobre operadores como `if`, `do` o `lambda` el argumento debe ser necesariamente una cadena de texto, como `example ("do")`. La función `example` no distingue entre minúsculas y mayúsculas. La mayor parte de ejemplos versan sobre funciones.

La sentencia `example ()` devuelve la lista de todos los ejemplos existentes.

El nombre del fichero que contine los ejemplos existentes se guarda en la variable global `manual_demo`, cuyo valor por defecto es `"manual.demo"`.

La función `example` no evalúa su argumento.

Ejemplos:

```
(%i1) example(append);
(%i2) append([x+y,0,-3.2],[2.5E+20,x])
(%o2)                     [y + x, 0, - 3.2, 2.5E+20, x]
(%o2)                                  done
(%i3) example("lambda");
(%i4) lambda([x,y,z],z^2+y^2+x^2)
                                              2    2    2
(%o4)                    lambda([x, y, z], z  + y  + x )
(%i5) %(1,2,a)
                                       2
(%o5)                                 a  + 5
(%i6) a+2+1
(%o6)                                  a + 3
(%o6)                                  done
(%i7) example("allROOTS");
(%i8) (1+2*x)^3 = 13.5*(1+x^5)
                                     3          5
(%o8)                       (2 x + 1)  = 13.5 (x  + 1)
(%i9) allroots(%)
(%o9) [x = .8296749902129361, x = - 1.015755543828121,
x = .9659625152196369 %i - .4069597231924075,
x = - .9659625152196369 %i - .4069597231924075, x = 1.0]
(%o9)                                  done
```

`manual_demo` [Variable opcional]

Valor por defecto: `"manual.demo"`

`manual_demo` especifica el nombre del fichero que contiene los ejemplo para la función `example`.

Véase `example`.

4 Línea de comandos

4.1 Introducción a la línea de comandos

Consola

Existen distintos interfaces para Maxima, tales como wxMaxima, Xmaxima, Imaxima y la consola o terminal de texto.

La consola trabaja en modo texto, al tiempo que para introducir instrucciones con ayuda de un menú y obtener resultados en modo gráfico es necesario instalar otros interfaces.

A lo largo de este manual se utilizará la consola, la cual está disponible en cualquiera de los sistemas operativos en los que trabaja Maxima. El usuario puede introducir todas las funciones de Maxima desde la consola; en el modo texto, los resultados se devolverán normalmente en un formato ASCII bidimensional, mientras que los gráficos necesitan de un programa adicional tal como Gnuplot.

Entrada, Evaluación, Simplificación y Salida

Desde que el usuario introduce una solicitud de cálculo hasta que obtiene el resultado en la consola, se desarrolla un proceso que consta de cuatro fases:

1. Bien sea desde un teclado o desde un fichero se lee una expresión que el analizador sintáctico se encargará de transformar en una cierta representación interna. En esta primera fase, se utilizan principalmente operadores tales como `"+"`, `"/"` o `"do"`.
2. La expresión leída por el analizador sintáctico es evaluada durante la segunda fase. Las variables se substituyen por sus valores y se ejecutan funciones tales como la derivación o la integración. El resultado de esta fase es una expresión evaluada.
3. La expresión evaluada de la fase anterior se simplifica en esta tercera fase, en la que una expresión tal como `a+a` se reduce a `2*a`, o `sin(%pi/2)` se simplifica a `1`.
4. Tras la tercera fase se dispone de una expresión que ha sido evaluada y posteriormente simplificada. Ya en la cuarta y última fase, se prepara el resultado para ser mostrado a través de la consola.

El usuario puede tomar el control en cualquiera de las cuatro fases recién descritas. En diferentes capítulos de este manual se detallan estas posibilidades, pero en éste se describen aquellas instrucciones relacionadas con las fases primera y cuarta, relacionadas con la entrada y salida a través de la consola. Los capítulos sobre Evaluación y Simplificación tratan de las otras dos fases intermedias.

Marcas

Maxima almacena todas las entradas con la marca `%i` seguida de un número entero en orden creciente, así como las salidas o resultados con la marca `%o` también seguida de un número de orden. Además, ciertas funciones utilizan la marca intermedia `%t`. Otras variables del sistema almacenan el último resultado devuelto por Maxima o la última entrada efectuada por el usuario. Los siguientes símbolos indican variables y funciones para la gestión de las marcas:

`--` `-`

```
%           %%          %th
inchar      linechar    outchar
linenum     nolabels
```

Listas informativas

Maxima gestiona listas informativas, cuyos nombres se guardan en la variable del sistema `infolists`. En el presente capítulo se describen las listas `labels`, `values` y `myoptions`. Los siguientes símbolos indican variables y funciones relacionadas con las listas informativas y variables opcionales.

```
infolists     labels        values
myoptions     optionset
```

Otras listas informativas, que se describirán en otros capítulos, son:

```
functions      arrays         macros
rules          aliases        dependencies
gradefs        props          let_rule_packages
structures
```

Borrado y reiniciación

A fin de establecer el contexto en el que trabaje Maxima, en el que no haya variables o funciones definidas, o en el que se eliminen hipótesis, propiedades o definiciones concretas, se dispone de las siguientes funciones:

```
kill     reset     reset_verbosely
```

Otras instrucciones

Se puede acceder a la documentación con los símbolos `?` y `??`. En caso de que se utilice `?` a modo de prefijo de un símbolo, éste se interpretará como símbolo de Lisp. Hay instrucciones para terminar una sesión de Maxima o para cambiar a una sesión de Lisp. También es posible conocer el tiempo que ha necesitado Maxima para realizar un cálculo. Para este tipo de cuestiones, Maxima dispone de las siguientes instrucciones:

```
?           ??
playback    prompt     showtime
quit        to_lisp
```

Las funciones `read` und `readonly` imprimen texto en la consola y leen la información introducida por el usuario.

Salida por consola

Antes de mostrar un resultado, se transforma su representación interna a otra externa. Por ejemplo, la representación interna de `sqrt(x)` es `x^(1/2)`, y ambos formatos pueden ser devueltos por Maxima en función del valor que guarde la variable opcional `sqrtdispflag`.

Los siguientes símbolos y variables opcionales controlan la salida de resultados por consola:

```
%edispflag          absboxchar       display2d
display_format_internal              exptdispflag
expt                nexpt            ibase
linel               lispdisp         negsumdispflag
```

```
obase              pfeformat        powerdisp
sqrtdispflag       stardisp         ttyoff
```

Con las siguientes funciones es posible formatear los resultados:

```
disp               display          dispterms
grind              ldisp            ldisplay
print
```

4.2 Funciones y variables para la línea de comandos

__ [Variable del sistema]

__ es la expresión de entrada que está siendo actualmente evaluada. Esto es, mientras se está evaluando una expresión de entrada, __ es igual a *expr*.

A __ se le asigna la expresión de entrada antes de que ésta sea simplificada o evaluada. Sin embargo, el valor de __ es simplificado, pero no evaluado, cuando su valor es mostrado en el terminal.

La variable __ es reconocida por `batch` y por `load`. Cuando un fichero es procesado por `batch`, la variable __ tiene el mismo significado que en el modo interactivo. Cuando un fichero es procesado por `load`, a la variable __ se le asigna la última expresión introducida, bien desde el modo interactivo, bien en un fichero por lotes; en ningún caso se le asigna a __ una expresión de entrada del fichero que está siendo procesado. En particular, si `load (`*`filename`*`)` es ejecutado desde el modo interactivo, entonces __ almacena la expresión `load (`*`filename`*`)` mientras el fichero está siendo procesado.

Véanse también _ y %.

Ejemplos:

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

_ [Variable del sistema]

El símbolo _ representa la última expresión de entrada (esto es, %i1, %i2, %i3, ...).

Al símbolo _ se le asigna la expresión de entrada antes de que ésta sea simplificada o evaluada. Sin embargo, el valor de _ se simplifica (pero no se evalúa) cuando se muestra en el terminal.

La variable _ es reconocida por `batch` y por `load`. Cuando un fichero es procesado por `batch`, la variable _ tiene el mismo significado que en el modo interactivo. Cuando un fichero es procesado por `load`, a la variable _ se le asigna la última expresión introducida, bien desde el modo interactivo, bien en un fichero por lotes; en ningún caso se le asigna a _ una expresión de entrada del fichero que está siendo procesado.

Véanse también __ y %.

Ejemplos:

```
(%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
```

% [Variable del sistema]

El símbolo % representa la expresión de salida (esto es, %o1, %o2, %o3, ...) más reciente calculada por Maxima, independientemente de que la haya mostrado o no.

La variable % es reconocida por `batch` y por `load`. Cuando un fichero es procesado por `batch`, la variable % tiene el mismo significado que en el modo interactivo. Cuando un fichero es procesado por `load`, a la variable % se le asigna la última expresión introducida, bien desde el modo interactivo, bien en un fichero por lotes; en ningún caso se le asigna a % una expresión de entrada del fichero que está siendo procesado.

Véanse también _, %% y %th.

%% [Variable del sistema]

En una sentencia compuesta, como `block`, `lambda` o `(`*`s_1`*`, ..., `*`s_n`*`)`, %% es el valor de la sentencia previa.

La variable %% no estÃ¡ definida cuando se utiliza en la primera sentencia, o fuera de una sentencia compuesta.

%% se puede utilizar con batch y load, manteniendo en ambos casos el mismo significado que en el modo interactivo.

Véase también %

Ejemplos:

Los siguientes dos ejemplos devuelven el mismo resultado.

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

Una sentencia compuesta puede contener otras sentencias compuestas. Independientemente de que una sentencia sea simple o compuesta, %% es el valor de la sentencia previa.

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

Dentro de una sentencia compuesta, el valor de %% puede inspeccionarse en un punto de interrupción que se abra ejecutando la función break. Por ejemplo, escribiendo %%; en el siguiente ejemplo se obtiene 42.

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

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

%th (*i*) [Función]

Es el valor de la expresión de la *i*-ésima salida anterior. Esto es, si la siguiente expresión a calcular es la salida *n*-ésima, %th (*m*) es la salida (*n* - *m*)-ésima.

%th es reconocido por batch y load, interpretándose de la misma manera que se acaba de indicar. Cuando un fichero es procesado por load, %th se refiere a los cálculos más recientes; %th no hace referencia a las expresiones de salida incluidas en el propio fichero que se está procesando.

Véanse también % y %%

Ejemplo:

%th es útil en ficheros batch para hacer referencia a grupos de resultados recién obtenidos. En este ejemplo se asigna a s la suma de los cinco resultados.

```
(%i1) 1;2;3;4;5;
(%o1)                           1
```

```
(%o2)                           2
(%o3)                           3
(%o4)                           4
(%o5)                           5
(%i6) block (s: 0, for i:1 thru 5 do s: s + %th(i), s);
(%o6)                          15
```

`?` [Símbolo especial]

Como prefijo de una función o nombre de variable, `?` significa que el nombre es de Lisp, no de Maxima. Por ejemplo, `?round` representa la función de Lisp `ROUND`. Véase `Lisp y Maxima` para más información.

La notación `? word` (un símbolo de interrogación seguido de una palabra y separados por un espacio) equivale a `describe ("word")`. El símbolo de interrogación debe escribirse al comienzo de la línea de entrada; en caso contrario no se reconoce como una solicitud de documentación.

`??` [Símbolo especial]

La notación `?? palabra` (`??` seguido de un espacio y una palabra) equivale a `describe("palabra", inexact)`. El símbolo de interrogación debe escribirse al comienzo de la línea de entrada; en caso contrario no se reconoce como una solicitud de documentación.

`inchar` [Variable opcional]

Valor por defecto: `%i`

La variable `inchar` es el prefijo de las etiquetas de las expresiones introducidas por el usuario. Maxima crea automáticamente una etiqueta para cada expresión de entrada concatenando `inchar` y `linenum`.

A `inchar` se le puede asignar cualquier símbolo o cadena, no necesariamente un caracácter sencillo. Puesto que internamente Maxima solo tiene en cuenta el primer carácter del prefijo, los prefijos `inchar`, `outchar` y `linechar` deben comenzar con caracteres diferentes; en caso contrario, sentencias como `kill(inlables)` pueden dar resultados inesperados.

Véase también `labels`.

Ejemplo:

```
(%i1) inchar: "input";
(%o1)                          input
(input2) expand((a+b)^3);
                     3        2      2      3
(%o2)               b  + 3 a b  + 3 a  b + a
(input3)
```

`infolists` [Variable del sistema]

Valor por defecto: `[]`

La variable `infolists` es una lista con los nombres de todas las listas que guardan información sobre Maxima. Estas son:

`labels` Todas las etiquetas `%i`, `%o` y `%t` con valores asignados.

`values` Todos los átomos que son variables de usuario, no opciones de Maxima creadas con `:` o `::`.

`functions`
Todas las funciones de usuario creadas con `:=` o `define`.

`arrays` Arreglos declarados y no declarados, creados por `:`, `::` o `:=`.

`macros` Cualquier macro definida por el usuario.

`myoptions`
Todas las opciones inicializadas por el usuario, independientemente de que posteriormente hayan sido devueltas a sus valores por defecto.

`rules` Reglas de patrones y simplificación definidas por el usuario, creadas con `tellsimp`, `tellsimpafter`, `defmatch` o `defrule`.

`aliases` Átomos que tienen un "alias" definido por el usuario, creado por las funciones `alias`, `ordergreat` o `orderless` o por haber declarado el átomo como `noun` (nombre) con `declare`.

`dependencies`
Átomos que tienen dependencias funcionales, creados por las funciones `depends` o `gradef`.

`gradefs` Funciones que tienen derivadas definidas por el usuario, creadas por la función `gradef`.

`props` Todos los átomos que tengan cualquier propiedad que no sea de las mencionadas hasta ahora, como las establecidas por `atvalue`, `matchdeclare`, etc., así como propiedades especificadas en la función `declare`.

`let_rule_packages`
Todos los paquetes de reglas `let` definidos por el usuario, junto con el paquete especial `default_let_rule_package`; `default_let_rule_package` es el nombre del paquete de reglas utilizado cuando no se use ningún otro especificado por el usuario.

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

Elimina todas las asignaciones (valor, función, arreglo o regla) hechas a los argumentos *a_1*, ..., *a_n*. Un argumento *a_k* puede ser un símbolo o el elemento de un array. Si *a_k* es elemento de un array, `kill` elimina la asignación hecha a este elemento sin afectar al resto del array.

Se reconocen varios argumentos especiales. Se pueden combinar diferentes clases de argumentos, como por ejemplo, `kill (inlabels, functions, allbut (foo, bar))`.

La instrucción `kill (labels)` borra todas las asignaciones asociadas a las etiquetas de entrada, de salida e intermedias creadas hasta el momento. La instrucción `kill (inlabels)` elimina únicamente las asignaciones de las etiquetas de entrada que comienzan con el valor actual de `inchar`. Del mismo modo, `kill (outlabels)` elimina únicamente las asignaciones de las etiquetas de salida que comienzan con el valor actual de `outchar`. Finalmente, `kill (linelabels)` elimina únicamente las asignaciones de las etiquetas de las expresiones intermedias que comienzan con el valor actual de `linechar`.

La instrucción `kill (n)`, siendo *n* un entero, elimina las asignaciones de las últimas *n* etiquetas, tanto de entrada como de salida.

La instrucción `kill ([m, n])` elimina las asignaciones hechas a las etiquetas de entrada y salida desde la *m* hasta la*n*.

La instrucción `kill (infolist)`, siendo *infolist* cualquier elemento de `infolists` (como `values`, `functions` o `arrays`), elimina todas las asignaciones hechas a los elementos de *infolist*. Véase también `infolists`.

La instrucción `kill (all)` elimina todas las asignaciones de todas las variables, pero no reinicia las variables globales a sus valores por defecto. Véase también `reset`.

La instrucción `kill (allbut (a_1, ..., a_n))` elimina las asignaciones hechas a todas las variables, excepto a *a_1*, ..., *a_n*; la instrucción `kill (allbut (infolist))` elimina todas las asignaciones, excepto las de los elementos de *infolist*, pudiendo ser *infolist* igual a `values`, `functions`, `arrays`, etc.

La memoria reservada para una asignación no se libera hasta que no se vacíen todos los símbolos asociados con esta asignación; por ejemplo, para liberar la memoria del valor de un símbolo es necesario eliminar tanto la asignación de la etiqueta de salida que muestra el resultado, como la del propio símbolo.

La función `kill` no evalua sus argumentos. El operador comilla-comilla, `''`, obliga a que se realice la evaluación.

La llamada `kill (symbol)` elimina todas las propiedades de *symbol*. Por el contrario, `remvalue`, `remfunction`, `remarray` y `remrule` eliminan propiedades específicas.

`kill` siempre devuelve `done`, incluso cuando alguno de sus argumentos carecía de asignación previa.

`labels (symbol)` [Función]
`labels` [Variable del sistema]

Retorna la lista de etiquetas de entrada, salida o de expresiones intermedias las cuales empiezan con *symbol*. Típicamente *symbol* es el valor de las variables `inchar`, `outchar` o `linechar`. El caractér de etiqueta puede ser pasado con o sin signo de porcentaje, así, por ejemplo, `i` y `%i` producen el mismo resultado.

Si ninguna etiqueta empieza con *symbol*, `labels` retorna a una lista vacía.

La función `labels` no evalua su argumento. El operador comilla-comilla, `''`, obliga a que se realice la evaluación. Por ejemplo, `labels (''inchar)` devuelve las etiquetas de entrada que empiezan con el caractér de etiqueta de entrada actual.

La variable `labels` es una lista de las etiquetas de entrada, salida y expresiones intermedias, incluyendo todas las etiquetas anteriores en el caso de que `inchar`, `outchar` o `linechar` hayan sido redefinidas.

Por defecto, Maxima muestra el resultado de cada expresión introducida por el usuario, asignando al resultado una etiqueta de salida. La salida (es decir el resultado) puede ser suprimida terminando la expresión de entrada con un `$` (signo de dólar) en vez de un `;` (punto y coma). En este caso, se crea la etiqueta de salida y se le asigna el resultado, aunque éste no se muestre; aún así, la etiqueta puede ser referenciada de la misma forma que se hace con aquéllas cuyos resultados sí son mostrados.

Véanse también: `%`, `%%` y `%th`.

Las etiquetas de expresiones intermedias pueden ser generadas por algunas funciones. El interruptor `programmode` controla si `solve` y algunas otras funciones generan etiquetas de expresiones intermedias en vez de retornar una lista de expresiones. Algunas otras funciones, tales como `ldisplay`, siempre generan etiquetas de expresiones intermedias.

Véase también: `inchar`, `outchar`, `linechar` y `infolists`.

`linechar` [Variable opcional]

Valor por defecto: `%t`

La variable `linechar` es el prefijo de las etiquetas que genera Maxima para expresiones intermedias. Cuando sea necesario, Maxima creará una etiqueta para cada expresión intermedia concatenando `linechar` y `linenum`.

A `linechar` se le puede asignar cualquier símbolo o cadena, no necesariamente un caracácter sencillo. Puesto que internamente Maxima solo tiene en cuenta el primer carácter del prefijo, los prefijos `inchar`, `outchar` y `linechar` deben comenzar con caracteres diferentes; en caso contrario, sentencias como `kill(inlables)` pueden dar resultados inesperados.

Las expresiones intermedias pueden ser mostradas o no. Véanse también `programmode` y `labels`.

`linenum` [Variable del sistema]

El número de la línea del par de expresiones de entrada y salida actuales.

`myoptions` [Variable del sistema]

Valor por defecto: `[]`

`myoptions` es la lista de todas las opciones que nunca fueron reconfiguradas por el usuario, aunque éstas hayan sido reconfiguradas a su valor por defecto.

`nolabels` [Variable opcional]

Valor por defecto: `false`

Cuando `nolabels` vale `true`, las etiquetas de entrada y salida (`%i` y `%o`, respectivamente) son mostradas, pero a éstas no se les asignan los resultados; además, las etiquetas no se incorporan a la lista `labels`. Puesto que a las etiquetas no se les asignan resultados, el colector de basura puede recuperar la memoria ocupada por éstos.

En el caso contrario, a las etiquetas de entrada y salida se les asignan los resultados correspondientes y son añadidas a la lista `labels`.

Las etiquetas de expresiones intermedias (`%t`) no se ven afectadas por la variable `nolabels`; independientemente de que `nolabels` valga `true` o `false`, a las etiquetas de expresiones intermedias se les asignan siempre valores, además de ser añadidas a la lista `labels`.

Véanse también `batch`, `batchload` y `labels`.

`optionset` [Variable opcional]

Valor por defecto: `false`

Cuando `optionset` tiene como valor `true`, Maxima imprime un mensaje cada vez que una opción de Maxima es reconfigurada. Esto es muy útil si el usuario duda con frecuencia de la correctitud de alguna opción y quiere estar seguro de la variable a la que él asignó un valor fue verdaramente una variable opción (o interruptor).

Ejemplo:

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

`outchar` [Variable opcional]

Valor por defecto: `%o`

La variable `outchar` es el prefijo de las etiquetas de las expresiones calculadas por Maxima. Maxima crea automáticamente una etiqueta para cada expresión calculada concatenando `outchar` y `linenum`.

A `outchar` se le puede asignar cualquier símbolo o cadena, no necesariamente un caracácter sencillo. Puesto que internamente Maxima solo tiene en cuenta el primer carácter del prefijo, los prefijos `inchar`, `outchar` y `linechar` deben comenzar con caracteres diferentes; en caso contrario, sentencias como `kill(inlables)` pueden dar resultados inesperados.

Véase también `labels`.

Ejemplo:

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

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

`playback` (*grind*) [Función]

Muestra las entradas, salidas y expresiones intermedias sin recalcularlas. `playback` sólo muestra las expresiones asociadas con etiquetas; cualquier otra salida (tal como texto impreso por `print` o `describe`, o mensajes de error) no es mostrada. Véase también: `labels`.

`playback` no evalua sus argumentos. El operador comilla-comilla, `''`, obliga a que se realice la evaluación. `playback` siempre devuelve `done`.

`playback ()` (sin argumentos) muestra todas las entradas, salidas y expresiones intermedias generadas hasta el momento. Una expresión de salida es mostrada incluso si ésta fue suprimida por el caracter de terminación `$`, cuando fue originalmente calculada.

`playback (`*n*`)` muestra las *n* expresiones más recientes. Cada entrada, salida y expresión intermedia cuenta como una.

`playback ([`*m*`, `*n*`])` muestra entradas, salidas y expresiones intermedias con los números desde *m* hasta *n*, ambos inclusive.

`playback ([`*m*`])` es equivalente a `playback ([`*m*`, `*m*`])`; esto usualmente imprime un par de expresiones de entrada y salida.

`playback (input)` muestra todas las expresiones de entrada generadas hasta el momento.

`playback (slow)` hace pausas entre expresiones y espera a que el usuario pulse la tecla `enter` para continuar. Esto es un comportamiento similar a `demo`.

`playback (slow)` es muy útil en conjunción con `save` o `stringout` cuando se crea un archivo secundario de almacenamiento con el objetivo de elegir cuidadosamente las expresiones realmente útiles.

`playback (time)` muestra el tiempo de computo por cada expresión.

`playback (grind)` muestra las expresiones de entrada en el mismo formato como la función `grind`. Las expresiones de salida no se ven afectadas por la opción `grind`. Vea `grind`. Los argumentos pueden ser combinados, por ejemplo, `playback ([5, 10], grind, time, slow)`.

`prompt` [Variable opcional]

Valor por defecto: `_`

`prompt` es el símbolo del prompt de la función `demo`, del modo `playback (slow)` y del bucle de interrupción de Maxima (el que se invoca con `break`).

`quit` () [Función]

Termina una sesión de Maxima. Nótese que la función debe ser invocada como `quit();` o `quit()$`, no como `quit`.

Para parar un cálculo muy demorado pulse `Control-C`. La acción por defecto es retornar a prompt de Maxima. Si `*debugger-hook*` tiene como valor `nil`, pulsar `Control-C` abrirá el depurador de Lisp. Vea también: `debugging`.

`read` (*expr_1*, ..., *expr_n*) [Función]

Imprime *expr_1*, ..., *expr_n* y a continuación lee una expresión desde la consola y devuelve la expresión evaluada. La expresión termina con un punto y coma `;` o con el símbolo de dólar `$`.

Véase también `readonly`.

```
(%i1) foo: 42$
(%i2) foo: read ("foo vale", foo, " -- nuevo valor.")$
foo vale 42  -- nuevo valor.
(a+b)^3;
(%i3) foo;
                                        3
(%o3)                          (b + a)
```

`readonly` (*expr_1*, ..., *expr_n*) [Función]

Imprime *expr_1*, ..., *expr_n* y a continuación lee una expresión desde la consola y devuelve la expresión sin evaluar. La expresión termina con un punto y coma `;` o con el símbolo de dólar `$`.

```
(%i1) aa: 7$
(%i2) foo: readonly ("Introducir expresion:");
Introducir expresion:
2^aa;
                                     aa
(%o2)                               2
(%i3) foo: read ("Introducir expresion:");
Introducir expresion:
2^aa;
(%o3)                               128
```

Véase también `read`.

`reset` () [Función]

Reconfigura muchas variables y opciones globales y algunas otras variables a sus valores por defecto.

`reset` procesa las variables que se encuentran en la lista Lisp `*variable-initial-values*`. La macro Lisp `defmvar` pone las variables en ésta lista (entre otras acciones). Muchas, pero no todas, las variables y opciones globales son definidas por `defmvar`, y algunas variables definidas por `defmvar` no son ni variables ni opciones globales.

`showtime` [Variable opcional]

Valor por defecto: `false`

Cuando `showtime` tiene como valor `true`, el tiempo de cálculo y el tiempo de retardo se imprimen junto con la salida de cada expresión.

El tiempo de cálculo se almacena siempre, de manera que `time` y `playback` puedan mostrar el tiempo de cálculo incluso cuando `showtime` vale `false`.

Véase también `timer`.

`to_lisp` () [Function]

Entra en el intérprete Lisp bajo Maxima. `(to-maxima)` retorna de nuevo a Maxima.

Ejemplo:

Define una función y entra en el nivel Lisp. La definición se consulta en la lista de propiedades, luego se extrae la definición de la función, se factoriza y almacena el

resultado en la variable `$result`. Esta variable se puede utilizar luego una vez se haya vuelto al nivel de Maxima.

```
(%i1) f(x):=x^2+x;
                                          2
(%o1)                            f(x) := x  + x
(%i2) to_lisp();
Type (to-maxima) to restart, ($quit) to quit Maxima.

MAXIMA> (symbol-plist '$f)
(MPROPS (NIL MEXPR ((LAMBDA) ((MLIST) $X) ((MPLUS) ((MEXPT) $X 2) $X))))
MAXIMA> (setq $result ($factor (caddr (mget '$f 'mexpr))))
((MTIMES SIMP FACTORED) $X ((MPLUS SIMP IRREDUCIBLE) 1 $X))
MAXIMA> (to-maxima)
Returning to Maxima
(%o2)                                 true
(%i3) result;
(%o3)                               x (x + 1)
```

`values` [Variable del sistema]

Valor inicial: `[]`

`values` es una lista de todas las variables que el usuario ha creado (no incluye las opciones de Maxima ni los interruptores). La lista comprende los símbolos a los que se ha asignado un valor mediante `:` o `::`.

Si el valor de una variable se borra con cualquiera de las instrucciones `kill`, `remove` o `remvalue`, dicha variable desaparece de la lista `values`.

Véase `functions` para una lista de funciones definidas por el usuario.

Ejemplos:

Primero, `values` muestra los símbolos `a`, `b` y `c`, pero no `d`, pues no tiene valor asignado, ni la función de usuario `f`. Luego los valores de las variables se borran y `values` queda como una lista vacía.

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

4.3 Funciones y variables para la impresión

`%edispflag` [Variable opcional]

Valor por defecto: `false`

Si `%edispflag` vale `true`, Maxima muestra `%e` elevado a un exponente negativo como un cociente. Por ejemplo, `%e^-x` se muestra como `1/%e^x`. Véase también `exptdispflag`.

Ejemplo:

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

`absboxchar` [Variable opcional]

Valor por defecto: `!`

La variable `absboxchar` es el carácter utilizado para representar el valor absoluto de una expresión que ocupa más de una línea de altura.

Ejemplo:

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

`disp` (*expr_1*, *expr_2*, ...) [Función]

Es como `display` pero sólo se muestran los valores de los argumentos, no las ecuaciones. Es útil para argumentos complicados que no tienen nombre o en situaciones en las que solamente es de interés el valor del argumento pero no su nombre.

Véanse también `ldisp` y `print`.

Ejemplos:

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

                                      2
                                  x - x

                              .8414709848078965

(%o3)                              done
```

`display` (*expr_1*, *expr_2*, ...) [Función]

Muestra las ecuaciones cuyos miembros izquierdos son *expr_i* sin evaluar y cuyos miembros derechos son los valores de las expresiones. Esta función es útil en los bloques y en las sentencias `for` para mostrar resultados intermedios. Los argumentos de `display` suelen ser átomos, variables subindicadas o llamadas a funciones.

Véanse también `ldisplay`, `disp` y `ldisp`.

Ejemplos:

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

                                          2
                              b     = x - x
                               1, 2

                        sin(1.0) = .8414709848078965

(%o3)                                done
```

`display2d` [Variable opcional]

Valor por defecto: `true`

Si `display2d` vale `false`, la salida por consola es una cadena unidimensional, en lugar de una expresión bidimensional.

Véase también `leftjust` para cambiar la justificación a la izquierda o el centrado de la ecuación.

Ejemplo:

```
(%i1) x/(x^2+1);
                                  x
(%o1)                           ------
                                 2
                                x  + 1
(%i2) display2d:false$
(%i3) x/(x^2+1);
(%o3) x/(x^2+1)
```

`display_format_internal` [Variable opcional]

Valor por defecto: `false`

Si `display_format_internal` vale `true`, las expresiones se muestran sin ser transformadas de manera que oculten su representación matemática interna. Se representa lo que la función `inpart` devolvería, en oposición a `part`.

Ejemplos:

```
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*) [Función]

Muestra *expr* en partes, una debajo de la otra. Esto es, primero se muestra el operador de *expr*, luego cada término si se trata de una suma, o cada factor si es un producto, o si no se muestra separadamente la parte de una expresión más general. Es útil si *expr* es demasiado grande para representarla de otra forma. Por ejemplo, si `P1`, `P2`, ... son expresiones muy grandes, entonces el programa de representación puede superar el espacio de almacenamiento tratando de mostrar `P1 + P2 + ...` todo junto. Sin embargo, `dispterms (P1 + P2 + ...)` muestra `P1`, debajo `P2`, etc. Cuando una expresión exponencial es demasiado ancha para ser representada como `A^B`, si no se utiliza `dispterms`, entonces aparecerá como `expt (A, B)` (o como `ncexpt (A, B)`, en lugar de `A^^B`).

Ejemplo:

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

+

2 a sin(x)

  x
%e

(%o1)                         done
```

`expt` (`a`, *b*) [Símbolo especial]
`ncexpt` (`a`, *b*) [Símbolo especial]

Si una expresión exponencial es demasiado ancha para ser mostrada como `a^`*b* aparecerá como `expt (a,` *b*`)` (o como `ncexpt (a,` *b*`)` en lugar de `a^^`*b*).

Las funciones `expt` y `ncexpt` no se reconocen en una entrada.

`exptdispflag` [Variable opcional]

Valor por defecto: `true`

Si `exptdispflag` vale `true`, Maxima muestra las expresiones con exponentes negativos como cocientes. Véase también `%edispflag`.

Ejemplo:

```
(%i1) exptdispflag:true;
(%o1)                         true
(%i2) 10^-x;
                                1
(%o2)                          ---
                                 x
                              10
```

```
(%i3) exptdispflag:false;
(%o3)                          false
(%i4) 10^-x;
                                 - x
(%o4)                          10
```

`grind` (*expr*) [Función]
`grind` [Variable opcional]

La función `grind` imprime *expr* en la consola en un formato admisible como entrada para Maxima. La función `grind` devuelve siempre `done`.

Cuando *expr* es el nombre de una función o macro, `grind` muestra la definición de la función o de la macro en lugar de sólo su nombre.

Véase también `string`, que devuelve una cadena en lugar de imprimir la salida. La función `grind` intenta imprimir la expresión de forma que sea lago más sencilla de leer que la salida de `string`.

Cuando la variable `grind` vale `true`, la salida de `string` y `stringout` tienen el mismo formato que la de `grind`; en caso contrario no se formatea la salida de esas funciones. El valor por defecto de la variable `grind` es `false`.

La variable `grind` también se puede utilizar como argumento en `playback`. Si `grind` está presente, `playback` imprime las expresiones de entrada en el mismo formato que lo hace la función `grind`; en caso contrario no se formatean la expresiones de entrada.

La función `grind` evalúa sus argumentos.

Ejemplos:

```
(%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` [Variable opcional]

Valor por defecto: `10`

`ibase` es la base en la que Maxima lee valores enteros.

A `ibase` se le puede asignar cualquier entero entre 2 y 36 (base decimal), ambos inclusive. Si `ibase` es mayor que 10, las cifras a utilizar serán los dígitos de 0 a 9, junto con las letras del alfabeto A, B, C, ..., tantas como sean necesarias para completar la base `ibase`. Las letras se interpretarán como cifras sólo cuando el primer dígito sea un valor entre 9. Es indiferente hacer uso de letras mayúsculas o minúsculas. Las cifras para la base 36, la mayor posible, son los dígitos numéricos de 0 a 9 y las letras desde la A hasta la Z.

Cualquiera que sea el valor de `ibase`, si un entero termina con un punto decimal, se interpretará en base 10.

Véase también `obase`.

Ejemplos:

`ibase` menor que 10.

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

`ibase` mayor que 10. Las letras se interpretan como dígitos sólo si el primer dígito es una cifra entre 0 y 9.

```
(%i1) ibase : 16 $
(%i2) obase;
(%o2)                          10
(%i3) 1000;
(%o3)                        4096
(%i4) abcd;
(%o4)                        abcd
(%i5) symbolp (abcd);
(%o5)                        true
(%i6) 0abcd;
(%o6)                        43981
(%i7) symbolp (0abcd);
(%o7)                        false
```

Independientemente del valor de `ibase`, si el entero termina con un punto decimal, se interpretará en base diez.

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

```
(%i4) 1234.;
(%o4)                          1234
```

ldisp (*expr_1*, ..., *expr_n*) [Función]

Muestra las expresiones *expr_1*, ..., *expr_n* en la consola con el formato de salida; `ldisp` asigna una etiqueta a cada argumento y devuelve la lista de etiquetas.

Véanse también `disp`, `display` y `ldisplay`.

```
(%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*) [Función]

Muestra las expresiones *expr_1*, ..., *expr_n* en la consola con el formato de salida. Cad expresión se muestra como una ecuación de la forma `lhs = rhs` en la que `lhs` es uno de los argumentos de `ldisplay` y `rhs` su valor. Normalmente, cada argumento será el nombre de una variable. La función `ldisp` asigna una etiqueta a cada ecuación y devuelve la lista de etiquetas.

Véanse también `disp`, `display` y `ldisp`.

```
(%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
```

`linel` [Variable opcional]

Valor por defecto: `79`

La variable `linel` es la anchura (medida en número de caracteres) de la consola que se le da a Maxima para que muestre las expresiones. A `linel` se le puede asignar cualquier valor, pero si éste es muy pequeño o grande resultará de poca utilidad. El texto que impriman las funciones internas de Maxima, como los mensajes de error y las salidas de la función `describe`, no se ve afectado por el valor de `linel`.

`lispdisp` [Variable opcional]

Valor por defecto: `false`

Si `lispdisp` vale `true`, los símbolos de Lisp se muestran precedidos del carácter de interrogación `?`. En caso contrario, los símbolos de Lisp se muestran sin esta marca.

Ejemplos:

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

`negsumdispflag` [Variable opcional]

Valor por defecto: `true`

Si `negsumdispflag` vale `true`, `x - y` se muestra como `x - y` en lugar de `- y + x`. Dándole el valor `false` se realiza un análisis adicional para que no se representen de forma muy diferente dos expresiones similares. Una aplicación puede ser para que `a + %i*b` y `a - %i*b` se representen ambas de la misma manera.

`obase` [Variable opcional]

Valor por defecto: `10`

`obase` es la base en la que Maxima imprime los números enteros.

A `obase` se le puede asignar cualquier entero entre 2 y 36 (base decimal), ambos inclusive. Si `obase` es mayor que 10, las cifras a utilizar serán los dígitos de 0 a 9, junto con las letras del alfabeto A, B, C, ..., tantas como sean necesarias para completar la base `obase`. Si el primer dígito resulta ser una letra, se le añadirá el cero como prefijo. Las cifras para la base 36, la mayor posible, son los dígitos numéricos de 0 a 9 y las letras desde la A hasta la Z.

Véase también `ibase`.

Ejemplos:

```
(%i1) obase : 2;
(%o1)                          10
(%i2) 2^8 - 1;
(%o10)                     11111111
(%i3) obase : 8;
(%o3)                          10
(%i4) 8^8 - 1;
(%o4)                      77777777
(%i5) obase : 16;
(%o5)                          10
(%i6) 16^8 - 1;
(%o6)                      0FFFFFFFF
(%i7) obase : 36;
(%o7)                          10
(%i8) 36^8 - 1;
(%o8)                      0ZZZZZZZZ
```

`pfeformat` [Variable opcional]

Valor por defecto: `false`

Si `pfeformat` vale `true`, una fracción de enteros será mostrada con el carácter de barra inclinada `/` entre ellos.

```
(%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)
```

`powerdisp` [Variable opcional]

Valor por defecto: `false`

Si `powerdisp` vale `true`, se muestran las sumas con sus términos ordenados de menor a mayor potencia. Así, un polinomio se presenta como una serie de potencias truncada con el término constante al principio y el de mayor potencia al final.

Por defecto, los términos de una suma se muestran en el orden de las potencias decrecientes.

Ejemplo:

```
(%i1) powerdisp:true;
```

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

`print` (*expr_1*, ..., *expr_n*) [Función]

Evalúa y muestra las expresiones *expr_1*, ..., *expr_n* secuencialmente de izquierda a derecha, comenzando la impresión por el borde izquierdo de la consola.

El valor devuelto por `print` es el valor de su último argumento. La función `print` no genera etiquetas para las expresiones intermedias.

Véanse también `display`, `disp`, `ldisplay` y `ldisp`, que muestran una expresión por línea, mientras que `print` trata de mostrar dos o más expresiones por línea.

Para mostrar el contenido de un archivo véase `printfile`.

```
(%i1) r: print ("(a+b)^3 is", expand ((a+b)^3), "log (a^10/b) is",
      radcan (log (a^10/b)))$
            3        2      2      3
(a+b)^3 is b  + 3 a b  + 3 a  b + a  log (a^10/b) is

                                                  10 log(a) - log(b)
(%i2) r;
(%o2)                   10 log(a) - log(b)
(%i3) disp ("(a+b)^3 is", expand ((a+b)^3), "log (a^10/b) is",
      radcan (log (a^10/b)))$
                            (a+b)^3 is

                         3        2      2      3
                        b  + 3 a b  + 3 a  b + a

                             log (a^10/b) is

                           10 log(a) - log(b)
```

`sqrtdispflag` [Variable opcional]

Valor por defecto: `true`

Si `sqrtdispflag` vale `false`, hará que `sqrt` se muestre con el exponente 1/2.

`stardisp` [Variable opcional]

Valor por defecto: `false`

Si `stardisp` vale `true`, la multiplicación se muestra con un asterisco `*` entre los operandos.

`ttyoff` [Variable opcional]

Valor por defecto: `false`

Si `ttyoff` vale `true`, no se muestran las expresiones resultantes, pero éstas se calculan de todos modos y se les asignan etiquetas. Véase `labels`.

El texto que escriban las funciones internas de Maxima, tales como los mensajes de error y las salidas de `describe`, no se ven afectadas por `ttyoff`.

5 Tipos de datos y estructuras

5.1 Números

5.1.1 Introducción a los números

Números enteros y racionales

Los cálculos aritméticos con números enteros y racionales son exactos. En principio, los números enteros y racionales admiten una cantidad arbitraria de cifras, con la única limitación que impongan las capacidades de memoria del sistema.

```
(%i1) 1/3+5/4+3;
                                55
(%o1)                           --
                                12
(%i2) 100!;
(%o2) 933262154439441526816992388562667004907159682643816214685\
929638952175999932299156089414639761565182862536979208272237582\
51852109168640000000000000000000000000
(%i3) 100!/101!;
                                 1
(%o3)                           ---
                                101
```

Funciones disponibles para los números enteros y racionales:

```
integerp        numberp        nonnegintegerp
oddp            evenp
ratnump         rationalize
```

Números decimales en coma flotante

Maxima hace los cálculos con números decimales en coma flotante en doble precisión. Además, Maxima puede hacer cálculos con números decimales grandes en coma flotante (*bigfloats*, en inglés), que, en principio, admiten precisión arbitraria.

La coma de los números decimales en coma flotante se escribe con un punto y el exponente se puede indicar con "f", "e" o "d". Por defecto, Maxima hace los cálculos en doble precisión y muestra el exponente como "e" en el resultado, mientras que representa el exponente con la letra "b" en el caso de decimales grandes de precisión arbitraria. Maxima no distingue si la letra del exponente se escribe en minúscula o mayúscula.

```
(%i1) [2.0,1f10,1,e10,1d10,1d300];
(%o1)        [2.0, 1.e+10, 1, e10, 1.e+10, 1.e+300]
(%i2) [2.0b0,1b10,1b300];
(%o2)               [2.0b0, 1.0b10, 1.0b300]
```

Si en un cálculo aritmético aparece un número decimal en coma flotante, se producirá un efecto de contagio que hará que el resultado se devuelva también como decimal. Esto también es cierto para el caso de decimales grandes de precisión arbitraria.

```
(%i1) 2.0+1/2+3;
```

```
(%o1)                                  5.5
(%i2) 2.0b0+1/2+3;
(%o2)                                5.5b0
```

Con las funciones `float` y `bfloat` se puede convertir un número en decimal de doble precisión, o de precisión arbitraria, respectivamente:

```
(%i1) float([2,1/2,1/3,2.0b0]);
(%o1)          [2.0, 0.5, .3333333333333333, 2.0]
(%i2) bfloat([2,1/2,1/3,2.0b0]);
(%o2)     [2.0b0, 5.0b-1, 3.333333333333333b-1, 2.0b0]
```

Funciones y variables disponibles para números decimales:

```
float        floatnump
bfloat       bfloatp       fpprec
float2bf     bftorat       ratepsilon

number_pbranch
m1pbranch
```

Números complejos

Maxima no tiene un tipo de dato especíco para números complejos; éstos se representan internamente como la suma de la parte real y la imaginaria multiplicada por el símbolo `%i`, que hace las veces de unidad imaginaria. Por ejemplo, las raíces de la ecuación `x^2 - 4*x + 13 = 0` se representan como `2 + 3*%i` y `2 - 3*%i`.

Maxima no simplifica automáticamente productos, cocientes, raíces y otras expresiones con números complejos. Por ejemplo, para hacer la multiplicación de números complejos se puede utilizar la función `expand`.

Funciones disponibles para los números complejos:

```
realpart     imagpart      rectform     polarform
cabs         carg          conjugate    csign
```

5.1.2 Funciones y variables para los números

`bfloat` (*expr*) [Función]

Convierte todos los números y funciones numéricas a números decimales de punto flotante grandes ("bigfloats"). El número de dígitos significativos de los "bigfloats" resultantes se especifica mediante la variable global `fpprec`.

Si `float2bf` vale `false` se mostrará un mensaje de aviso cuando un número en punto flotante se convierte a decimal de tipo "bigfloats", puesto que tal transformación puede conllevar pérdida de precisión.

`bfloatp` (*expr*) [Función]

Devuelve `true` si *expr* es un número decimal en punto flotante grande ("bigfloats"), en caso contrario devuelve `false`.

`bftorat` [Variable optativa]

Valor por defecto: `false`

La variable `bftorat` controla la conversión de números decimales de punto flotante grandes ("bigfloats") a números racionales. Si `bftorat` vale `false`, se utilizará `ratepsilon` para controlar la conversión (lo cual resulta en números racionales relativamente pequeños). Si `bftorat` vale `true`, el número racional generado representará exactamente al número decimal de punto flotante grande ("bigfloat").

`bftrunc` [Variable optativa]

Valor por defecto: `true`

La variable `bftrunc` provoca la eliminación de ceros en números decimales grandes no nulos para que no se muestren. Así, si `bftrunc` vale `false`, `bfloat (1)` se muestra como `1.000000000000000B0`. En otro caso, se mostrará como `1.0B0`.

`evenp` (*expr*) [Función]

Devuelve `true` si *expr* es un entero par y `false` en cualquier otro caso.

`float` (*expr*) [Función]

Convierte los enteros, números racionales y los decimales de punto flotante grandes ("bigfloats") que están presentes en *expr* a números de punto flotante. También actúa como símbolo `evflag`.

`float2bf` [Variable optativa]

Valor por defecto: `true`

Si `float2bf` vale `false` se mostrará un mensaje de aviso cuando un número en punto flotante se convierte a decimal de tipo "bigfloats", puesto que tal transformación puede conllevar pérdida de precisión.

`floatnump` (*expr*) [Función]

Devuelve `true` si *expr* es un número de punto flotante, en caso contario retorna `false`.

`fpprec` [Variable optativa]

Valor por defecto: 16

La variable `fpprec` guarda el número de dígitos significativos en la aritmética con números decimales de punto flotante grandes ("bigfloats"). La variable `fpprec` no afecta a los cálculos con números decimales de punto flotante ordinarios.

Véanse también `bfloat` y `fpprintprec`.

`fpprintprec` [Variable optativa]

Valor por defecto: 0

La variable `fpprintprec` guarda el número de dígitos a imprimir de los números decimales en coma flotante, tanto los ordinarios como los de precisión ilimitada (*bigfloats*).

En el caso de los decimales ordinarios, si `fpprintprec` toma un valor entre 2 y 16 (inclusive), el número de dígitos que se imprimen es igual a `fpprintprec`. En caso contrario, `fpprintprec` es 0 o mayor que 16, siendo el número de dígitos a imprimir en todos loa casos igual a 16.

En el caso de los decimales de precisión ilimitada (*bigfloats*), si `fpprintprec` toma un valor entre 2 y 16 (inclusive), el número de dígitos que se imprimen es igual a `fpprintprec`. En caso contrario, `fpprintprec` es 0 o mayor que `fpprec`, siendo el número de dígitos a imprimir igual a `fpprec`.

La variable `fpprintprec` no admite el valor 1.

integerp (*expr*) [Función]

Devuelve `true` si *expr* es un número entero y `false` en cualquier otro caso.

La función `integerp` devuelve `false` si su argumento es un símbolo, incluso cuando éste ha sido declarado como entero.

Ejemplos:

```
(%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 [Variable opcional]

Valor por defecto: `false`

La variable `m1pbranch` es la rama principal de `-1` elevado a una potencia. Cantidades como `(-1)^(1/3)` (esto es, un exponente racional impar) y `(-1)^(1/4)` (esto es, un exponente racional par) son tratados como sigue:

```
                  dominio real

(-1)^(1/3):      -1
(-1)^(1/4):   (-1)^(1/4)

              dominio complejo
m1pbranch:false          m1pbranch:true
(-1)^(1/3)               1/2+%i*sqrt(3)/2
(-1)^(1/4)              sqrt(2)/2+%i*sqrt(2)/2
```

nonnegintegerp (*n*) [Función]

Devuelve `true` si y solo si *n* `>= 0`, siendo *n* un entero.

numberp (*expr*) [Función]

Devuelve `true` si *expr* es un enúmero entero, racional, de coma flotante o "bigfloat", en caso contrario devuelve `false`.

La función `numberp` devuelve `false` si su argumento es un símbolo, incluso cuando el argumento es un número simbólico como `%pi` o `%i`, o aunque haya sido declarado

como `even` (par), `odd` (impar), `integer` (entero), `rational` (racional), `irrational` (irracional), `real` (real), `imaginary` (imaginario) o `complex` (complejo).

Ejemplos:

```
(%i1) numberp (42);
(%o1)                         true
(%i2) numberp (-13/19);
(%o2)                         true
(%i3) numberp (3.14159);
(%o3)                         true
(%i4) numberp (-1729b-4);
(%o4)                         true
(%i5) map (numberp, [%e, %pi, %i, %phi, inf, minf]);
(%o5)      [false, false, false, false, false, false]
(%i6) declare (a, even, b, odd, c, integer, d, rational,
     e, irrational, f, real, g, imaginary, h, complex);
(%o6)                         done
(%i7) map (numberp, [a, b, c, d, e, f, g, h]);
(%o7) [false, false, false, false, false, false, false, false]
```

`numer` [Variable opcional]

La variable `numer` hace algunas funciones matemáticas con argumentos numéricos se evalúen como decimales de punto flotante. También hace que las variables de una expresión a las cuales se les ha asignado un número sean sustituidas por sus valores. Además, activa la variable `float`.

Véase también `%enumer`.

Ejemplos:

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

`numer_pbranch` [Variable opcional]

Valor por defecto: `false`

La variable opcional `numer_pbranch` controla la evaluación numérica de las potencias de números enteros, racionales y decimales negativos. Si `numer_pbranch` vale `true` y el exponente es decimal o la variable opcional `numer` vale `true`, Maxima evalúa el resultado numérico utilizando la rama principal. En caso contrario, se devuleve un resultado simplificado pero no evaluado.

Ejemplos:

```
(%i1) (-2)^0.75;
(%o1) (-2)^0.75

(%i2) (-2)^0.75,numer_pbranch:true;
(%o2) 1.189207115002721*%i-1.189207115002721
```

```
(%i3) (-2)^(3/4);
(%o3) (-1)^(3/4)*2^(3/4)

(%i4) (-2)^(3/4),numer;
(%o4) 1.681792830507429*(-1)^0.75

(%i5) (-2)^(3/4),numer,numer_pbranch:true;
(%o5) 1.189207115002721*%i-1.189207115002721
```

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

Declara las variables `x_1`, ..., *x_n* asignándoles los valores numéricos `expr_1`, ..., `expr_n`. El valor numérico se evalúa y sustituye a la variable en cualquier expresión en la que ésta aparezca si `numer` toma el valor `true`. Véase también `ev`.

Las expresiones `expr_1`, ..., `expr_n` pueden ser expresiones no necesariamente numéricas.

oddp (*expr*) [Función]

Devuelve `true` si *expr* es un entero impar y `false` en caso contrario.

ratepsilon [Variable opcional]

Valor por defecto: 2.0e-8

La variable `ratepsilon` guarda la tolerancia utilizada en la conversión de números decimales en coma flotante a números racionales.

rationalize (*expr*) [Función]

Convierte todos los números en coma flotante de doble precisión y grandes (big float) presentes en una expresión *expr* de Maxima a sus formas racionales exactas equivalentes. Si el usuario no está familiarizado con la representación binaria de números en coma flotante, le puede extrañar que `rationalize (0.1)` no sea igual que 1/10. Este comportamiento no es único de Maxima, ya que el número 1/10 en su forma binaria es periódico y no exacto.

```
(%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);
```

```
                        23611832414348226068 5
(%o6)                   ----------------------
                        23611832414348226068 48
(%i7) rationalize (sin (0.1*x + 5.6));
                                  x    28
(%o7)                        sin(-- + --)
                                  10   5
```

`ratnump` (*expr*) [Función]

Devuelve `true` si *expr* es un entero literal o una fracción de enteros literales, en caso contrario devuelve `false`.

5.2 Cadenas de texto

5.2.1 Introducción a las cadenas de texto

Las cadenas de caracteres deben ir acotadas por comillas dobles (") al ser introducidas en Maxima, siendo luego mostradas con o sin ellas, dependiendo del valor de la variable global `stringdisp`.

Las cadenas pueden contener todo tipo de caracteres, incluyendo tabulaciones, caracteres de nueva línea y de retorno. La secuencia \" se reconoce literalmente como una comilla doble, al tiempo que \\ se interpreta como una barra invertida. Cuando la barra invertida aparece al final de una línea, tanto la barra como el final de línea (representado éste bien por el carácter de nueva línea o el de retorno) son ignorados, de forma que la cadena continúa en el siguiente renglón. No se reconocen más combinaciones especiales de la barra invertida con otros caracteres aparte de las comentadas; de modo que si la barra invertida aparece antes de cualquier otro carácter distinto de ", \, o de un final de línea, dicha barra será ignorada. No hay manera de representar los caracteres especiales (tabulación, nueva línea o retorno) de otra forma que no sea incluyéndolos literalmente en la cadena.

No existe en Maxima el tipo de variable carácter, debiéndose representar un carácter simple como una cadena de un solo carácter.

El paquete adicional `stringproc` contiene muchas funciones que permiten trabajar con cadenas.

Ejemplos:

```
(%i1) s_1 : "This is a string.";
(%o1)               This is a string.
(%i2) s_2 : "Embedded \"double quotes\" and backslash \\ characters.";
(%o2) Embedded "double quotes" and backslash \ characters.
(%i3) s_3 : "Embedded line termination
in this string.";
(%o3) Embedded line termination
in this string.
(%i4) s_4 : "Ignore the \
line termination \
characters in \
this string.";
(%o4) Ignore the line termination characters in this string.
(%i5) stringdisp : false;
(%o5)                          false
(%i6) s_1;
(%o6)                This is a string.
(%i7) stringdisp : true;
(%o7)                          true
(%i8) s_1;
(%o8)              "This is a string."
```

5.2.2 Funciones y variables para las cadenas de texto

`concat` (*arg_1*, *arg_2*, ...) [Función]

Concatena sus argumentos, que deben ser todos átomos. El valor devuelto es un símbolo si el primer argumento es a su vez un símbolo, o una cadena en caso contrario.

La función `concat` evalúa sus argumentos. El apátrofo `'` evita la evaluación.

```
(%i1) y: 7$
(%i2) z: 88$
(%i3) concat (y, z/2);
(%o3)                          744
(%i4) concat ('y, z/2);
(%o4)                          y44
```

A un símbolo construido por `concat` se le puede asignar un valor y ser utilizado posteriormente en expresiones. La asignación con el operador `::` evalúa su expresión izquierda.

```
(%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
```

Nótese que aunque `concat (1, 2)` parezca un número, se trata de una cadena.

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

`sconcat` (*arg_1*, *arg_2*, ...) [Función]

Concatena sus argumentos para producir una cadena. Al contrario que `concat`, sus argumentos *no* necesitan ser átomos.

El resultado es una cadena.

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

`string` (*expr*) [Función]

Convierte `expr` a la notación lineal de Maxima, tal como si fuese tecleada.

El valor que retorna la función `string` es una cadena, por lo que no puede ser utilizada en los cálculos.

`stringdisp` [Variable opcional]

Valor por defecto: `false`

Si `stringdisp` vale `true`, las cadenas alfanuméricas se muestran encerradas entre comillas dobles. En caso contrario, no se nuestran las comillas.

La variable `stringdisp` vale siempre `true` cuando se muestra la definición de una función.

Ejemplos:

```
(%i1) stringdisp: false$
(%i2) "This is an example string.";
(%o2)              This is an example string.
(%i3) foo () :=
      print ("This is a string in a function definition.");
(%o3) foo() :=
               print("This is a string in a function definition.")
(%i4) stringdisp: true$
(%i5) "This is an example string.";
(%o5)             "This is an example string."
```

5.3 Constantes

5.3.1 Funciones y variables para Constantes

`%e` [Constante]

El símbolo `%e` representa a la base de los logaritmos naturales, también conocido como número de Euler. El valor numérico de `%e` es el número decimal en coma flotante 2.718281828459045d0.

`%i` [Constante]

El símbolo `%i` representa la unidad imaginaria, $sqrt(-1)$.

`false` [Constante]

El símbolo `false` representa al valor lógico "falso". Maxima implementa `false` con el valor `NIL` de Lisp.

`%gamma` [Constante]

Es la constante de Euler-Mascheroni, 0.5772156649015329 ....

`ind` [Constante]

`ind` representa un resultado acotado indeterminado.

Véase también `limit`.

Ejemplo:

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

`inf` [Constante]

El símbolo `inf` representa al infinito real positivo.

`infinity` [Constante]

El símbolo `infinity` representa al infinito complejo.

`minf` [Constante]

El símbolo `minf` representa al infinito real negativo.

`%phi` [Constante]

El símbolo `%phi` representa a la llamada *razón áurea*, $(1+sqrt(5))/2$. El valor numérico de `%phi` es el número decimal en coma flotante 1.618033988749895d0.

La función `fibtophi` expresa los números de Fibonacci `fib(n)` en términos de `%phi`.

Por defecto, Maxima desconoce las propiedades algebraicas de `%phi`. Tras evaluar `tellrat(%phi^2 - %phi - 1)` y `algebraic: true`, `ratsimp` puede simplificar algunas expresiones que contengan `%phi`.

Ejemplos:

`fibtophi` expresa el número de Fibonacci `fib(n)` en términos de `%phi`.

```
(%i1) fibtophi (fib (n));
                              n             n
                          %phi  - (1 - %phi)
(%o1)                     ------------------
```

```
                              2 %phi - 1
(%i2) fib (n-1) + fib (n) - fib (n+1);
(%o2)          - fib(n + 1) + fib(n) + fib(n - 1)
(%i3) fibtophi (%);
            n + 1             n + 1       n             n
          %phi      - (1 - %phi)        %phi  - (1 - %phi)
(%o3) - -------------------------- + -------------------
                2 %phi - 1                2 %phi - 1
                                          n - 1             n - 1
                                        %phi      - (1 - %phi)
                                      + --------------------------
                                                2 %phi - 1
(%i4) ratsimp (%);
(%o4)                             0
```

Por defecto, Maxima desconoce las propiedades algebraicas de `%phi`. Después de evaluar `tellrat (%phi^2 - %phi - 1)` y `algebraic: true`, `ratsimp` puede simplificar algunas expresiones que contengan `%phi`.

```
(%i1) e : expand ((%phi^2 - %phi - 1) * (A + 1));
                  2                       2
(%o1)         %phi  A - %phi A - A + %phi  - %phi - 1
(%i2) ratsimp (e);
                   2                      2
(%o2)         (%phi  - %phi - 1) A + %phi  - %phi - 1
(%i3) tellrat (%phi^2 - %phi - 1);
                              2
(%o3)                    [%phi  - %phi - 1]
(%i4) algebraic : true;
(%o4)                           true
(%i5) ratsimp (e);
(%o5)                             0
```

`%pi` [Constante]

El símbolo `%pi` representa la razón entre la longitud de una circunferencia y su radio. El valor numérico de `%pi` es el número decimal en coma flotante 3.141592653589793d0.

`true` [Constante]

El símbolo `true` representa al valor lógico "verdadero". Maxima implementa `true` con el valor `T` de Lisp.

`und` [Constante]

`und` representa un resultado indefinido.

Véase también `limit`.

Ejemplo:

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

`zeroa` [Constante]

`zeroa` representa un infinitesimal mayor que cero. `zeroa` puede utilizarse en expresiones. `limit` simplifica expresiones que contienen infinitesimales.

Véanse también `zerob` y `limit`.

Ejemplo:

`limit` simplifica expresiones que contienen infinitesimales:

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

`zerob` [Constante]

`zerob` representa un infinitesimal menor que cero. `zerob` puede utilizarse en expresiones. `limit` simplifica expresiones que contienen infinitesimales.

Véanse también `zeroa` y `limit`.

5.4 Listas

5.4.1 Introducción a las listas

Las listas son bloques de construcción básica para Maxima y Lisp. Todos los tipos de datos diferentes a los arreglos, tablas mixtas o números son representados como listas Lisp, estas listas Lisp tienen la forma

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

para indicar la expresión `a+2`. Al nivel de Maxima se observará la notación infija `a+2`. Maxima también tiene listas con el formato

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

para una lista de 4 elementos. Internamente esto se corresponde con una lista Lisp de la forma

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

El elemento que denota el tipo de expresión en Maxima es también una lista, la cual tras ser analizada y simplificada tomará la forma

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

5.4.2 Funciones y variables para listas

`[` [Operador]
`]` [Operador]

`[` y `]` marcan, respectivamente, el comienzo y el final de una lista.

`[` y `]` también se utilizan para indicar los subíndices de una lista o de un array.

Ejemplos:

```
(%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
```

append (*lista_1*, ..., *lista_n*) [Función]

Devuelve una lista cuyos elementos son los de la lista *lista_1* seguidos de los de *lista_2*, ... La función `append` también opera con expresiones generales, como la llamada `append (f(a,b), f(c,d,e));`, de la que se obtiene `f(a,b,c,d,e)`.

Tecléese `example(append);` para ver un ejemplo.

assoc (*clave*, *lista*, *valor_por_defecto*) [Función]
assoc (*clave*, *lista*) [Function]

Esta función busca la *clave* en el lado derecho de la *lista*, la cual es de la forma `[x,y,z,...]`, donde cada elemento es una expresión formada por un operador binario y dos elementos. Por ejemplo, `x=1`, `2^3`, `[a,b]` etc. La *clave* se compara con el primer operando. La función `assoc` devuelve el segundo operando si se encuentra con que la `clave` coincide. Si la `clave` no coincide entonces devuelve el valor *valor_por_defecto*. El argumento *valor_por_defecto* es opcional; en caso de no estar presente, se devolverá `false`.

cons (*expr*, *lista*) [Función]

Devuelve una nueva lista en la que el elemento *expr* ocupa la primera posición, seguido de los elementos de *lista*. La función `cons` también opera con otro tipo de expresiones, como `cons(x, f(a,b,c));` -> `f(x,a,b,c)`.

copylist (*lista*) [Función]

Devuelve una copia de la *lista*.

create_list (*form*, *x_1*, *list_1*, ..., *x_n*, *list_n*) [Función]

Crea una lista mediante la evaluación de *form* con *x_1* tomando cada uno de los valores de *list_1*, para cada uno de estos valores liga *x_2* con cada elemento de *list_2*, ... El número de elementos en el resultado será el producto del número de elementos en cada lista. Cada variable *x_i* debe ser un símbolo y no será evaluado. La lista de argumentos será evaluada una vez al comienzo de la iteración.

Ejemplos:

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

Con una doble iteración:

```
(%i1) create_list ([i, j], i, [a, b], j, [e, f, h]);
(%o1)   [[a, e], [a, f], [a, h], [b, e], [b, f], [b, h]]
```

En lugar de *list_i* se pueden suministrar dos argumentos cada uno de los cuales debería poder evaluarse a un número, los cuales serán los límites inferior y superior, ambos inclusive, para cada iteración.

```
(%i1) create_list ([i, j], i, [1, 2, 3], j, 1, i);
(%o1)   [[1, 1], [2, 1], [2, 2], [3, 1], [3, 2], [3, 3]]
```

Nótese que los límites o lista para la variable `j` pueden depender del valor actual de `i`.

`delete` (*expr_1*, *expr_2*) [Función]
`delete` (*expr_1*, *expr_2*, *n*) [Función]

`delete(`*expr_1*`,` *expr_2*`)` elimina de *expr_2* cualesquiera argumentos del operador del nivel superior que sean iguales a *expr_1*. Nótese que los argumentos de las subexpresiones no se ven afectados por esta función.

expr_1 puede ser un átomo o una expresión no atómica. *expr_2* puede ser cualquier expresión no atómica. La función `delete` devuelve una nueva expresión sin modificar *expr_2*.

`delete(`*expr_1*`,` *expr_2*`,` *n*`)` elimina de *expr_2* los primeros *n* argumentos del operador del nivel superior que sean iguales a *expr_1*. Si hay menos de *n* argumentos iguales, entonces se eliminan todos ellos.

Ejemplos:

Eliminando elementos de una lista.

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

Eliminando términos de una suma.

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

Eliminando factores de un producto.

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

Eliminando argumentos de una expresión arbitraria.

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

Limitando el número de argumentos a eliminar.

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

Los argumentos se comparan respecto de "=". Aquellos argumentos que verifiquen la condición `equal`, pero no "=" no serán eliminados.

```
(%i1) [is (equal (0, 0)), is (equal (0, 0.0)), is (equal (0, 0b0))];
rat: replaced 0.0 by 0/1 = 0.0
'rat' replaced 0.0B0 by 0/1 = 0.0B0
(%o1)                   [true, true, true]
(%i2) [is (0 = 0), is (0 = 0.0), is (0 = 0b0)];
(%o2)                  [true, false, false]
(%i3) delete (0, [0, 0.0, 0b0]);
(%o3)                       [0.0, 0.0b0]
(%i4) is (equal ((x + y)*(x - y), x^2 - y^2));
(%o4)                          true
(%i5) is ((x + y)*(x - y) = x^2 - y^2);
(%o5)                          false
```

```
(%i6) delete ((x + y)*(x - y), [(x + y)*(x - y), x^2 - y^2]);
                                  2    2
(%o6)                            [x  - y ]
```

`eighth` (*expr*) [Función]

Devuelve el octavo elemento de la lista o expresión *expr*. Véase `first` para más detalles.

`endcons` (*expr*, *lista*) [Función]

Devuelve una nueva lista formada por los elementos de `lista` seguidos de los de *expr*. La función `endcons` también opera con expresiones generales, por ejemplo `endcons(x, f(a,b,c));` -> `f(a,b,c,x)`.

`fifth` (*expr*) [Función]

Devuelve el quinto elemento de la lista o expresión *expr*. Véase `first` para más detalles.

`first` (*expr*) [Función]

Devuelve la primera parte de *expr*, que puede consistir en el primer elemento de una lista, la primera fila de una matriz, el primer término de una suma, etc. Nótese que tanto `first` como sus funciones relacionadas, `rest` y `last`, operan sobre la forma en la que *expr* es mostrada por Maxima, no sobre la forma en la que es introducida la expresión. Sin embargo, cuando la variable `inflag` toma el valor `true` estas funciones tendrán en cuenta el formato interno de *expr*. Téngase en cuenta que el simplificador reordena las expresiones. Así, `first(x+y)` devolverá `x` si `inflag` vale `true` y `y` cuando `inflag` tome el valor `false` (`first(y+x)` devuelve el mismo resultado). Las funciones `second` ... `tenth` devuelven desde el segundo hasta el décimo elemento del argumento de entrada.

`fourth` (*expr*) [Función]

Devuelve el cuarto elemento de la lista o expresión *expr*. Véase `first` para más detalles.

`join` (*l*, *m*) [Función]

Crea una nueva lista con los elementos de las listas *l* y *m* alternados. El resultado tiene como elementos `[`*l*`[1], `*m*`[1], `*l*`[2], `*m*`[2], ...]`. Las listas *l* y *m* pueden contener cualquier tipo de elementos.

Si las listas son de diferente longitud, `join` ignora los elementos sobrantes de la lista más larga.

Maxima da error si o bien *l* o *m* no son listas.

Ejemplos:

```
(%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*) [Función]

Devuelve la última parte (término, fila, elemento, etc.) de *expr*.

`length` (*expr*) [Función]

Devuelve (por defecto) el número de partes de que consta *expr* en la versión correspondiente a la que muestra. En el caso de listas, se devuelve el número de elementos, si se trata de matrices el número de filas y se se trata de sumas el número de términos o sumandos (véase `dispform`).

La función `length` se ve afectada por el valor de la variable `inflag`. Así, `length(a/(b*c));` devuelve 2 si `inflag` vale `false` (dando por hecho que `exptdispflag` vale `true`), pero devuelve 3 si `inflag` vale `true` (ya que la representación interna es `a*b^-1*c^-1`).

`listarith` [Variable opcional]

Valor por defecto: `true`

Cuando vale `false` provoca que no se realicen operaciones aritméticas con listas; cuando vale `true`, las operaciones con listas y matrices son contagiosas, en el sentido de que las listas se transforman en matrices, retornando resultados de este último tipo. Sin embargo, operaciones que involucren listas con listas devolverán también listas.

`listp` (*expr*) [Función]

Devuelve el valor `true` si *expr* es una lista, y `false` en caso contrario.

`makelist` () [Función]
`makelist` (*expr*, *n*) [Función]
`makelist` (*expr*, *i*, *i_max*) [Función]
`makelist` (*expr*, *i*, *i_0*, *i_max*) [Función]
`makelist` (*expr*, *i*, *i_0*, *i_max*, *step*) [Función]
`makelist` (*expr*, *x*, *list*) [Función]

El primer formato, `makelist ()`, crea una lista vacía. El segundo formato, `makelist (expr)`, crea una lista con *expr* como único elemento. `makelist (expr, n)` crea una lista de *n* elementos generados a partir de *expr*.

El formato más general, `makelist (expr, i, i_0, i_max, step)`, devuelve la lista de elementos obtenidos al aplicar `ev (expr, i=j)` a los elementos *j* de la secuencia *i_0*, *i_0* + *step*, *i_0* + 2**step*, ..., siendo |*j*| menor o igual que |*i_max*|.

El incremento *step* puede ser un número (positivo o negativo) o una expresión. En caso de omitirse, se utilizará 1 como valor por defecto. Si se omiten *i_0* y *step*, se le asignará a ambos 1 como valor por defecto.

`makelist (expr, x, list)` devuelve una lista, cuyo `j`-ésimo elemento es igual a `ev (expr, x=list[j])` tomando `j` valores desde 1 hasta `length (list)`.

Ejemplos:

```
(%i1) makelist (concat (x,i), i, 6);
(%o1)               [x1, x2, x3, x4, x5, x6]
(%i2) makelist (x=y, y, [a, b, c]);
(%o2)                 [x = a, x = b, x = c]
```

```
(%i3) makelist (x^2, x, 3, 2*%pi, 2);
(%o3)                         [9, 25]
(%i4) makelist (random(6), 4);
(%o4)                      [2, 0, 2, 5]
(%i5) flatten (makelist (makelist (i^2, 3), i, 4));
(%o5)        [1, 1, 1, 4, 4, 4, 9, 9, 9, 16, 16, 16]
(%i6) flatten (makelist (makelist (i^2, i, 3), 4));
(%o6)         [1, 4, 9, 1, 4, 9, 1, 4, 9, 1, 4, 9]
```

`member` (*expr_1*, *expr_2*) [Función]

Devuelve `true` si `is(`*expr_1* `= a)` para algún elemento *a* de `args(`*expr_2*`)`, en caso contrario devuelve `false`.

Normalmente, `expr_2` será una lista, en cuyo caso `args(`*expr_2*`) =` *expr_2*, y la comprobación será si `is(`*expr_1* `= a)` para algún elemento *a* de `expr_2`.

La función `member` no inspecciona las partes de los argumentos de `expr_2`, por lo que puede devolver `false` si `expr_1` es parte de alguno de los argumentos de `expr_2`.

Véase también `elementp`.

Ejemplos:

```
(%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*) [Función]

Devuelve el noveno elemento de la lista o expresión *expr*. Véase `first` para más detalles.

`pop` (*list*) [Función]

Borra el primer elemento de la lista *list* y devuelve este mismo elemento.

Si el argumento *list* es una lista vacía, o simplemente no es una lista, Maxima devuelve un mensaje de error.

Véase `push` para los ejemplos.

Ejecútese `load("basic")` antes de utilizar esta función.

`push` (*item*, *list*) [Función]

Añade al comienzo de la lista *list* el elemento *item*, devolviendo este mismo elemento. El argumento *list* debe ser necesariamente una lista, mientras que *item* puede ser cualquier símbolo o expresión.

Si el argumento *list* no es una lista, Maxima devuelve un mensaje de error.

Véase `pop` para eliminar el primer elemento de una lista.

Ejecútese `load("basic")` antes de utilizar esta función.

Ejemplos:

```
(%i1) load ("basic")$
(%i2) ll: [];
(%o2)                           []
(%i3) push (x, ll);
(%o3)                           [x]
(%i4) push (x^2+y, ll);
                                 2
(%o4)                      [y + x , x]
(%i5) a: push ("string", ll);
                                       2
(%o5)                [string, y + x , x]
(%i6) pop (ll);
(%o6)                        string
(%i7) pop (ll);
                                   2
(%o7)                        y + x
(%i8) pop (ll);
(%o8)                           x
(%i9) ll;
(%o9)                           []
(%i10) a;
                                       2
(%o10)               [string, y + x , x]
```

`rest` (*expr*, *n*) [Función]

`rest` (*expr*) [Función]

Devuelve *expr* sin sus primeros *n* elementos si *n* es positivo, o sus últimos `-` *n* elementos si *n* es negativo. En caso de que *n* tome el valor 1 puede ser omitido. La expresión *expr* puede ser una lista, una matriz o cualquier otra expresión.

`reverse` (*lista*) [Función]

Invierte el orden de los elementos de la *lista* (no los propios elementos). La función `reverse` también opera sobre expresiones generales, como en `reverse(a=b);` gives `b=a`.

`second` (*expr*) [Función]

Devuelve el segundo elemento de la lista o expresión *expr*. Véase `first` para más detalles.

`seventh` (*expr*) [Función]
: Devuelve el séptimo elemento de la lista o expresión *expr*. Véase `first` para más detalles.

`sixth` (*expr*) [Función]
: Devuelve el sexto elemento de la lista o expresión *expr*. Véase `first` para más detalles.

`sort` (*L*, *P*) [Función]
`sort` (*L*) [Función]
: `sort(L, P)` ordena la lista *L* de acuerdo con el predicado `P` de dos argumentos, el cual define un preorden total sobre los elementos de *L*. Si `P(a, b)` vale `true`, entonces `a` aparece antes que `b` en el resultado. Si ninguna de las expresiones `P(a, b)` y `P(b, a)` valen `true`, entonces `a` y `b` son equivalentes y aparecen en el resultado en el mismo orden que a la entrada; esto es, `sort` es un orden estable.

  Si tanto `P(a, b)` como `P(b, a)` valen ambos `true` para algunos elementos de *L*, entonces *P* no es un predicado de orden correcto, siento entonces el resultado indefinido. Si `P(a, b)` toma un valor diferente a `true` o `false`, entonces `sort` devuelve un error.

  El predicado puede especificarse como el nombre de una función, de una operación binaria infija o como una expresión `lambda`. Si se especifica con el nombre de un operador, dicho nombre debe encerrarse con comillas dobles.

  La lista ordenada se devuelve como un nuevo objeto, no modificándose el argumento *L*.

  `sort(L)` equivale a `sort(L, orderlessp)`.

  La ordenación por defecto es ascendente, tal como queda determinada por `orderlessp`. El predicado `ordergreatp` ordena las listas en orden descendente.

  Todos los átomos y expresiones de Maxima son comparables respecto de los predicados `orderlessp` y `ordergreatp`.

  Los operadores `<` y `>` ordenan números, constantes y expresiones constantes por magnitud. Nótese que `orderlessp` y `ordergreatp` no ordenan números, constantes y expresiones constantes por magnitud.

  `ordermagnitudep` ordena números, constantes y expresiones constantes de igual modo que lo hace `<`, y cualesquiera otros elementos lo hace igual que `orderlessp`.

  Ejemplos:

  `sort` ordena una lista respecto de un predicado de dos argumentos que define un preorden total en los elementos de la lista.

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

  El predicado puede especificarse con el nombre de una función, de un operador binario infijo o una expresión `lambda`. Si se especifica con el nombre de un operador, dicho nombre debe encerrarse con comillas dobles.

```
(%i1) L : [[1, x], [3, y], [4, w], [2, z]];
(%o1)           [[1, x], [3, y], [4, w], [2, z]]
(%i2) foo (a, b) := a[1] > b[1];
(%o2)                 foo(a, b) := a  > b
                                    1    1
(%i3) sort (L, 'foo);
(%o3)           [[4, w], [3, y], [2, z], [1, x]]
(%i4) infix (">>");
(%o4)                           >>
(%i5) a >> b := a[1] > b[1];
(%o5)                   a >> b := a  > b
                                  1    1
(%i6) sort (L, ">>");
(%o6)           [[4, w], [3, y], [2, z], [1, x]]
(%i7) sort (L, lambda ([a, b], a[1] > b[1]));
(%o7)           [[4, w], [3, y], [2, z], [1, x]]
```

`sort(L)` equivale a `sort(L, orderlessp)`.

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

La ordenación por defecto es ascendente, tal como queda determinada por `orderlessp`. El predicado `ordergreatp` ordena las listas en orden descendente.

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

Todos los átomos y expresiones de Maxima son comparables respecto de los predicados `orderlessp` y `ordergreatp`.

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

Los operadores < y > ordenan números, constantes y expresiones constantes por magnitud. Nótese que `orderlessp` y `ordergreatp` no ordenan números, constantes y expresiones constantes por magnitud.

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

`ordermagnitudep` ordena números, constantes y expresiones constantes de igual modo que lo hace <, y cualesquiera otros elementos lo hace igual que `orderlessp`.

```
(%i1) L : [%i, 1+%i, 2*x, minf, inf, %e, sin(1), 0, 1, 2, 3, 1.0, 1.0b0];
(%o1) [%i, %i + 1, 2 x, minf, inf, %e, sin(1), 0, 1, 2, 3, 1.0,
                                                              1.0b0]
(%i2) sort (L, ordermagnitudep);
(%o2) [minf, 0, sin(1), 1, 1.0, 1.0b0, 2, %e, 3, inf, %i,
                                                    %i + 1, 2 x]
(%i3) sort (L, orderlessp);
(%o3) [0, 1, 1.0, 2, 3, %e, %i, %i + 1, inf, minf, sin(1),
                                                     1.0b0, 2 x]
```

`sublist` (*list*, *p*) [Función]

Devuelve la lista de elementos de *list* para los cuales el predicado `p` retorna `true`.

Ejemplo:

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

`sublist_indices` (*L*, *P*) [Función]

Devuelve los índices de los elementos `x` de la lista *L* para la cual el predicado `maybe(`*P*`(x))` devuelve `true`, lo que excluye a `unknown` y a `false`. *P* puede ser el nombre de una función o de una expresión lambda. *L* debe ser una lista literal.

Ejemplos:

```
(%i1) sublist_indices ('[a, b, b, c, 1, 2, b, 3, b],
                       lambda ([x], x='b));
(%o1)                    [2, 3, 7, 9]
(%i2) sublist_indices ('[a, b, b, c, 1, 2, b, 3, b], symbolp);
(%o2)                  [1, 2, 3, 4, 7, 9]
(%i3) sublist_indices ([1 > 0, 1 < 0, 2 < 1, 2 > 1, 2 > 0],
                       identity);
(%o3)                       [1, 4, 5]
(%i4) assume (x < -1);
(%o4)                       [x < - 1]
(%i5) map (maybe, [x > 0, x < 0, x < -2]);
(%o5)                [false, true, unknown]
```

```
(%i6) sublist_indices ([x > 0, x < 0, x < -2], identity);
(%o6)                          [2]
```

`unique` (*L*) [Función]

Devuelve la lista *L* sin redundancias, es decir, sin elementos repetidos

Cuando ninguno de los elementos de *L* está repetido, `unique` devuelve una réplica de *L*, no la propia *L*.

Si *L* no es una lista, `unique` devuelve *L*.

Ejemplo:

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

`tenth` (*expr*) [Función]

Devuelve el décimo elemento de la lista o expresión *expr*. Véase `first` para más detalles.

`third` (*expr*) [Función]

Devuelve el tercer elemento de la lista o expresión *expr*. Véase `first` para más detalles.

5.5 Arrays

5.5.1 Introducción a los arrays

Los arrays más flexibles son aquellos que no necesitan ser declarados, llamados también, en inglés, *hashed-arrays*, y se basan en que a una variable subindicada se le puede asignar un valor cualquiera. Los índices no necesitan ser números enteros, admitiéndose símbolos o expresiones. Los arrays no declarados crecen dinámicamente según se le van asignando valores a sus elementos. En el siguiente ejemplo se muestra cómo se va construyendo un array no declarado `a`. Para obtener un listado de los elementos de un array se utiliza la función `listarray`.

```
(%i1) a[1,2]: 99;
(%o1)                           99
(%i2) a[x,y]: x^y;
                                 y
(%o2)                           x
(%i3) listarray(a);
                                   y
(%o3)                         [99, x ]
```

Otro tipo de arrays son los declarados, los cuales admiten hasta cinco dimensiones y pueden guardar valores de un tipo concreto, como `fixnum` para enteros o `flonum` para reales de coma flotante. Maxima distingue dos tipos de arrays declarados; los primeros se pueden definir declarando un símbolo como array, haciendo uso de la función `array`; los segundos son arrays de Lisp, en los que un símbolo se declara como tal con la función `make_array`.

En el primer ejemplo se declara el símbolo `a` como array, mientras que en el segundo se declara `b` como array de Lisp.

```
(%i1) array(a, fixnum, 2, 2);
(%o1)                            a
(%i2) b: make_array(fixnum, 2, 2);
(%o2)              {Array:  #2A((0 0) (0 0))}
```

Cuando a la variable opcional `use_fast_arrays` se le asigna el valor `true`, la función `array` también generará un array de Lisp. Tal es lo que sucede en el ejemplo siguiente, en el que el símbolo `c` queda declarado como array de Lisp. Téngase en cuenta que por este método no se puede asignar el tipo de array, ya que al introducirle el tipo `fixnum` se genera un mensaje de error.

```
(%i3) use_fast_arrays: true;
(%o3)                          true
(%i4) array(c, 2, 2);
(%o4)    #2A((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))
(%i5) c;
(%o5)    #2A((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))
(%i6) array(c, fixnum, 2, 2);

make_array: dimensions must be integers; found [fixnum + 1, 3, 3]
 -- an error. To debug this try: debugmode(true);
```

Maxima también dispone de funciones array, que pueden almacenar valores de funciones, y de funciones subindicadas.

Se puede encontrar más información en las descripciones de las funciones. Los siguientes símbolos y funciones de Maxima permiten trabajar con arrays:

```
array          arrayapply     arrayinfo
arraymake      arrays         fillarray
listarray      make_array     rearray
remarray       subvar         subvarp
use_fast_arrays
```

5.5.2 Funciones y variables para los arrays

`array` (*nombre*, *dim_1*, ..., *dim_n*) [Función]
`array` (*nombre*, *type*, *dim_1*, ..., *dim_n*) [Función]
`array` ([*nombre_1*, ..., *nombre_m*], *dim_1*, ..., *dim_n*) [Función]

Crea un array de dimensión n, que debe ser menor o igual que 5. Los subíndices de la i-ésima dimensión son enteros que toman valores entre 0 y *dim_i*.

La llamada `array (nombre, dim_1, ..., dim_n)` crea un array de tipo general.

La llamada `array (nombre, type, dim_1, ..., dim_n)` crea un array con sus elementos del tipo especificado. El tipo *type* puede ser `fixnum` para enteros de tamaño limitado o `flonum` para números decimales en coma flotante.

La llamada `array ([nombre_1, ..., nombre_m], dim_1, ..., dim_n)` crea m arrays, todos ellos de igual dimensión.

Si el usuario asigna un valor a una variable subindicada antes de declarar el array correspondiente, entonces se construye un array no declarado. Los arrays no declarados, también conocidos por el nombre de "arrays de claves" (hashed arrays), son más generales que los arrays declarados. El usuario no necesita declarar su tamaño máximo y pueden ir creciendo de forma dinámica. Los subíndices de los arrays no declarados no necesitan ser necesariamente números. Sin embargo, a menos que un array tenga sus elementos dispersos, probablemente sea más eficiente declararlo siempre que sea posible antes que dejarlo como no declarado. La función `array` puede utilizarse para transformar un array no declarado a a uno declarado.

`arrayapply` (*A*, [*i_1*, ..., *i_n*]) [Función]

Evalúa `A [i_1, ..., i_n]`, donde A es un array y *i_1*, ..., *i_n* son enteros.

Esto es como `apply`, excepto por el hecho de que el primer argumento es un array en lugar de una función.

`arrayinfo` (*A*) [Función]

Devuelve información sobre el array$A$. El argumento A puede ser un array declarado o no declarado, una función array o una función subindicada.

En el caso de arrays declarados, `arrayinfo` devuelve una lista que contiene el átomo `declared`, el número de dimensiones y el tamaño de cada dimensión. Los elementos del array, tanto los que tienen valores asignados como los que no, son devueltos por `listarray`.

En el caso de arrays no declarados (*hashed arrays*), `arrayinfo` devuelve una lista que contiene el átomo `hashed`, el número de subíndices y los subíndices de aquellos elementos que guarden un valor. Los valores son devueltos por `listarray`.

En el caso de funciones array, `arrayinfo` devuelve una lista que contiene el átomo `hashed`, el número de subíndices y los subíndices para los que la función tiene valores almacenados. Los valores almacenados de la función array son devueltos por `listarray`.

En el caso de funciones subindicadas, `arrayinfo` devuelve una lista que contiene el átomo `hashed`, el número de subíndices y los subíndices para los que hay expresiones lambda. Las expresiones lambda son devueltas por `listarray`.

Ejemplos:

`arrayinfo` y `listarray` aplicadas a una 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` y `listarray` aplicadas a una array no declarado (*hashed arrays*).

```
(%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` y `listarray` aplicadas a una función 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` y `listarray` aplicadas a una función subindicada.

```
(%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*]) [Función]

El resultado es una referencia a array no evaluada.

Devuelve la expresión *name* [*i_1*, ..., *i_n*].

Esta función es similar a `funmake`, excepto que el valor retornado es referencia a un array no evaluado, en lugar de una llamada a una función no evaluada.

Ejemplos:

```
(%i1) arraymake (A, [1]);
                               A
(%o1)                          1
(%i2) arraymake (A, [k]);
                               A
(%o2)                          k
(%i3) arraymake (A, [i, j, 3]);
                             A
(%o3)                        i, j, 3
(%i4) array (A, fixnum, 10);
                               A
(%o4)
(%i5) fillarray (A, makelist (i^2, i, 1, 11));
                               A
(%o5)
(%i6) arraymake (A, [5]);
```

```
(%o6)                           A
                                 5
(%i7) ''%;
(%o7)                           36
(%i8) L : [a, b, c, d, e];
(%o8)                     [a, b, c, d, e]
(%i9) arraymake ('L, [n]);
(%o9)                           L
                                 n
(%i10) ''%, n = 3;
(%o10)                           c
(%i11) A2 : make_array (fixnum, 10);
(%o11)          {Array:  #(0 0 0 0 0 0 0 0 0 0)}
(%i12) fillarray (A2, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
(%o12)          {Array:  #(1 2 3 4 5 6 7 8 9 10)}
(%i13) arraymake ('A2, [8]);
(%o13)                          A2
                                  8
(%i14) ''%;
(%o14)                           9
```

`arrays` [Variable del sistema]

Valor por defecto: `[]` La variable `arrays` es una lista con todos los arrays que han sido alojados, lo que comprende a los arrays declarados por `array`, a los no declarados (*hashed arrays*) construidos implícitamente (asignando algo al elemento de un array) y a las funciones array definidas mediante `:=` y `define`. Los arrays definidos mediante `make_array` no se incluyen en este grupo.

Véanse también `array`, `arrayapply`, `arrayinfo`, `arraymake`, `fillarray`, `listarray` y `rearray`.

Ejemplos:

```
(%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]
```

`arraysetapply` (*A*, [*i_1*, ..., *i_n*], *x*) [Función]

Asigna *x* a `A[i_1, ..., i_n]`, siendo *A* un array y *i_1*, ..., *i_n* enteros.

`arraysetapply` evalúa sus argumentos.

`fillarray` (*A*, *B*) [Función]

Rellena el array *A* con los valores de *B*, que puede ser una lista o array.

Si se ha declarado *A* de un determinado tipo en el momento de su creación, sólo podrá contener elementos de ese tipo, produciéndose un error en caso de intentar asignarle un elemento de tipo distinto.

Si las dimensiones de los arrays *A* y *B* son diferentes, *A* se rellena según el orden de las filas. Si no hay suficientes elementos en *B* el último elemento se utiliza para cubrir el resto de *A*. Si hay demasiados, los elementos sobrantes son ignorados.

La función `fillarray` devuelve su primer argumento.

Ejemplos:

Creación de un array de 9 elementos y posterior relleno a partir de una lista.

```
(%i1) array (a1, fixnum, 8);
(%o1)                           a1
(%i2) listarray (a1);
(%o2)              [0, 0, 0, 0, 0, 0, 0, 0, 0]
(%i3) fillarray (a1, [1, 2, 3, 4, 5, 6, 7, 8, 9]);
(%o3)                           a1
(%i4) listarray (a1);
(%o4)              [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Si no hay suficientes elementos para cubrir el array, se repite el último elemento. Si hay demasiados, los elementos sobrantes son ignorados.

```
(%i1) a2 : make_array (fixnum, 8);
(%o1)             {Array:  #(0 0 0 0 0 0 0 0)}
(%i2) fillarray (a2, [1, 2, 3, 4, 5]);
(%o2)             {Array:  #(1 2 3 4 5 5 5 5)}
(%i3) fillarray (a2, [4]);
(%o3)             {Array:  #(4 4 4 4 4 4 4 4)}
(%i4) fillarray (a2, makelist (i, i, 1, 100));
(%o4)             {Array:  #(1 2 3 4 5 6 7 8)}
```

Arrays multidimensionales se rellenan según el orden de las filas.

```
(%i1) a3 : make_array (fixnum, 2, 5);
(%o1)        {Array:  #2A((0 0 0 0 0) (0 0 0 0 0))}
(%i2) fillarray (a3, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
(%o2)        {Array:  #2A((1 2 3 4 5) (6 7 8 9 10))}
(%i3) a4 : make_array (fixnum, 5, 2);
(%o3)     {Array:  #2A((0 0) (0 0) (0 0) (0 0) (0 0))}
(%i4) fillarray (a4, a3);
(%o4)     {Array:  #2A((1 2) (3 4) (5 6) (7 8) (9 10))}
```

`listarray` (*A*) [Función]

Devuelve una lista con los elementos del array *A*. El argumento *A* puede ser un array declarado o no declarado, una función array o una función subindicada.

Los elementos se ordenan en primera instancia respecto del primer índice, después respecto del segundo índice y así sucesivamente. La ordenación de los índices es la misma que la establecida por `orderless`.

En el caso de arrays no declarados, funciones array y funciones subindicadas, los elementos corresponden a los índices devueltos por `arrayinfo`.

Los elementos de los arrays declarados que no tienen valores asignados (excepto `fixnum` y `flonum`) se devuelven como `#####`. Los elementos sin valores asignados de los arrays `fixnum` y `flonum` son devueltos como 0 y 0.0, respectivamente. Los elementos sin valor asignado de los arrays no declarados, funciones array y funciones subindicadas no son devueltos.

Ejemplos:

`listarray` y `arrayinfo` aplicadas a un 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` y `arrayinfo` aplicadas a un array no declarado (*hashed array*).

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

`listarray` y `arrayinfo` aplicadas a una función 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` y `arrayinfo` aplicadas a una función subindicada.

```
(%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` (*tipo*, *dim_1*, ..., *dim_n*) [Función]

Construye y devuelve un array de Lisp. El argumento *tipo* puede ser `any`, `flonum`, `fixnum`, `hashed` o `functional`. Hay n índices, y el índice i-ésimo va de 0 a $dim\_i - 1$.

La ventaja de `make_array` sobre `array` estriba en que el valor retornado no tiene nombre, y una vez que un puntero deja de referenciarlo, el valor desaparece. Por ejemplo, si `y: make_array (...)` entonces `y` apunta a un objeto que ocupa cierto espacio en la memoria, pero después de `y: false`, `y` ya no apunta al objeto, por lo que éste puede ser considerado basura y posteriormente eliminado.

Ejemplos:

```
(%i1) A1 : make_array (fixnum, 10);
(%o1)           {Array:  #(0 0 0 0 0 0 0 0 0 0)}
(%i2) A1 [8] : 1729;
(%o2)                          1729
(%i3) A1;
(%o3)          {Array:  #(0 0 0 0 0 0 0 0 1729 0)}
(%i4) A2 : make_array (flonum, 10);
(%o4) {Array:  #(0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0)}
(%i5) A2 [2] : 2.718281828;
(%o5)                       2.718281828
(%i6) A2;
(%o6)
     {Array:  #(0.0 0.0 2.718281828 0.0 0.0 0.0 0.0 0.0 0.0 0.0)}
(%i7) A3 : make_array (any, 10);
```

```
(%o7) {Array:  #(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)}
(%i8) A3 [4] : x - y - z;
(%o8)                        - z - y + x
(%i9) A3;
(%o9) {Array:  #(NIL NIL NIL NIL ((MPLUS SIMP) $X ((MTIMES SIMP)\
 -1 $Y) ((MTIMES SIMP) -1 $Z))
  NIL NIL NIL NIL NIL)}
(%i10) A4 : make_array (fixnum, 2, 3, 5);
(%o10) {Array:  #3A(((0 0 0 0 0) (0 0 0 0 0) (0 0 0 0 0)) ((0 0 \
0 0 0) (0 0 0 0 0) (0 0 0 0 0)))}
(%i11) fillarray (A4, makelist (i, i, 1, 2*3*5));
(%o11) {Array:  #3A(((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15))
    ((16 17 18 19 20) (21 22 23 24 25) (26 27 28 29 30)))}
(%i12) A4 [0, 2, 1];
(%o12)                          12
```

`rearray` (*A*, *dim_1*, ..., *dim_n*) [Función]

Cambia las dimensiones de un array. El nuevo array será rellenado con los elementos del viejo según el orden de las filas. Si el array antiguo era demasiado pequeño, los elementos restantes se rellenan con `false`, `0.0` o `0`, dependiendo del tipo del array. El tipo del array no se puede cambiar.

`remarray` (*A_1*, ..., *A_n*) [Función]

`remarray` (*all*) [Función]

Borra los arrays y las funciones relacionadas con ellos, liberando el espacio de memoria ocupado. Los argumentos pueden ser arrays declarados, arrays no declarados (*hashed arrays*), funciones array y funciones subindicadas.

La llamada `remarray (all)` borra todos los elementos de la lista global `arrays`.

La función `remarray` devuelve la lista de los arrays borrados.

La función `remarray` no evalúa sus argumentos.

`subvar` (*x*, *i*) [Función]

Evalúa la expresión subindicada *x*`[`*i*`]`.

La función `subvar` evalúa sus argumentos.

La instrucción `arraymake (`*x*`, [`*i*`])` construye la expresión *x*`[`*i*`]`, pero no la evalúa.

Ejemplos:

```
(%i1) x : foo $

(%i2) i : 3 $

(%i3) subvar (x, i);
(%o3)                          foo
                                  3
(%i4) foo : [aa, bb, cc, dd, ee]$

(%i5) subvar (x, i);
```

```
(%o5)                           cc
(%i6) arraymake (x, [i]);
(%o6)                          foo
                                  3
(%i7) ''%;
(%o7)                           cc
```

`subvarp` (*expr*) [Función]

Devuelve `true` si *expr* es una variable subindicada, como `a[i]`.

`use_fast_arrays` [Variable opcional]

Valor por defecto: `false`

Si `use_fast_arrays` vale `true` entonces tan solo se reconocen dos tipos de arrays.

5.6 Estructuras

5.6.1 Introducción a las estructuras

Maxima dispone de estructuras para la agregación de datos. Una estructura es una expresión en la que los argumentos se definen con un nombre (nombre del campo) y la estructura en su conjunto se define por medio de su operador (nombre de la estructura). Los valores dados a los campos pueden ser expresiones cualesquiera.

Una estructura se define con la función `defstruct`, guardando la variable `structures` la lista de todas las estructuras definidas por el usuario. El operador `@` permite hacer referencias a los campos de la estructura. Con `kill(`*S*`)` se borra la estructura *S* completa y `kill(`*x*`@a)` elimina la asignación actual del campo *a* en la estructura *x*.

En la impresión por consola (con `display2d` igual a `true`), las estructuras se representan con las asignaciones de los campos en forma de ecuación, con el nombre del campo a la izquierda y su valor asociado a la derecha. En la impresión unidimensional (mediante `grind` o dándole a `display2d` el valor `false`), las estructuras se escriben sin los nombres de los campos.

No es posible utilizar el nombre de un campo como nombre de función, pero el valor de un campo sí puede ser una expresión lambda. Tampoco es posible restringir los valores de los campos a tipos de datos concretos, siendo el caso que a cada campo se le puede asignar cualquier tipo de expresión. Por último, no es posible hacer que ciertos campos sean o no accesibles desde determinados contextos, ya que todos los campos son siempre visibles.

5.6.2 Funciones y variables para las estructuras

`structures` [Variable global]

`structures` es la lista que contiene las estructuras definidas por el usuario con `defstruct`.

`defstruct (`*S*`(`*a_1*`, ..., `*a_n*`))` [Función]
`defstruct (`*S*`(`*a_1* `=` *v_1*`, ..., `*a_n* `=` *v_n*`))` [Función]

Define una estructura, la cual es una lista de nombres de campos *a_1*, ..., *a_n* asociados a un símbolo *S*. Todo individuo de una estructura dada consiste en una expresión con operador *S* y exactamente `n` argumentos. La sentencia `new(`*S*`)` crea un nuevo individuo con estructura *S*.

Un argumento consistente en un símbolo *a* especifica el nombre de un campo. Un argumento consistente en una ecuación `a = v` especifica el nombre del campo *a* junto con su valor por defecto *v*. El valor por defecto puede ser cualquier tipo de expresión.

La llamada a `defstruct` añade *S* a la lista de estructuras definidas por el usuario, `structures`.

La sentencia `kill(`*S*`)` borra *S* de la lista de estructuras definidas por el usuario y elimina la definición de la estructura.

Ejemplos:

```
(%i1) defstruct (foo (a, b, c));
(%o1)                     [foo(a, b, c)]
(%i2) structures;
```

```
(%o2)                     [foo(a, b, c)]
(%i3) new (foo);
(%o3)                      foo(a, b, c)
(%i4) defstruct (bar (v, w, x = 123, y = %pi));
(%o4)             [bar(v, w, x = 123, y = %pi)]
(%i5) structures;
(%o5)      [foo(a, b, c), bar(v, w, x = 123, y = %pi)]
(%i6) new (bar);
(%o6)              bar(v, w, x = 123, y = %pi)
(%i7) kill (foo);
(%o7)                          done
(%i8) structures;
(%o8)             [bar(v, w, x = 123, y = %pi)]
```

`new` (*S*) [Función]
`new` (*S* (*v_1*, ..., *v_n*)) [Función]

`new` crea nuevos individuos de estructuras declaradas.

La sentencia `new(`*S*`)` crea un nuevo individuo de estructura *S* en el que cada campo toma su valor por defecto, si este fue definido, o sin valor alguno en caso de que no se haya fijado un valor por defecto en la definición de la estructura.

La sentencia `new(`*S*`(`*v_1*`, ..., `*v_n*`))` crea un nuevo individuo de estructura *S* en el que los campos adquieren los valores *v_1*, ..., *v_n*.

Ejemplos:

```
(%i1) defstruct (foo (w, x = %e, y = 42, z));
(%o1)               [foo(w, x = %e, y = 42, z)]
(%i2) new (foo);
(%o2)                foo(w, x = %e, y = 42, z)
(%i3) new (foo (1, 2, 4, 8));
(%o3)             foo(w = 1, x = 2, y = 4, z = 8)
```

`@` [Operador]

`@` es el operador para acceder a los campos de las estructuras.

La expresión *x*`@ a` hace referencia al valor del campo *a* del individuo *x* de una estructura dada. El nombre del campo no se evalúa.

Si el campo *a* de *x* no tiene valor asignado, *x*`@ a` se evalúa a sí mismo; es decir, devuelve la propia expresión *x*`@ a` tal cual.

La sentencia `kill(x@ a)` elimina el valor del campo *a* de *x*.

Ejemplos:

```
(%i1) defstruct (foo (x, y, z));
(%o1)                      [foo(x, y, z)]
(%i2) u : new (foo (123, a - b, %pi));
(%o2)           foo(x = 123, y = a - b, z = %pi)
(%i3) u@z;
(%o3)                           %pi
(%i4) u@z : %e;
```

```
(%o4)                          %e
(%i5) u;
(%o5)            foo(x = 123, y = a - b, z = %e)
(%i6) kill (u@z);
(%o6)                         done
(%i7) u;
(%o7)              foo(x = 123, y = a - b, z)
(%i8) u@z;
(%o8)                          u@z
```

El nombre del campo no se evalúa.

```
(%i1) defstruct (bar (g, h));
(%o1)                       [bar(g, h)]
(%i2) x : new (bar);
(%o2)                        bar(g, h)
(%i3) x@h : 42;
(%o3)                           42
(%i4) h : 123;
(%o4)                           123
(%i5) x@h;
(%o5)                           42
(%i6) x@h : 19;
(%o6)                           19
(%i7) x;
(%o7)                     bar(g, h = 19)
(%i8) h;
(%o8)                           123
```

6 Operadores

6.1 Introducción a los operadores

Maxima reconoce los operadores aritméticos, relacionales y lógicos usuales de la matemática. Además, Maxima dispone de operadores para la asignación de valores a variables y la definición de funciones. La siguiente tabla muestra los operadores que se describen en este capítulo, en la que se incluye el nombre del operador, el rango de enlace por la izquierda `lbp`, el rango de enlace por la derecha `rbp`, el tipo de operador y un ejemplo, para finalizar, en cada caso, con su formato interno tal como es leída por el analizador sintáctico.

Operador	lbp	rbp	Tipo	Ejemplo	Formato interno
+	100	134	nary	a+b	((mplus) $A $B)
-	100	134	prefix	-a	((mminus) $A)
*	120		nary	a*b	((mtimes) $A $B)
/	120	120	infix	a/b	((mquotient) $A $B)
^	140	139	infix	a^b	((mexpt) $A $B)
**	140	139	infix	a**b	((mexpt) $A $B)
^^	140	139	infix	a^^b	((mncexpt) $A $B)
.	130	129	infix	a.b	((mnctimes) $A $B)
<	80	80	infix	a<b	((mlessp) $A $B)
<=	80	80	infix	a<=b	((mleqp) $A $B)
>	80	80	infix	a>b	((mqreaterp) $A $B)
>=	80	80	infix	a>=b	((mgeqp) $A $B)
not		70	prefix	not a	((mnot) $A)
and	65		nary	a and b	((mand) $A $B)
or	60		nary	a or b	((mor) $A $B)
#	80	80	infix	a#b	((mnotequal) $A $B)
=	80	80	infix	a=b	((mequal) $A $B)
:	180	20	infix	a:b	((msetq) $A $B)
::	180	20	infix	a::b	((mset) $A $B)
:=	180	20	infix	a:=b	((mdefine) $A $B)
::=	180	20	infix	a::=b	((mdefmacro) $A $B)

Con los rangos de enlace de los operadores se definen las reglas de prioridad de cálculo de los mismos. Así, por ejemplo, el analizador sintáctico interpreta la expresión `a + b * c` como `a + (b * c)` , pues el rango de enlace por la izquierda de la multiplicación es mayor que rango de enlace por la izquierda de la suma.

Maxima clasifica los operadores de la siguiente manera:

Prefijo (prefix)
: Los operadores prefijos son unarios con un único operando que se escribe a continuación del operando. Ejemplos son `-` y `not`.

Sufijo (postfix)
: Los operadores sufijos son unarios con un único operando que se escribe precediendo al operando. Un ejemplo es el factorial `!`.

Infijo (infix)
: Los operadores infijos son operadores binarios que necesitan dos operandos, los cuales se colocan uno a cada lado del operador. Ejemplos son el operador para la exponenciación, `^`, y el operador de asignación, `:`.

N-ario (n-ary)
: Los operadores n-arios admiten un número arbitrario de operandos. Son ejemplos la multiplicación, `*`, y la suma, `+`.

Acotador (matchfix)
: Los acotadores son operadores que se utilizan para establecer el comienzo y final de una lista de operandos. Los operadores `[` y `]` son ejemplos de acotadores, que se utilizan para definir una lista tal como `[a, b, ...]`.

No-fijo (nofix)
: Un operador no-fijo carece de operandos. Maxima no tiene operadores internos no-fijos, pero se pueden crear como cuando se escribe `nofix(quit)`, lo que permite obviar el uso de paréntesis, y utilizar simplemente `quit` en lugar de `quit()`, para cerrar una sesión de Maxima.

En la sección dedicada a los operadores definidos por el usuario se describe cómo redefinir los operadores internos de Maxima y cómo crear otros nuevos.

El mecanismo para definir un nuevo operador es sencillo. Tan solo es necesario declarar una función como operador; la función operador puede estar definida o no.

Un ejemplo de operador definido por el usuario es el siguiente. Nótese que la llamada a función `"dd" (a)` equivale a `dd a`, de igual manera que `"<-" (a, b)` también equivale a `a <- b`. Nótese también que las funciones `"dd"` y `"<-"` no están definidas en este ejemplo.

```
(%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
```

La tabla anterior no incluye todos los operadores definidos en Maxima, ya que también lo son `!` para el factorial, `for`, `do` y `while` para programar bucles, o `if`, `then` y `else` para definir condicionales.

Las funciones `remove` y `kill` eliminan propiedades de operadores de un átomo. La llamada `remove ("a", op)` sólo elimina las propiedades de operador de *a*. La llamada `kill ("a")` elimina todas las propiedades de *a*, incluidas las propiedades de operador. Nótese que el nombre del operador debe ir entre comillas.

```
(%i1) infix ("##");
(%o1)                           ##
(%i2) "##" (a, b) := a^b;
                                      b
(%o2)                     a ## b := a
(%i3) 5 ## 3;
(%o3)                           125
(%i4) remove ("##", op);
(%o4)                          done
(%i5) 5 ## 3;
Incorrect syntax: # is not a prefix operator
5 ##
  ^
(%i5) "##" (5, 3);
(%o5)                           125
(%i6) infix ("##");
(%o6)                           ##
(%i7) 5 ## 3;
(%o7)                           125
(%i8) kill ("##");
(%o8)                          done
(%i9) 5 ## 3;
Incorrect syntax: # is not a prefix operator
5 ##
  ^
(%i9) "##" (5, 3);
(%o9)                       ##(5, 3)
```

6.2 Operadores aritméticos

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

Los símbolos `+ * /` y `^` representan la suma, resta, multiplicación, división y exponenciación, respectivamente. Los nombres de estos operadores son `"+" "*" "/"` y `"^"`, que pueden aparecer allá donde se requiera el nombre de una función u operador.

Los símbolos `+` y `-` representan el positivo y negativo unario, siendo los nombres de estos operadores `"+"` y `"-"`, respectivamente.

En Maxima, la resta `a - b` se representa como la suma `a + (- b)`. Expresiones tales como `a + (- b)` se muestran como restas. Maxima reconoce `"-"` tan solo como el operador de negación unaria, no como el nombre del operador de resta binaria.

La división `a / b` se representa en maxima como la multiplicación `a * b^(- 1)`. Expresiones tales como `a * b^(- 1)` se muestran como divisiones. Maxima reconoce `"/"` como el nombre del operador de división.

La suma y la multiplicación son operadores conmutativos n-arios. La división y la exponenciación son operadores no conmutativos binarios.

Maxima ordena los operandos de los operadores conmutativos para formar lo que se conoce como representación canónica. A efectos de almacenamiento interno, la ordenación viene determinada por `orderlessp`. A efectos de presentación de las expresiones, la ordenación de la suma la determina `ordergreatp`, y en el caso de la multiplicación, la ordenación coincide con la del almacenamiento interno.

Los cálculos aritméticos se realizan con números literales (enteros, racionales, decimales ordinarios y decimales grandes). Excepto en el caso de la exponenciación, todas las operaciones aritméticas con números dan lugar a resultados en forma de números. La exponenciación da como resultado un número si alguno de los operandos es decimal ordinario o grande (*bigfloat*), o si el resultado es un entero o racional; en caso contrario, la exponenciación puede expresarse como una raíz cuadrada (`sqrt`), como otra potencia, o simplemente no sufre cambios.

Se produce contagio de los decimales en coma flotante en los cálculos aritméticos: si algún operando es un número decimal grande (*bigfloat*), el resultado será también un número decimal grande; no habiendo decimales grandes, pero sí ordinarios, el resultado srá también un decimal ordinario; de no haber operandos decimales, el resultado será un número racional o entero.

Los cálculos aritméticos son simplificaciones, no evaluaciones, por lo que se realizan en expresiones comentadas.

Las operaciones aritméticas se aplican elemento a elemento en el caso de las listas cuando la variable global `listarith` vale `true`; pero en el caso de las matrices, siempre se aplican elemento a elemento. Cuando un operando es una lista o matriz y otro operando lo es de otro tipo cualquiera, éste se combina con cada uno de los elementos de la lista o matriz.

Ejemplos:

La suma y la multiplicación son operadores conmutativos n-arios. Maxima ordena los operandos para formar lo que se conoce como representación canónica. Los nombres de estos operadores son `"+"` y `"-"`.

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

```
(%o5)                     3 x + 2 a + 19
(%i6) apply ("*", [a, 8, x, 2, 9, x, x, a]);
                                   2  3
(%o6)                        144 a  x
```

La división y la exponenciación son operadores no conmutativos binarios. Los nombres de estos operadores son "/" y "^".

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

La resta y la división se representan internamente en términos de la suma y multiplicación, respectivamente.

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

Los cálculos se realizan con números literales. Se produce el contagio de los números decimales.

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

Los cálculos aritméticos son una simplificación, no una evaluación.

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

Los cálculos aritméticos se realizan elemento a elemento en las listas (según sea el valor de `listarith`) y matrices.

```
(%i1) matrix ([a, x], [h, u]) - matrix ([1, 2], [3, 4]);
                         [ a - 1  x - 2 ]
(%o1)                    [              ]
                         [ h - 3  u - 4 ]
(%i2) 5 * matrix ([a, x], [h, u]);
                           [ 5 a  5 x ]
(%o2)                      [          ]
                           [ 5 h  5 u ]
(%i3) listarith : false;
(%o3)                          false
(%i4) [a, c, m, t] / [1, 7, 2, 9];
                           [a, c, m, t]
(%o4)                      ------------
                           [1, 7, 2, 9]
(%i5) [a, c, m, t] ^ x;
                                       x
(%o5)                      [a, c, m, t]
(%i6) listarith : true;
(%o6)                          true
(%i7) [a, c, m, t] / [1, 7, 2, 9];
                               c  m  t
(%o7)                      [a, -, -, -]
                               7  2  9
(%i8) [a, c, m, t] ^ x;
                            x   x   x   x
(%o8)                    [a , c , m , t ]
```

**** [Operador]

Operador de exponenciación. Maxima identifica `**` con el operador `^` en la entrada de expresiones, pero se representa como `^` en las salidas no formateadas (`display2d=false`), o colocando un superíndice en la salida formateada (`display2d=true`).

La función `fortran` representa el operador de exponenciación con `**`, tanto si se ha introducido como `**` o como `^`.

Ejemplos:

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

```
(%o4)                          done
```

^^ [Operator]

Operador de exponenciación no conmutativa. Se trata del operador de exponenciación correspondiente a la multiplicación no conmutativa `.`, del mismo modo que el operador de exponenciación ordinario `^` se corresponde con la multiplicación conmutativa `*`.

La exponenciación no conmutativa se representa como `^^` en las salidas no formateadas (`display2d=false`), o colocando un superíndice entre ángulos (`< >`) en la salida formateada (`display2d=true`).

Ejemplos:

```
(%i1) a . a . b . b . b + a * a * a * b * b;
                         3  2    <2>    <3>
(%o1)                   a  b  + a    . b
(%i2) string (a . a . b . b . b + a * a * a * b * b);
(%o2)                   a^3*b^2+a^^2 . b^^3
```

. [Operador]

El operador punto, para multiplicación de matrices (no-conmutativo). Cuando `"."` se usa de esta forma, se dejarán espacios a ambos lados de éste, como en `A . B`. Así se evita que se confunda con el punto decimal de los números.

Véanse: `dot`, `dot0nscsimp`, `dot0simp`, `dot1simp`, `dotassoc`, `dotconstrules`, `dotdistrib`, `dotexptsimp`, `dotident` y `dotscrules`.

6.3 Operadores relacionales

< [Operator]
<= [Operator]
>= [Operator]
> [Operator]

Los símbolos `<`, `<=`, `>=` y `>` representan menor que, menor o igual que, mayor o igual que y mayor que, respectivamente. Los nombres de estos operadores son `"<"` `"<="` `">="` y `">"`, que pueden aparecer allá donde se requiera el nombre de una función u operador.

Estos operadores relacionales son todos operadores binarios. Maxima no reconoce expresiones del estilo `a < b < c`.

Las expresiones relacionales devuelven valores booleanos haciendo uso de las funciones `is` o `maybe`, así como de las sentencias condicionales `if`, `while` y `unless`. Las expresiones relacionales no se evalúan de otra manera, aunque sus argumentos sí sean evaluados.

Cuando una expresión relacional no pueda ser evaluada a `true` o `false`, el comportamiento de `is` y de `if` estará controlado por la variable global `prederror`. Si `prederror` toma el valor `true`, `is` y `if` emiten un mensaje de error. Si `prederror` toma el valor `false`, `is` devuelve `unknown` y `if` devuelve una expresión condicional parcialmente evaluada.

`maybe` se comporta siempre como si `prederror` fuese `false`, al tiempo que `while` y `unless` se comportan siempre como si `prederror` fuese `true`.

Los operadores relacionales no se distribuyen sobre listas ni sobre cualesquiera otros tipos de estructuras de datos.

Véanse también `=`, `#`, `equal` y `notequal`.

Ejemplos:

Las expresiones relacionales se reducen a valores booleanos a través de ciertas funciones y sentencias condicionales.

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

Las expresiones relacionales no se evalúan de otra manera, aunque sus argumentos sí sean evaluados.

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

6.4 Operadores lógicos

`and` [Operador]

Operador de conjunción lógica. El operador `and` es un operador infijo `n`-ario; sus operandos son expresiones booleanas y su resultado es un valor lógico.

El operador `and` impone la evaluación (igual que `is`) de uno o más operandos, y puede forzar la evaluación de todos los operandos.

Los operandos se evalúan en el orden en el que aparecen; sólo evalúa tantos operandos como sean necesarios para determinar el resultado. Si algún operando vale `false`, el resultado es `false` y ya no se evalúan más operandos.

La variable global `prederror` controla el comportamiento de `and` cuando la evaluación de un operando no da como resultado `true` o `false`; `and` imprime un mensaje de error cuando `prederror` vale `true`. Cuando los operandos devuelven un valor diferente a `true` o `false` al ser evaluados, el resultado es una expresión booleana.

El operador `and` no es conmutativo: `a and b` puede no ser igual a `b and a` debido al tratamiento de operandos indeterminados.

`not` [Operador]

Operador de negación lógica. El operador `not` es un operador prefijo; su operando es una expresión booleana y su resultado es un valor lógico.

El operador `not` impone la evaluación (igual que `is`) de su operando.

La variable global `prederror` controla el comportamiento de `not` cuando la evaluación de su operando no da como resultado `true` o `false`; `not` imprime un mensaje de error cuando `prederror` vale `true`. Cuando los operandos devuelven un valor diferente a `true` o `false` al ser evaluados, el resultado es una expresión booleana.

`or` [Operador]

Operador de disyunción lógica. El operador `or` es un operador infijo `n`-ario; sus operandos son expresiones booleanas y su resultado es un valor lógico.

El operador `or` impone la evaluación (igual que `is`) de uno o más operandos, y puede forzar la evaluación de todos los operandos.

Los operandos se evalúan en el orden en el que aparecen; `or` sólo evalúa tantos operandos como sean necesarios para determinar el resultado. Si un operando vale `true`, el resultado es `true` y ya no se evalúan más operandos.

La variable global `prederror` controla el comportamiento de `or` cuando la evaluación de un operando no da como resultado `true` o `false`; `or` imprime un mensaje de error cuando `prederror` vale `true`. Cuando los operandos devuelven un valor diferente a `true` o `false` al ser evaluados, el resultado es una expresión booleana.

El operador `or` no es conmutativo: `a or b` puede no ser igual a `b or a` debido al tratamiento de operandos indeterminados.

6.5 Operadores para ecuaciones

`#` [Operador]

Representa la negación de la igualdad sintáctica `=`.

Nótese que debido a las reglas de evaluación de expresiones de tipo predicado (en concreto debido a que `not` *expr* obliga a la evaluación previa de *expr*), `not a = b` equivale a `is(a # b)`, pero no a `a # b`.

Ejemplos:

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

`=` [Operador]

Operador de ecuación.

La expresión `a = b` representa una ecuación sin evaluar, la cual puede verificarse o no. Las ecuaciones sin evaluar pueden aparecer como argumentos de `solve`, `algsys` y de algunas otras funciones.

La función `is` evalúa el operador `=` a un resultado booleano; `is(a = b)` asigna un valor de verdad a `a = b`, siendo `true` si *a* y *b* son idénticos, lo cual acontece si ambos *a* y *b* son átomos idénticos, o si no siendo átomos, sus operadores y argumentos respectivos son idénticos; en caso contrario, `is(a = b)` devuelve el valor `false`. Nunca se devuelve el valor `unknown`. Cuando `is(a = b)` toma el valor `true`, se dice que *a* y *b* son sintácticamente iguales, no expresiones equivalentes, para las cuales `is(equal(a, b))` devuelve `true`. Las expresiones pueden ser equivalentes, pero no sintácticamente iguales.

La negación de `=` se representa por `#`. Como en el caso de `=`, la expresión `a # b` no está evaluada; sin embargo, `is(a # b)` evalúa `a # b` a `true` o `false`.

Además de `is`, hay otros operadores que evalúan `=` y `#` a `true` o `false`; a saber, `if`, `and`, `or` y `not`.

Nótese que debido a las reglas de evaluación de expresiones de tipo predicado (en concreto debido a que `not` *expr* obliga a la evaluación previa de *expr*), `not a = b` equivale a `is(a # b)`, pero no a `a # b`.

Las funciones `rhs` y `lhs` devuelven los miembros derecho e izquierdo, respectivamente, de una ecuación o inecuación.

Véanse también `equal` y `notequal`.

Ejemplos:

La expresión `a = b` representa una ecuación sin evaluar, la cual puede verificarse o no.

```
(%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)` evalúa `a = b` a `true` si *a* y *b* son sintácticamente iguales (es decir, idénticas). Las expresiones pueden ser equivalentes, pero no sintácticamente iguales.

```
(%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]
```

Algunos operadores evalúan `=` y `#` a `true` o `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]
```

Debido a que `not` *expr* obliga a la evaluación previa de *expr*, `not a = b` equivale 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
```

6.6 Operadores de asignación

: [Operador]

Operador de asignación.

Cuando el miembro de la izquierda es una variable simple (no subindicada), `:` evalúa la expresión de la derecha y asigna ese valor a la variable del lado izquierdo.

Cuando en el lado izquierdo hay un elemento subindicado correspondiente a una lista, matriz, array declarado de Maxima o array de Lisp, la expresión de la derecha se asigna a ese elemento. El subíndice debe hacer referencia a un elemento ya existente, ya que los objetos anteriores no pueden ampliarse nombrando elementos no existentes.

Cuando en el lado izquierdo hay un elemento subindicado correspondiente a un array no declarado de Maxima, la expresión de la derecha se asigna a ese elemento en caso de que ya exista, o a un nuevo elemento, si éste todavía no existe.

Cuando el miembro de la izquierda es una lista de átomos y/o variables subindicadas, el miembro derecho debe evaluar también a una lista, cuyos elementos serán asignados en paralelo a las variables de la lista de la izquierda.

Véanse también `kill` y `remvalue`, que deshacen las asociaciones hechas por el operador `:`.

Ejemplos:

Asignación a una variable simple.

```
(%i1) a;
(%o1)                           a
(%i2) a : 123;
(%o2)                          123
(%i3) a;
(%o3)                          123
```

Asignación a un elemento de una lista.

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

La asignación crea un array no declarado.

```
(%i1) c[99] : 789;
(%o1)                          789
(%i2) c[99];
(%o2)                          789
(%i3) c;
(%o3)                           c
(%i4) arrayinfo (c);
(%o4)                  [hashed, 1, [99]]
(%i5) listarray (c);
(%o5)                         [789]
```

Asignación múltiple.

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

La asignación múltiple se hace en paralelo. Los valores de `a` y `b` se intercambian en este ejemplo.

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

:: [Operador]

Operador de asignación.

El operador :: es similar a :, excepto que :: evalúa ambos miembros, tanto el derecho como el izquierdo.

Ejemplos:

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

::= [Operador]

El operador de definición de macros ::= define una función (llamada macro por razones históricas) que no evalúa sus argumentos, siendo la expresión que retorna (llamada "macroexpansión") evaluada dentro del contexto desde el cual se ha invocado la macro. En cualquier otro sentido, una función macro es igual que una función ordinaria.

macroexpand devuelve la expresión que a su vez fue devuelta por una macro (sin evaluar la expresión); macroexpand (foo (x)) seguida de ''% es equivalente a foo (x) si foo es una función macro.

::= coloca el nombre de la nueva función macro en la lista global macros. Por otro lado, las funciones kill, remove y remfunction borran las definiciones de las funciones macro y eliminan sus nombres de la lista macros.

Las funciones fundef y dispfun devuelven la definición de una función macro y le asignan una etiqueta, respectivamente.

Las funciones macro normalmente contienen expresiones buildq y splice para construir una expresión, que luego será evaluada.

Ejemplos:

Una función macro no evalúa sus argumentos, por lo que el mensaje (1) muestra y - z, no el valor de y - z. La macroexpansión (es decir, la expresión no evaluada '(print ("(2) x is equal to", x)) se evalúa en el contexto desde el cual se produjo la llamada a la macro, imprimiendo el mensaje (2).

```
(%i1) x: %pi$
```

```
(%i2) y: 1234$

(%i3) z: 1729 * w$

(%i4) printq1 (x) ::= block (print ("(1) x is equal to", x),
'(print ("(2) x is equal to", x)))$

(%i5) printq1 (y - z);
(1) x is equal to y - z
(2) x is equal to %pi
(%o5)                                %pi
```

Una función ordinaria evalúa sus argumentos, por lo que el mensaje (1) muestra el valor de `y - z`. El valor de retorno no se evalúa, por lo que el mensaje (2) no se imprime hasta la evaluación explícita `''%`.

```
(%i1) x: %pi$

(%i2) y: 1234$

(%i3) z: 1729 * w$

(%i4) printe1 (x) := block (print ("(1) x is equal to", x),
'(print ("(2) x is equal to", x)))$

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

`macroexpand` devuelve la macroexpansión; `macroexpand (foo (x))` seguida de `''%` es equivalente a `foo (x)` si `foo` es una función macro.

```
(%i1) x: %pi$

(%i2) y: 1234$

(%i3) z: 1729 * w$

(%i4) g (x) ::= buildq ([x], print ("x is equal to", x))$

(%i5) macroexpand (g (y - z));
(%o5)                    print(x is equal to, y - z)
(%i6) ''%;
x is equal to 1234 - 1729 w
(%o6)                           1234 - 1729 w
(%i7) g (y - z);
x is equal to 1234 - 1729 w
```

```
(%o7)                                1234 - 1729 w
```

`:=` [Operador]

El operador de definición de funciones. La expresión *`f(x_1, ..., x_n) := expr`* define una función de nombre *f* con argumentos *x_1*, ..., *x_n* y cuerpo *expr*. El operador `:=` no evalúa el cuerpo de la función (a menos que se indique lo contrario mediante el operador comilla-comilla `''`). La función así definida puede ser una función ordinaria de Maxima (con argumentos encerrados entre paréntesis) o una función array (con argumentos encerrados entre corchetes).

Cuando el último o único argumento *x_n* es una lista de un solo elemento, la función definida por `:=` acepta un número variable de argumentos. Los valores de los argumentos se asignan uno a uno a los argumentos formales *x_1*, ..., *x_(n - 1)*, y cualesquiera otros valores de argumentos, si existen, se asignan a *x_n* en forma de lista.

Todas las definiciones de funciones aparecen en el mismo espacio de nombres; definiendo una función `f` dentro de otra función `g` no limita automáticamente el alcance de `f` a `g`. No obstante, `local(f)` hace que la función `f` sea efectiva solamente dentro del bloque o empaquetado de expresiones en la que aparece `local`.

Si un argumento formal *x_k* es un símbolo afectado por el operador comilla (expresión nominal), la función definida por `:=` no evalúa el correspondiente valor de argumento. En cualquier otro caso, los argumentos que se pasan son evaluados.

Véanse también `define` y `::=`.

Ejemplos:

`:=` no evalúa el cuerpo de la función (a menos que se indique lo contrario mediante el operador comilla-comilla `''`).

```
(%i1) expr : cos(y) - sin(x);
(%o1)                      cos(y) - sin(x)
(%i2) F1 (x, y) := expr;
(%o2)                    F1(x, y) := expr
(%i3) F1 (a, b);
(%o3)                      cos(y) - sin(x)
(%i4) F2 (x, y) := ''expr;
(%o4)                F2(x, y) := cos(y) - sin(x)
(%i5) F2 (a, b);
(%o5)                      cos(b) - sin(a)
```

La función así definida puede ser una función ordinaria de Maxima o una función array.

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

Cuando el último o único argumento *x_n* es una lista de un solo elemento, la función definida por `:=` acepta un número variable de argumentos.

```
(%i1) H ([L]) := apply ("+", L);
(%o1)                 H([L]) := apply("+", L)
```

```
(%i2) H (a, b, c);
(%o2)                          c + b + a
```

`local` define una función como local.

```
(%i1) foo (x) := 1 - x;
(%o1)                     foo(x) := 1 - x
(%i2) foo (100);
(%o2)                            - 99
(%i3) block (local (foo), foo (x) := 2 * x, foo (100));
(%o3)                             200
(%i4) foo (100);
(%o4)                            - 99
```

6.7 Operadores definidos por el usuario

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

Declara *op* como operador infijo. Un operador infijo es una función de dos argumentos, con el nombre de la función escrito entre sus argumentos. Por ejemplo, el operador de sustracción `-` es un operador infijo.

`infix` (*op*) declara *op* como operador infijo con fuerzas de ligadura por la izquierda y por la derecha iguales a 180, que es el valor por defecto, y partes izquierda y derecha iguales a `any`.

`infix` (*op*, *lbp*, *rbp*) declara *op* como operador infijo con fuerzas de ligadura por la izquierda y por la derecha declaradas en los argumentos, siendo las partes izquierda y derecha iguales a `any`.

`infix` (*op*, *lbp*, *rbp*, *lpos*, *rpos*, *pos*) declara *op* como operador infijo con fuerzas de ligadura por la izquierda y por la derecha, junto con los tipos de expresiones correspondientes a *lpos*, *rpos* y *pos*, que son el operando de la izquierda, el de la derecha y el operador del resultado; los tipos reconocidos son: `expr`, `clause` y `any`, que indican expresión algebraica, expresión booleana o cualquier otra, respectivamente. Maxima puede detectar algunos errores sintácticos comparando los tipos declarados con los de la expresión actual.

La precedencia de *op* con respecto a otros operadores deriva de las fuerzas de ligadura de los operadores en cuestión. Si las fuerzas de ligadura a izquierda y derecha de *op* son ambas mayores que las fuerzas de ligadura a izquierda y derecha de otro operador, entonces *op* tiene preferencia sobre el otro operador. Si las fuerzas de ligadura no son ambas mayores o menores, se aplican otras relaciones más complejas.

La asociatividad de *op* depende de las fuerzas de ligadura. Una mayor fuerza de ligadura a la izquierda (*lbp*) implica que *op* sea evaluado antes que otros operadores a su izquierda en la expresión, mientras que mayor fuerza de ligadura a la derecha (*rbp*) implica que *op* sea evaluado antes que otros operadores a su derecha en la expresión. Así, si *lbp* es mayor, *op* es asociativo por la derecha, mientras que si *rbp* es mayor, *op* es asociativo por la izquierda.

Véase también `Syntax`.

Ejemplos:

Si las fuerzas de ligadura a izquierda y derecha de *op* son ambas mayores que las fuerzas de ligadura a izquierda y derecha de otro operador, entonces *op* tiene preferencia sobre el otro operador.

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

Mayor *lbp* hace a *op* asociativo por la derecha, mientras que mayor *rbp* hace a *op* asociativo por la izquierda.

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

Maxima puede detectar algunos errores sintácticos comparando los tipos declarados con los de la expresión actual.

```
(%i1) infix ("##", 100, 99, expr, expr, expr);
(%o1)                           ##
(%i2) if x ## y then 1 else 0;
Incorrect syntax: Found algebraic expression where logical expression expected
if x ## y then
             ^
(%i2) infix ("##", 100, 99, expr, expr, clause);
(%o2)                           ##
(%i3) if x ## y then 1 else 0;
(%o3)                if x ## y then 1 else 0
```

`matchfix` (*ldelimiter*, *rdelimiter*) [Función]
`matchfix` (*ldelimiter*, *rdelimiter*, *arg_pos*, *pos*) [Función]

Declara un operador "matchfix" con delimitadores a la izquierda y derecha, *ldelimiter* y *rdelimiter*, respectivamente. Los delimitadores son cadenas alfanuméricas.

Un operador "matchfix" es una función con un número arbitrario de argumentos, de manera que los argumentos se presentan entre los delimitadores de la izquierda y derecha. Los delimitadores pueden ser cualquier tipo de cadena, en tanto que el analizador sintáctico pueda distinguirlos de los operandos y de expresiones con operadores. En la práctica esto excluye delimitadores como `%`, `,`, `$` y `;`, necesitando aislar los delimitadores con espacios en blanco. El delimitador de la derecha puede ser igual o diferente del de la izquierda.

Un delimitador de la izquierda sólo puede asociarse con un único delimitador de la derecha; dos operadores "matchfix" diferentes no pueden tener el mismo delimitador por la izquierda.

Un operador ya existente puede declararse como operador "matchfix" sin necesidad de que cambie el resto de propiedades. En particular, los operadores de Maxima tales como la suma `+` pueden ser declarados como "matchfix".

La llamada `matchfix (`*`ldelimiter`*`, `*`rdelimiter`*`, `*`arg_pos`*`, `*`pos`*`)` declara el argumento *arg_pos* y el resultado *pos*, así como los delimitadores *ldelimiter* y *rdelimiter*.

Los argumentos *arg_pos* y *pos* son tipos de funciones, reconociéndose como tales: `expr`, `clause` y `any`, los cuales hacen referencia a una expresión algebraica, booleana o de cualquier otro tipo, respectivamente. Maxima puede detectar ciertos errores sintácticos comparando el tipo de expresión declarado con el de la expresión actual.

La función que ejecutará una operación "matchfix" será una típica función definida por el usuario. La función de operador se define por el método habitual con `:=` o `define`. Los argumentos pueden escribirse entre los delimitadores, o con el delimitador izquierdo como una cadena precedida de apóstrofo y seguidamente los argumentos entre paréntesis. La llamada `dispfun (`*`ldelimiter`*`)` muestra la definición de la función.

El único operador "matchfix" de Maxima es el constructor de listas `[ ]`. Los paréntesis `( )` y las comillas dobles `" "` actúan como operadores "matchfix", pero son tratados como operadores "matchfix" por el analizador sintáctico de Maxima.

Ejemplos:

- Los delimitadores pueden ser practicamente cualquier cadena.

```
(%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~
```

- Los operadores "matchfix" son funciones definidas por el usuario.

```
(%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
```

nary (*op*) [Función]
nary (*op*, *bp*, *arg_pos*, *pos*) [Función]

Un operador n-ario denota una función con un número arbitrario de argumentos entre los que se intercal el símbolo del operador, como en `A+B+C`. La instrucción `nary("x")` declara `x` como operador n-ario. Las funciones se pueden declarar como n-arias; de modo que si se ejecuta `declare(j,nary)`, el simplificador transforma `j(j(a,b),j(c,d))` en `j(a, b, c, d)`.

nofix (*op*) [Función]
nofix (*op*, *pos*) [Función]

Los operadores no-fijos se utilizan para definir funciones sin argumentos. La mera presencia de tal operador en una instrucción hará que se evalúe la función correspondiente. Por ejemplo, cuando se teclea `exit;` para salir de una interrupción de Maxima, `exit` se comporta como una función no-fija. La instrucción `nofix("x")` declara `x` como operador no-fijo.

`postfix` (*op*) [Función]
`postfix` (*op*, *lbp*, *lpos*, *pos*) [Función]
Los operadores sufijos son funciones de un único argumento en las que éste precede al operador, como en `3!`. La instrucción `postfix("x")` declara `x` como operador sufijo.

`prefix` (*op*) [Función]
`prefix` (*op*, *rbp*, *rpos*, *pos*) [Función]
Los operadores prefijos son funciones de un único argumento en las que éste se coloca a continuación del operador. La instrucción `prefix("x")` declara `x` como operador prefijo.

7 Evaluación

7.1 Introducción a la evaluación

Las fases que se suceden desde que el usuario solicita un cálculo hasta que obtiene el resultado son: la propia solicitud del cálculo, la evaluación, la simplificación y la respuesta.

Toda expresión *expr* que se introduzca en Maxima será evaluada, de manera que los símbolos que no tengan asociado ningún valor y los números evalúan a sí mismos; en cambio, aquellos símbolos que tengan un valor asociado serán sustituídos por tales valores.

Dos ejemplos. En el primero, los símbolos y los números se evalúan a sí mismos; en el segundo ejemplo, al asignarle a la variable `a` el valor `2`, allá donde se escriba `a` será sustituido por dicho valor.

```
(%i1) [a, b, 2, 1/2, 1.0];
                                  1
(%o1)                  [a, b, 2, -, 1.0]
                                  2
(%i2) a:2$

(%i3) [a, sin(a), a^2];
(%o3)                   [2, sin(2), 4]
```

Maxima distingue entre funciones en forma verbal y funciones en forma nominal. Las funciones en forma verbal son evaluadas tomando en cuenta los valores dados a sus argumentos; en cambio, las funciones nominales no son evaluadas, aunque sus argumentos tengan valores asignados. Las funciones son susceptibles de ser tratadas de ambos modos; ejemplos típicos son la función de diferenciación `diff` y la de integración `integrate`.

En el siguiente ejemplo se le asigna a la variable `a` cierto valor, a continuación se invoca la función `diff` en su forma verbal con sus argumentos tomando los valores `a*x^2` y `x`. Seguidamente se invoca a la misma función `diff` en su forma nominal, lo cual se consigue mediante el operador de comilla simple (`'`); en este caso la función no es evaluada y devuelve una expresión simbólica en la que los argumentos sí han sido evaluados, pues la variable `a` es sustituida por el valor `1/2`.

```
(%i1) a:1/2;
                                  1
(%o1)                             -
                                  2
(%i2) diff(a*x^2, x);
(%o2)                             x
(%i3) 'diff(a*x^2, x);
                                    2
                               d   x
(%o3)                          -- (--)
                               dx  2
```

Sin embargo, no todas las funciones de Maxima sustituyen sus argumentos por sus valores. La documentación para cada función informará si sus argumentos son evaluados o no.

Por ejemplo, la función `properties` no evalúa sus argumentos, lo cual resulta práctico para el usuario, ya que en caso contrario debería utilizar el operador de comilla simple `'` a fin de poder mostrar las propiedades del símbolo `a`. A continuación se muestra como en el primer caso se devuelve una lista vacía, ya que no se le ha encontrado ninguna propiedad al símbolo `a`; una vez se le ha asignado el valor `2`, la función `properties` nos dice que la variable guarda un valor y esto es así porque no ha sustituido el símbolo `a` por su valor `2`. En consecuencia, la función `properties` muestra las propiedades de `'a`.

```
(%i1) properties(a);
(%o1)                          []
(%i2) a:2$

(%i3) properties(a);
(%o3)                       [value]
```

La evaluación de símbolos, funciones y expresiones se puede controlar con los operadores de comilla simple (`'`) y de doble comilla simple (`''`). La evaluación se suprime con la comilla simple y se fuerza con la doble comilla simple (que no comilla doble).

Con la función `ev` se evalúa una expresión dentro de un contexto determinado controlado por el valor de ciertas variables `evflag` y funciones de evaluación `evfun`.

7.2 Funciones y variables para la evaluación

`'` [Operador]

El operador comilla simple `'` evita la evaluación.

Aplicado a un símbolo, la comilla simple evita la evaluación del símbolo.

Aplicado a la llamada de una función, la comilla simple evita la evaluación de la función llamada, aunque los argumentos de la función son evaluados (siempre y cuando la evaluación no se evite de otra manera). El resultado es una forma de nombre de la función llamada.

Aplicado a una expresión con paréntesis, la comilla simple evita la evaluación de todos los símbolos y llamadas a funciones que hayan en la expresión. E.g., `'(f(x))` significa que no se evalua la expresión `f(x)`. `'f(x)` (con la comilla simple aplicada a `f` en cambio de a `f(x)`) significa el retorno de la forma de nombre de `f` aplicada a `[x]`.

La comilla simple no evita la simplificación.

Cuando el interruptor global `noundisp` es `true`, los nombres se muestran con una comilla simple. Este interruptor siempre tiene como valor `true` cuando se muestran definiciones de funciones.

Ver también los operadores comilla-comilla `''` y `nouns`.

Ejemplos:

Aplicado a un símbolo, la comilla simple evita la evaluación del símbolo.

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

```
                                  2
(%o3)                            aa
(%i4) ''%;
(%o4)                          1048576
```

Aplicado a la llamada de una función, la comilla simple evita la evaluación de la función llamada, aunque los argumentos de la función son evaluados (siempre y cuando la evaluación no se evite de otra manera). El resultado es una forma de nombre de la función llamada.

```
(%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 una expresión con paréntesis, la comilla simple evita la evaluación de todos los símbolos y llamadas a funciones que haya dentro en la expresión.

```
(%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
```

La comilla simple no evita la simplificación.

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

Internamente, Maxima considera que las operaciones con números decimales de coma flotante son simples simplificaciones.

```
(%i1) sin(1.0);
(%o1)                          .8414709848078965
(%i2) '(sin(1.0));
(%o2)                          .8414709848078965
```

'' [Operador]

El operador comilla-comilla '' (dos comillas simples) modifica la evaluación en las expresiones de entrada.

Aplicado a cualquier expresión general *expr*, las dos comillas simples hacen que el valor de *expr* sea sustituido por *expr* en la expresión de entrada.

Aplicado al operador de una expresión, el operador comilla-comilla hace que el operador pase de ser un nombre a ser un verbo, a menos que ya sea un verbo.

El operador comilla-comilla es aplicado por el analizador sintáctico de entrada; no se almacena como una parte de la expresión de entrada analizada. Este operador se aplica siempre tan pronto como es detectado y no puede ser comentado con una comilla simple. De esta manera, el operador comilla-comilla provoca la evaluación de una expresión cuando ésta no estaba previsto que fuese evaluada, como en la definición de funciones, expresiones lambda y expresiones comentadas con una comilla simple '.

El operador comilla-comilla es reconocido tanto por `batch` como por `load`.

Véanse también el operador comilla simple ' y `nouns`.

Ejemplos:

Aplicado a cualquier expresión general *expr*, las dos comillas simples hacen que el valor de *expr* sea sustituido por *expr* en la expresión 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 al operador de una expresión, la doble comilla simple hace que el operador pase de ser nominal a verbal, a menos que ya sea un verbo.

```
(%i1) declare (foo, noun);
(%o1)                           done
(%i2) foo (x) := x - 1729;
(%o2)                 ''foo(x) := x - 1729
(%i3) foo (100);
(%o3)                        foo(100)
(%i4) ''foo (100);
(%o4)                          - 1629
```

El operador comilla-comilla es aplicado por el analizador sintáctico de entrada; no se almacena como una parte de la expresión de entrada analizada.

```
(%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
```

El operador comilla-comilla provoca la evaluación de una expresión cuando ésta no estaba previsto que fuese evaluada, como en la definición de funciones, expresiones lambda y expresiones comentadas con una comilla simple '.

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

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

(%o4)                         [%t3, %t4]
(%i5) integrate (log (x), x);
(%o5)                        x log(x) - x
(%i6) foo_2a (x) := ''%;
(%o6)               foo_2a(x) := x log(x) - x
(%i7) foo_2b (x) := %;
(%o7)                    foo_2b(x) := %
(%i8) dispfun (foo_2a, foo_2b);
(%t8)               foo_2a(x) := x log(x) - x

(%t9)                    foo_2b(x) := %

(%o9)                         [%t7, %t8]
(%i10) F : lambda ([u], diff (sin (u), u));
(%o10)             lambda([u], diff(sin(u), u))
(%i11) G : lambda ([u], ''(diff (sin (u), u)));
(%o11)                   lambda([u], cos(u))
(%i12) '(sum (a[k], k, 1, 3) + sum (b[k], k, 1, 3));
(%o12)         sum(b , k, 1, 3) + sum(a , k, 1, 3)
                    k                    k
(%i13) '(''(sum (a[k], k, 1, 3)) + ''(sum (b[k], k, 1, 3)));
(%o13)             b  + a  + b  + a  + b  + a
                    3    3    2    2    1    1
```

ev (*expr*, *arg_1*, ..., *arg_n*) [Función]

Evalua la expresión *expr* en el entorno especificado por los argumentos *arg_1*, ..., *arg_n*. Los argumentos son interruptores (Variables Booleanas), variables de asignación, ecuaciones y funciones. `ev` retorna el resultado (otra expresión) de la evaluación.

La evaluación se realiza por etapas, como sigue:

1. Primero se configura el entorno de acuerdo a los argumentos los cuales pueden ser algunos o todos de la siguiente lista:
   - `simp` causa que *expr* sea simplificada sin importar el valor de la variable interruptor `simp` la cual inhibe la simplificación cuando su valor es `false`.

- `noeval` suprime la fase de evaluación de `ev` (Vea el paso (4) más adelante). Esto es muy útil en conjunción con otras variables interruptor y causan en *expr* que sea resimplificada sin ser reevaluada.
- `nouns` causa la evaluación de las formas nominales (típicamente funciones sin evaluar tales como `'integrate` or `'diff`) en `expr`.
- `expand` causa expansión.
- `expand (m, n)` causa expansión, asignando los valores de `maxposex` y `maxnegex` a *m* y *n*, respectivamente.
- `detout` hace que cualesquiera matrices inversas calculadas en *expr* conserven su determinante fuera de la inversa, en vez de que divida a cada elemento.
- `diff` realiza todas las diferenciaciones indicadas en *expr*.
- `derivlist (x, y, z, ...)` realiza sólo las diferenciaciones con respecto a las variables indicadas.
- `risch` hace que las integrales presentes en *expr* se evalúen mediante el algoritmo de Risch. Véase también `risch`. Cuando se utiliza el símbolo especial `nouns`, se aplica la rutina estándar de integración.
- `float` provoca la conversión de los números racionales no-enteros a números decimales de coma flotante.
- `numer` causa que algunas funciones matemáticas (incluyendo potenciación) con argumentos numéricos sean evaluados como punto flotante. Esto causa que las variables en *expr* las cuales hayan sido declaradas como variables numéricas sean reemplazadas por sus respectivos valores. Esto también configura la variable interruptor `float` a `true`.
- `pred` provoca la evaluación de los predicados (expresiones las cuales se evaluan a `true` o `false`).
- `eval` provoca una post-evaluación extra de *expr* (véase el paso (5) más adelante), pudiendo aparecer `eval` varias veces; por cada aparición de `eval`, la expresión es reevaluada.
- `A`, donde `A` es un átomo declarado como una variable de tipo interruptor, (Vea `evflag`) causa que `A` tenga como valor `true` durante la evaluación de *expr*.
- `V: expresion` (o alternativamente `V=expresion`) causa que `V` tenga el valor de `expresion` durante la evaluación de *expr*. Notese que si `V` es una opción Maxima, entonces `expresion` se usa como su valor durante la evaluación de *expr*. Si más de un argumento de `ev` es de este tipo entonces el vínculo se hace en paralelo. Si `V` es una expresión no atómica entonces se hace una sustitución más que un vínculo.
- `F` donde `F`, un nombre de función, ha sido declarado para ser una función de evaluación (Vea `evfun`) causa que `F` sea aplicada a *expr*.
- Cualquier otro nombre de función (e.g., `sum`) causa la evaluación de las ocurrencias de esos nombres en `expr` como si ellos fueran verbos.
- En adición de que una función ocurra en *expr* (digamos `F(x)`) puede ser definida localmente para el propósito de esta evaluación de *expr* pasando `F(x) := expresion` como un argumento a `ev`.

- Si un átomo no mencionado anteriormente o una variable o expresión con subíndices fueran pasadas como un argumento, esta es evaluada y si el resultado es una ecuación o una asignación entonces el vínculo o sustitución se llevará a cabo. Si el resultado es una lista entonces los miembros de la lista tratados como si ellos fueran argumentos adicionales pasados a `ev`. Esto permite que una lista de argumentos sea pasada (e.g., `[X=1, Y=A**2]`) o una lista de nombres de ecuaciones (e.g., `[%t1, %t2]` donde `%t1` y `%t2` son ecuaciones) tal como lo que es retornado por `solve`.

Los argumentos de `ev` pueden ser pasados en cualquier orden con excepción de la sustitución de ecuaciones las cuales son manipuladas en secuencia, de izquierda a derecha y las funciones de evaluación las cuales son compuestas, e.g., `ev (`*expr*`, ratsimp, realpart)` es manipulada como `realpart (ratsimp (`*expr*`))`.

Los interruptores `simp`, `numer` y `float` pueden también ser configurados localmente en una sentencia block, o globalmente en Maxima para que su efecto permanezca hasta que sean reconfiguradas.

Si *expr* es una Expresión Racional Canónica (CRE, por sus siglas en inglés), entonces la expresión retornada por `ev` es también de tipo CRE, siempre que los interruptores `numer` y `float` no sean `true`.

2. Durante el paso (1), se fabrica una lista de las variables que no contienen subíndices que aparecen en el lado izquierdo de las ecuaciones en los argumentos o en el valor de algunos argumentos si el valor es una ecuación. Las variables (variables que contienen subíndices las cuales no tienen asociado un arreglo de funciones como también las variables que no contienen subíndices) en la expresión *expr* son reemplazadas por sus valores globales, excepto por aquellos que aparezcan en esa lista. Usualmente, *expr* es sólo una etiqueta o un `%` (como en `%i2` en el ejemplo de más abajo) así que este paso simplemente recupera la expresión a la que hace referencia la etiqueta y así `ev` puede trabajarla.
3. Si algunas sustituciones son indicadas por los argumentos, ellas serán llevadas a cabo ahora.
4. La expresión resultante es también reevaluada (a menos que uno de los argumentos fuese `noeval`) y simplificada de acuerdo a los argumentos. Notese que cualquier llamada a una función en *expr* será llevada a cabo después de que las variables sean evaluadas en ella y que `ev(F(x))` pueda comportarse como `F(ev(x))`.
5. Por cada aparición de `eval` en los argumentos, se repetirán los pasos (3) y (4).

Ejemplos:

```
(%i1) sin(x) + cos(y) + (w+1)^2 + 'diff (sin(w), w);
                                    d                     2
(%o1)              cos(y) + sin(x) + -- (sin(w)) + (w + 1)
                                    dw
(%i2) ev (%, numer, expand, diff, x=2, y=1);
                                 2
(%o2)                cos(w) + w  + 2 w + 2.449599732693821
```

Una sintaxis alternativa de alto nivel ha sido desarrollada para `ev` por medio de la cual se pueden escribir solamente sus argumentos, sin el comando `ev()`; se trata de una forma de escritura simplificada:

`expr, arg_1, ..., arg_n`

Esta sintaxis no está permitida dentro de otras expresiones, como funciones, bloques, etc.

Nótese el proceso de vínculo en paralelo en el siguiente ejemplo:

```
(%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]);
Solución

                                         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
```

`eval` [Símbolo especial]

Como argumento en una llamada a `ev (`*`expr`*`)`, `eval` fuerza una evaluación extra de *expr*.

Véase también `ev`.

Ejemplo:

```
(%i1) [a:b,b:c,c:d,d:e];
(%o1)                             [b, c, d, e]
(%i2) a;
(%o2)                                  b
(%i3) ev(a);
(%o3)                                  c
```

```
(%i4) ev(a),eval;
(%o4)                                 e
(%i5) a,eval,eval;
(%o5)                                 e
```

`evflag` [Propiedad]

Cuando un símbolo *x* goza de la propiedad `evflag`, las expresiones `ev(`*`expr`*`,` *`x`*`)` y *`expr`*`,` *`x`* (en modo interactivo) equivalen a `ev(`*`expr`*`,` *`x`* `= true)`. Esto es, a *x* se le asigna `true` al tiempo que se evalúa *expr*.

La expresión `declare(`*`x`*`, evflag)` dota a la variable *x* de la propiedad `evflag`.

Los interruptores que tienen la propiedad `evflag` son:

`algebraic`	`cauchysum`	`demoivre`
`dotscrules`	`%emode`	`%enumer`
`exponentialize`	`exptisolate`	`factorflag`
`float`	`halfangles`	`infeval`
`isolate_wrt_times`	`keepfloat`	`letrat`
`listarith`	`logabs`	`logarc`
`logexpand`	`lognegint`	
`m1pbranch`	`numer_pbranch`	`programmode`
`radexpand`	`ratalgdenom`	`ratfac`
`ratmx`	`ratsimpexpons`	`simp`
`simpproduct`	`simpsum`	`sumexpand`
`trigexpand`		

Ejemplos:

```
(%i1) sin (1/2);
                                 1
(%o1)                       sin(-)
                                 2
(%i2) sin (1/2), float;
(%o2)                  0.479425538604203
(%i3) sin (1/2), float=true;
(%o3)                  0.479425538604203
(%i4) simp : false;
(%o4)                        false
(%i5) 1 + 1;
(%o5)                        1 + 1
(%i6) 1 + 1, simp;
(%o6)                          2
(%i7) simp : true;
(%o7)                        true
(%i8) sum (1/k^2, k, 1, inf);
                             inf
                             ====
                             \     1
(%o8)                         >    --
                             /      2
```

```
                                ====  k
                                k = 1
(%i9) sum (1/k^2, k, 1, inf), simpsum;
                                    2
                                  %pi
(%o9)                             ----
                                   6
(%i10) declare (aa, evflag);
(%o10)                           done
(%i11) if aa = true then YES else NO;
(%o11)                            NO
(%i12) if aa = true then YES else NO, aa;
(%o12)                            YES
```

`evfun` [Propiedad]

Cuando la función F goza de la propiedad `evfun`, las expresiones `ev(`*`expr`*`, `*`F`*`)` y *`expr`*`, `*`F`* (en modo interactivo) equivalen a *`F`*`(ev(`*`expr`*`))`.

Si se especifican dos o más funciones, F, G, etc., como poseedoras de la propiedad `evfun`, éstas se aplican en el mismo orden en el que han sido especificadas como tales.

La expresión `declare(`*`F`*`, evfun)` dota a la función F de la propiedad `evfun`.

Las funciones que tienen la propiedad `evfun` por defecto son:

```
bfloat          factor       fullratsimp
logcontract     polarform    radcan
ratexpand       ratsimp      rectform
rootscontract   trigexpand   trigreduce
```

Ejemplos:

```
(%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` [Variable opcional]

Habilita el modo de "evaluación infinita". `ev` repetidamente evalua una expresión hasta que se pare de hacer cambios. Para prevenir que una variable, digamos `X`, sea evaluada sin parar en este modo, simplemente incluya `X='X` como argumento de `ev`. Esta claro que expresiones como `ev (X, X=X+1, infeval)` generarán un bucle infinito.

`noeval` [Símbolo especial]

El símbolo `noeval` evita la fase de evaluación de `ev`. Es útil conjuntamente con otras variables globales y para poder volver a simplificar expresiones sin tener que evaluarlas otra vez.

`nouns` [Símbolo especial]

El símbolo `nouns` es una `evflag`, lo que significa que cuando se utilice como una opción de la instrucción `ev`, todas las formas nominales que aparezcan en una expresión las convierte en verbales, esto es, las evalúa. Véanse también `noun`, `nounify`, `verb` y `verbify`.

`pred` [Símbolo especial]

Cuando se utiliza como argumento en una llamada a `ev (`*`expr`*`)`, `pred` provoca que los predicados (expresiones que se reducen a `true` o `false`) se evalúen.

Véase `ev`.

Ejemplo:

```
(%i1) 1<2;
(%o1)                                1 < 2
(%i2) 1<2,pred;
(%o2)                                true
```

8 Expresiones

8.1 Introducción a las expresiones

Existe un cierto número de palabras reservadas que no deberían utilizarse como nombres de variables. Su uso podría causar errores sintácticos.

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

La mayoría de los objetos en Maxima son expresiones. Una secuencia de expresiones puede constituir una expresión, separándolas por comas y colocando paréntesis alrededor de ellas. Esto es similar a las *expresiones con coma* en **C**.

```
(%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
```

Incluso los bucles en Maxima son expresiones, aunque el valor que retornan (`done`) no es muy útil.

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

pero quizás se quiera incluir un tercer término en la *expresión con coma* para que devuelva el valor de interés.

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

8.2 Nombres y verbos

Maxima distingue entre operadores que son "nombres" y operadores que son "verbos". Un verbo es un operador que puede ser ejecutado. Un nombre es un operador que aparece como un símbolo en una expresión pero sin ser ejecutado. Por defecto, los nombres de funciones son verbos. Un verbo puede transformarse en nombre utilizando el apóstrofo o aplicando la función `nounify`. Un nombre puede transformarse en verbo aplicando la función `verbify`. La variable `nouns` hace que `ev` evalúe los nombres presentes en una expresión.

La forma verbal se distingue mediante la precedencia del carácter dólar `$` al correspondiente símbolo de Lisp. Por otro lado, la forma nominal se distingue mediante la precedencia del carácter porcentaje `%` al correspondiente símbolo de Lisp. Algunos nombres gozan de propiedades especiales para su representación, como `'integrate` o `'derivative` (devuelto

por `diff`), pero la mayoría no. Por defecto, las formas nominal y verbal de una función son idénticas cuando se muestran en un terminal. La variable global `noundisp` hace que Maxima muestre los nombres precedidos del apóstrofo `'`.

Véanse también `noun`, `nouns`, `nounify` y `verbify`.

Ejemplos:

```
(%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)
```

8.3 Identificadores

En Maxima, los identificadores pueden contener caracteres alfabéticos, números del 0 al 9 y cualquier otro carácter precedido de la barra invertida \.

Un identificador puede comenzar con un carácter numérico si éste va precedido de la barra invertida \. Los caracteres numéricos que ocupen la segunda posición o posterior no necesitan ir precedidos de la barra invertida.

Los caracteres pueden declararse como alfabéticos con la función `declare`. Así declarados, no necesitan ir precedidos de la barra invertida en un identificador. En principio, los caracteres alfabéticos son las letras de `A` a `Z` y `a` a `z`, junto con `%` y `_`.

Maxima distingue minúsculas y mayúsculas. Los identificadores `foo`, `FOO` y `Foo` son distintos. Véase Sección 37.1 [Lisp y Maxima], página 587, para más información.

Un identificador en Maxima es un símbolo Lisp que comienza con el símbolo dólar `$`. Cualquier otro símbolo de Lisp va precedido de la interrogación `?` cuando aparece en Maxima. Véase Sección 37.1 [Lisp y Maxima], página 587, para más información.

Ejemplos:

```
(%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
```

8.4 Desigualdades

Maxima dispone de los operadores de desigualdad `<`, `<=`, `>=`, `>`, `#` y `notequal`. Véase `if` para una descripción de las expresiones condicionales.

8.5 Funciones y variables para expresiones

`alias` (*new_name_1*, *old_name_1*, ..., *new_name_n*, *old_name_n*) [Función]

provee un nombre alternativo para una (bien sea definida por el usuario o por el sistema) función, variable, arreglo, etc. Cualquier número par de argumentos puede ser usado.

`aliases` [Variable del sistema]

Valor por defecto: `[]`

La variable `aliases` es la lista de átomos que tienen un "alias" definido por el usuario (establecido mediante las funciones `alias`, `ordergreat` o `orderless` o declarando el átomo como un `noun` (nombre) con `declare`.

`allbut` [Clave]

Opera con los comandos `part` (como `part`, `inpart`, `substpart`, `substinpart`, `dpart` y `lpart`). Por ejemplo:

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

mientras que:

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

La función `kill` también reconoce a `allbut`.

```
(%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]
```

La sentencia `kill(allbut(a_1, a_2, ...))` tiene el mismo efecto que `kill(all)`, excepto que no elimina los símbolos *a_1*, *a_2*, ...

`args` (*expr*) [Función]

Devuelve la lista de argumentos de `expr`, que puede ser cualquier tipo de expresión a excepción de un átomo. Tan solo se muestran los argumentos del operador principal; subexpresiones de `expr` aparecen como elementos o subexpresiones de elementos de la lista de argumentos.

El orden de los miembros de la lista puede depender de la variable global `inflag`.

La llamada `args (expr)` es equivalente a `substpart ("[", expr, 0)`.

Véanse también `substpart` y `op`.

`atom` (*expr*) [Función]

Devuelve `true` si *expr* es un átomo (número, nombre o cadena alfanumérica) y `false` en caso contario. Así, `atom(5)` devolverá `true`, mientras que `atom(a[1])` y `atom(sin(x))` darán como resultado `false` (dando por hecho que tanto `a[1]` como `x` no tienen valores asignados).

box (*expr*) [Función]
box (*expr*, a) [Función]

Devuelve *expr* encerrada en una caja. El valor devuelto es una expresión con box como operador y *expr* como argumento. Se dibujará una caja cuando display2d valga true.

La llamada box (*expr*, a) encierra *expr* en una caja etiquetada con el símbolo *a*. La etiqueta se recorta si es más larga que el ancho de la caja.

La función box evalúa su argumento. Sin embargo, la expresión encerrada no se evalúa, siendo excluída de los cálculos.

La variable boxchar guarda el carácter a utilizar para dibujar la caja en las funciones box, dpart y lpart.

Ejemplos:

```
(%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 [Variable opcional]

Valor por defecto: "

La variable boxchar guarda el carácter a utilizar para dibujar la caja en las funciones box, dpart y lpart.

Todas las cajas en una expresión se dibujan con el valor actual de `boxchar`, carácter que no se almacena con las expresión encerrada.

`collapse` (*expr*) [Función]

Colapsa *expr* haciendo que todas las subexpresiones que sean iguales compartan las mismas celdas, ahorrando espacio. `collapse` es una subrutina utilizada por la instrucción `optimize`. El uso de `collapse` puede ser útil después de cargar un fichero creado con `save`. Se pueden colapsar varias expresiones de forma conjunta utilizando `collapse ([expr_1, ..., expr_n])`. También se pueden colapsar los elementos del array `A` haciendo `collapse (listarray ('A))`.

`disolate` (*expr*, *x_1*, ..., *x_n*) [Función]

Es similar a `isolate (expr, x)`, excepto que permite al usuario aislar más de una variable simultáneamente. Puede ser útil para hacer un cambio de variables en integrales múltiples en las que tales variables dependan de de dos o más variables de integración. Esta función se carga automáticamente desde `simplification/disol.mac`. Se dispone de una demostyración en `demo("disol")$`.

`dispform` (*expr*) [Función]
`dispform` (*expr*, *all*) [Function]

`dispform(expr)` devuelve la representación externa de *expr* respecto del operador del nivel superior. `dispform(expr, all)` devuelve la representación externa respecto de todos los operadores que haya en *expr*.

Véase también `part`, `inpart` y `inflag`.

Ejemplos:

La representación interna de `- x` es "menos uno multiplicado por `x`", mientras que la representación externa es "menos `x`".

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

La representación interna de `sqrt(x)` es "`x` elevado a 1/2", mientras que su representación externa es "raíz de `x`".

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

```
(%i4) ?format (true, "~S~%", %);
((%SQRT SIMP) $X)
(%o4)                          false
```

Utilización del argumento opcional `all`.

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

`dpart (`*`expr`*`, `*`n_1`*`, ..., `*`n_k`*`)` [Función]

Selecciona la misma expresión que `part`, pero en lugar de devolver esa expresión como su valor, devuelve la expresión completa con la subexpresión seleccionada dentro de una caja. La caja es parte de la expresión.

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

`exptisolate` [Variable opcional]

Valor por defecto: `false`

Véase `isolate`.

`exptsubst` [Variable opcional]

Valor por defecto: `false`

Si `exptsubst` vale `true` permite la sustitución `y` por `%e^x` en `%e^(a x)`.

`freeof (`*`x_1`*`, ..., `*`x_n`*`, `*`expr`*`)` [Función]

`freeof (`*`x_1`*`, `*`expr`*`)` devuelve `true` si ninguna subexpresión de *expr* coincide con *x_1*, o si *x_1* aparece como variable muda en *expr*, o si *x_1* no es ni una forma nominal ni verbal de cualesquiera operadores presentes en *expr*, devolviendo `false` en otro caso.

La llamada `freeof (`*`x_1`*`, ..., `*`x_n`*`, `*`expr`*`)` equivale a `freeof (`*`x_1`*`, `*`expr`*`) and ... and freeof (`*`x_n`*`, `*`expr`*`)`.

Los argumentos *x_1*, ..., *x_n* pueden seer nombres de funciones y variables, nombres subindicados, operadores (encerrados entre comillas dobles) o expresiones generales.

La función `freeof` evalúa sus argumentos.

Una variable es una variable muda en una expresión si no tiene valor asignado fuera de la expresión. Variable mudas reconocidas por `freeof` son el índice de una suma o producto, la variable límite en `limit`, la variable de integración en la versión de integral definida de `integrate`, la variable original en `laplace`, variables formales en expresiones `at` y los argumentos de las expresiones `lambda`.

La versión indefinida de `integrate` no está libre de su variable de integración.

Ejemplos:

Los argumentos son nombres de funciones, variables, nombres subindicados, operaores y expresiones. La llamada `freeof (a, b, expr)` equivale 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` evalúa sus argumentos.

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

`freeof` no considera funciones equivalentes. La simplificación puede dar una expresión equivalente pero 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
```

Un sumatorio o integral definida está libre de su variable muda. Una integral indefinida de `integrate` no está libre de su variable de integración.

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

`inflag` [Variable opcional]

Valor por defecto: `false`

Si `inflag` vale `true`, las funciones para la extracción de partes inspeccionan la forma interna de `expr`.

Nótese que el simplificador reordena expresiones. Así, `first (x + y)` devuelve `x` si `inflag` vale `true` y `y` si `inflag` vale `false`. (`first (y + x)` devuelve el mismo resultado.)

Además, dándole a `inflag` el valor `true` y llamando a `part` o a `substpart` es lo mismo que llamar a `inpart` o a `substinpart`.

Las funciones que se ven afectadas por el valor de `inflag` son: `part`, `substpart`, `first`, `rest`, `last`, `length`, la construcción `for ... in`, `map`, `fullmap`, `maplist`, `reveal` y `pickapart`.

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

Similar a `part`, pero trabaja con la representación interna de la expresión, siendo más rápida. Se debe tener cuidado con el orden de subexpresiones en sumas y productos, pues el orden de las variables en la forma interna es normalmente diferente al que se muestra por el terminal, y cuando se trata con el signo menos unario, resta y división, pues estos operadores desaparecen de la expresión. Las llamadas `part (x+y, 0)` o `inpart (x+y, 0)` devuelven `+`, siendo necesario encerrar el operador entre comillas dobles cuando se haga referencia aél. Por ejemplo, `... if inpart (%o9,0) = "+" then ...`.

Ejemplos:

```
(%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*) [Función]

Devuelve *expr* con subexpresiones que son sumas y que no contienen variables reemplazadas por etiquetas de expresiones intermedias (tales etiquetas son símbolos

atómicos como %t1, %t2, ...). Esta función es de utilidad para evitar la expansión innecesaria de subexpresiones que no contienen la variable de interés. Puesto que las etiquetas intermedias toman el valor de subexpresiones pueden ser todas sustituidas evaluando la expresión en la que aparecen.

Si la variable exptisolate, cuyo valor por defecto es false, vale true hará que isolate busque exponentes de átomos (como %e) que contengan la variable.

Si isolate_wrt_times vale true, entonces isolate también aislará respecto de los productos. Véase isolate_wrt_times.

Para ejemplos, ejecútese example (isolate).

isolate_wrt_times [Variable opcional]

Valor por defecto: false

Si isolate_wrt_times vale true, entonces isolate también aislará respecto de los productos. Compárese el comportamiento de isolate al cambiar el valor de esta variable global en el siguiente ejemplo,

```
(%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 [Variable opcional]

Valor por defecto: false

Si listconstvars vale true, hará que listofvars incluya %e, %pi, %i y cualquier otra variable que sea declarada constante de las que aparezcan en el argumento de listofvars. Estas constantes se omiten por defecto.

listdummyvars [Variable opcional]

Valor por defecto: true

Si listdummyvars vale false, las "variables mudas" de la expresión no serán incluídas en la lista devuelta por listofvars. (La definición de "variables mudas" se encuentra en la descripción de freeof. "Variables mudas" son objetos matemáticos como el

índice de un sumatorio o producto, una variable límite o la variable de una integración definida.) Ejemplo:

```
(%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*) [Función]

Devuelve una lista con las variables presentes en *expr*.

Si la variable `listconstvars` vale `true` entonces `listofvars` incluirá `%e`, `%pi`, `%i` y cualquier otra variable declarada constante de las que aparezcan en *expr*. Estas constantes se omiten por defecto.

Véase también la variable opcional `listdummyvars` para excluir o incluir variables ficticias en la lista de variables.

Ejemplo:

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

`lfreeof` (*list*, *expr*) [Función]

Para cada miembro *m* de *list*, realiza la llamada `freeof (m, expr)`. Devuelve `false` si alguna de estas llamadas a `freeof` retornó `false`, y `true` en caso contrario.

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

Similar a `dpart` pero utiliza una caja etiquetada. Una caja etiquetada es similar a la que produce `dpart`, pero con un nombre en la línea superior.

`mainvar` [Propiedad]

Se pueden declarar variables de tipo `mainvar`. El orden de los átomos es: números `<` constantes (como `%e` o `%pi`) `<` escalares `<` otras variables `<` "mainvars". Por ejemplo, compárese `expand ((X+Y)^4)` con `(declare (x, mainvar), expand ((x+y)^4))`. (Nota: Se debe tener cuidado si se quiere hacer uso de esta declaración. Por ejemplo, si se resta una expresión en la que `x` ha sido declarada como `mainvar` de otra en la que `x` no es `mainvar`, puede ser necesario volver a simplificar, `ev (expr, simp)`, a fin de obtener cancelaciones. Además, si se guarda una expresión en la que `x` es `mainvar`, quizás sea necesario guardar también `x`.)

`noun` [Propiedad]

El símbolo `noun` es una de las opciones de la instrucción `declare`. Hace que una función se declare como "nombre", lo que significa que no se evaluará automáticamente.

`noundisp` [Variable opcional]

Valor por defecto: `false`

Si `noundisp` vale `true`, los nombres se muestran precedidos de un apóstrofo. Siempre debe valer `true` cuando se quiera representar la definición de funciones.

`nounify (`*f*`)` [Función]

Devuelve la forma nominal de la función cuyo nombre es *f*. Puede ser útil cuando se quiera hacer referencia al nombre de una función sin que ésta se ejecute. Nótese que algunas funciones verbales devolverán su forma nominal si no pueden ser evaluadas para ciertos argumentos. Esta es también la expresión que se obtiene cuando la llamada a una función va precedida por del apóstrofo.

`nterms (`*expr*`)` [Función]

Devuelve el número de términos que *expr* llegaría a tener si fuese completamente expandida y no hubiesen cancelaciones ni combinaciones de términos semejantes. Nótese que expresiones como `sin (`*expr*`)`, `sqrt (`*expr*`)`, `exp (`*expr*`)`, etc. cuentan como un sólo término, independientemente de cuántos términos tenga a su vez *expr* en caso de tratarse de una suma.

`op (`*expr*`)` [Función]

Devuelve el operador principal de la expresión *expr*. La llamada `op (`*expr*`)` equivale a `part (`*expr*`, 0)`.

La función `op` devuelve una cadena si el operador principal es un operador prefijo, infijo (binario o `n`-ario), postfijo, "bi-fijo" o "no-fijo" ("bi-fijo" se refiere a un par de símbolos que encierran su o sus argumentos, y "no-fijo" es un operador que no necesita argumentos). Si *expr* es la expresión de una función subindicada, `op` devuelve la función subindicada; en cuyo caso el valor devuelto no es un átomo. En otro caso, *expr* es la expresión de una función array u ordinaria, y entonces `op` devuelve un símbolo.

La función `op` observa el valor de la variable global `inflag`.

La función `op` evalúa sus argumentos.

Véase también `args`.

Ejemplos:

```
(%i1) stringdisp: true$
(%i2) op (a * b * c);
(%o2)                          "*"
(%i3) op (a * b + c);
(%o3)                          "+"
(%i4) op ('sin (a + b));
(%o4)                          sin
(%i5) op (a!);
(%o5)                          "!"
(%i6) op (-a);
(%o6)                          "-"
(%i7) op ([a, b, c]);
(%o7)                          "["
(%i8) op ('(if a > b then c else d));
(%o8)                         "if"
(%i9) op ('foo (a));
(%o9)                          foo
(%i10) prefix (foo);
```

```
(%o10)                          "foo"
(%i11) op (foo a);
(%o11)                          "foo"
(%i12) op (F [x, y] (a, b, c));
(%o12)                          F
                                 x, y
(%i13) op (G [u, v, w]);
(%o13)                           G
```

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

La llamada `operatorp (expr, op)` devuelve `true` si *op* es igual al operador de *expr*.

La llamada `operatorp (expr, [op_1, ..., op_n])` devuelve `true` si algún elemento *op_1*, ..., *op_n* es igual al operador de *expr*.

opsubst [Variable opcional]

Valor por defecto: `true`

Si `opsubst` vale `false`, `subst` no sustituye el operdor de una expresión, de manera que `(opsubst: false, subst (x^2, r, r+r[0]))` trabajará correctamente.

optimize (*expr*) [Función]

Devuelve una expresión que produce el mismo valor y efectos secundarios que *expr*, pero de forma más eficiente al evitar recalcular subexpresiones comunes. La función `optimize` también tiene el efecto secundario de colapsar su argumento de manera que se compartan todas sus subexpresiones comunes. Hágase `example (optimize)` para ver ejemplos.

optimprefix [Variable opcional]

Valor por defecto: `%`

La variable `optimprefix` es el prefijo utilizado para los símbolos generados por la instrucción `optimize`.

ordergreat (*v_1*, ..., *v_n*) [Función]
orderless (*v_1*, ..., *v_n*) [Función]

`ordergreat` cambia el orden canónico de las expresiones de Maxima, de manera que *v_1* prevalece sobre *v_2*, que prevalece sobre ..., que prevalece sobre *v_n*, que prevalece sobre cualquier otro símbolo no presente en la lista de argumentos.

`orderless` cambia el orden canónico de las expresiones de Maxima, de manera que *v_1* precede a *v_2*, que precede a ..., que precede a *v_n*, que precede a cualquier otra variable no presente en la lista de argumentos.

El orden impuesto por `ordergreat` y `orderless` se destruye con `unorder`. `ordergreat` y `orderless` sólo se pueden llamar una vez, a menos que se invoque a `unorder`. La última llamada a `ordergreat` y `orderless` es la que se mantiene activa.

Véase también `ordergreatp`.

ordergreatp (*expr_1*, *expr_2*) [Función]
orderlessp (*expr_1*, *expr_2*) [Función]

`ordergreatp` devuelve `true` si *expr_1* prevalece sobre *expr_2* en el orden canónico de las expresiones de Maxima, o `false` en caso contrario.

`orderlessp` devuelve `true` si *expr_1* precede a *expr_2* en el orden canónico de las expresiones de Maxima, o `false` en caso contrario.

Todos los átomos y expresiones de Maxima son comparables bajo `ordergreatp` y `orderlessp`, aunque existen ejemplos aislados de expresiones para los que estos predicados no son transitivos.

La ordenación canónica de átomos (símbolos, números literales y cadenas) es la siguiente: (enteros y decimales en coma flotante) preceden a (números decimales grandes o *bigfloats*), que preceden a (constantes declaradas), que preceden a (cadenas), que preceden a (escalares declarados), que preceden a (primer argumento de `orderless`), que precede a ..., que precede a (último argumento de `orderless`), que precede a (otros símbolos), que preceden a (último argumento de `ordergreat`), que precede a ..., que precede a (primer argumento de `ordergreat`), que precede a (variables principales declaradas).

Para las expresiones no atómicas, la ordenación canónica se deriva de la ordenación de átomos. Para los operadores nativos `+`, `*` y `^`, los criterios de ordenación no son sencillos de resumir. Para otros operadores nativos, y todas las demás funciones y operadores, las expresiones se ordenan por sus argumentos (empezando por el primero), después por el nombre del operador o función. En caso de expresiones con subíndices, el símbolo subindicado se considera operador y el subíndice un argumento del mismo.

El orden canónico de expresiones se modifica mediante las funciones `ordergreat` y `orderless`, así como por las declaraciones `mainvar`, `constant` y `scalar`.

Véase también `sort`.

Ejemplos:

Ordenación de símbolos comunes y constantes. Nótese que `%pi` no se ordena en función de su valor numérico.

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

Efecto producido por las funciones `ordergreat` y `orderless`.

```
(%i1) sort ([M, H, K, T, E, W, G, A, P, J, S]);
(%o1)           [A, E, G, H, J, K, M, P, S, T, W]
(%i2) ordergreat (S, J);
(%o2)                          done
(%i3) orderless (M, H);
(%o3)                          done
(%i4) sort ([M, H, K, T, E, W, G, A, P, J, S]);
(%o4)           [M, H, A, E, G, K, P, T, W, J, S]
```

Efecto producido por las declaraciones `mainvar`, `constant` y `scalar`.

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

```
(%o3)                          done
(%i4) declare ([A1, B1], scalar);
(%o4)                          done
(%i5) sort ([aa, foo, bar, bb, baz, quux, cc, dd, A1, B1, C1]);
(%o5)   [baz, quux, A1, B1, bar, bb, cc, dd, foo, C1, aa]
```

Ordenación de expresiones no atómicas.

```
(%i1) sort ([1, 2, n, f(1), f(2), f(2, 1), g(1), g(1, 2), g(n), f(n, 1)]);
(%o1) [1, 2, f(1), g(1), g(1, 2), f(2), f(2, 1), n, g(n),
                                                        f(n, 1)]
(%i2) sort ([foo(1), X[1], X[k], foo(k), 1, k]);
(%o2)            [1, foo(1), X , k, foo(k), X ]
                                1              k
```

part (*expr*, *n_1*, ..., *n_k*) [Función]

Devuelve partes de la forma mostrada de `expr`. Obtiene la parte de `expr` que se especifica por los índices *n_1*, ..., *n_k*. Primero se obtiene la parte *n_1* de `expr`, después la parte *n_2* del resultado anterior, y así sucesivamente. El resultado que se obtiene es la parte *n_k* de ... la parte *n_2* de la parte *n_1* de `expr`. Si no se especifican índices, devuelve `expr`.

La función `part` se puede utilizar para obtener un elemento de una lista, una fila de una matriz, etc.

Si el último argumento de la función `part` es una lista de índices, entonces se toman varias subexpresiones, cada una de las cuales correspondiente a un índice de la lista. Así, `part (x + y + z, [1, 3])` devuelve `z+x`.

La variable `piece` guarda la última expresión seleccionada con la función `part`. Se actualiza durante la ejecución de la función, por lo que puede ser referenciada en la misma función.

Si `partswitch` vale `true` entonces de devuelve `end` cuando no exista la parte seleccionada de una expresión, si vale `false` se mostrará un mensaje de error.

Véanse también `inpart`, `substpart`, `substinpart`, `dpart` y `lpart`.

Ejemplos:

```
(%i1) part(z+2*y+a,2);
(%o1)                                2 y
(%i2) part(z+2*y+a,[1,3]);
(%o2)                               z + a
(%i3) part(z+2*y+a,2,1);
(%o3)                                2
```

La instrucción `example (part)` muestra más ejemplos.

partition (*expr*, *x*) [Función]

Devuelve una lista con dos expresiones, que son: (1) los factores de *expr* si es un producto, los términos de *expr* si es una suma, o los elementos de *expr*, si es una lista, que no contengan a *x*, (2) los factores, términos o lista que contengan a *x*.

```
(%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` [Variable opcional]

Valor por defecto: `false`

Si `partswitch` vale `true` entonces de devuelve `end` cuando no exista la parte seleccionada de una expresión, si vale `false` se mostrará un mensaje de error.

`pickapart` (*expr*, *n*) [Función]

Asigna etiquetas de expresiones intermedias a subexpresiones de *expr* al nivel de profundidad *n*, que es un entero. A las subexpresiones a un nivel de profundidad mayor o menor no se les asignan etiquetas. La función `pickapart` devuelve una expresión en términos de expresiones intermedias equivalente a la expresión original *expr*.

Véanse también `part`, `dpart`, `lpart`, `inpart` y `reveal`.

Ejemplos:

```
(%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` [Variable del sistema]

Guarda la última expresión seleccionada por las funciones `part`.

`psubst` (*`list`*, *`expr`*) [Función]
`psubst` (`a`, *`b`*, *`expr`*) [Función]

`psubst(a, b, expr)` es simliar a `subst`. Véase `subst`.

A diferencia de `subst`, la función `psubst` hace sustituciones en paralelo si su primer argumento es una lista de ecuaciones.

Véase también `sublis` para hacer sustituciones en paralelo.

Ejemplo:

El primer ejemplo muestra la sustitución en paralelo con `psubst`. El segundo ejemplo muestra el resultado de la función `subst`, que realiza la sustitución en serie.

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

`rembox` (*`expr`*, *unlabelled*) [Función]
`rembox` (*`expr`*, *`label`*) [Función]
`rembox` (*`expr`*) [Función]

Elimina cajas de *expr*.

La llamada `rembox (expr, unlabelled)` elimina todas las cajas no etiquetadas de *expr*.

La llamada `rembox (expr, label)` sólo elimina las cajas etiquetadas con *label*.

La llamada `rembox (expr)` elimina todas las caajs, independientemente de que estén etiquetadas o no.

Las cajas son dibujadas por las funciones `box`, `dpart` y `lpart`.

Ejemplos:

```
(%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
```

`reveal` (*expr*, *nivel*) [Función]

Reemplaza partes de *expr* al *nivel* especificado y las sutituye por descripciones cortas.

- Las sumas y restas se reemplazan por `Sum(`*n*`)`, siendo *n* el número de términos de la suma.
- Los productos se reemplazan por `Product(`*n*`)`, siendo *n* el número de factores del producto.
- Las potencias se reemplazan por `Expt`.
- Los cocientes se reemplazan por `Quotient`.
- El símbolo negativo se reemplaza por `Negterm`.
- Las listas se reemplazan por `List(`*n*`)`, siendo *n* el número de elementos de la lista.

Si el entero *depth* es mayor o igual que la profundidad máxima de *expr*, `reveal (expr, depth)` devuelve *expr* sin modificar.

La función `reveal` evalúa sus argumentos y devuelve la expresión con las modificaciones solicitadas.

Ejemplo:

```
(%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
```

`sublis (list, expr)` [Función]

Hace sustituciones múltiples en paralelo dentro de las expresiones. *list* es una lista de ecuaciones, cuyos miembros izquierdos deben ser átomos.

La variable `sublis_apply_lambda` controla la simplificación después de `sublis`.

Véase también `psubst` para hacer sustituciones en paralelo.

Ejemplo:

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

`sublis_apply_lambda` [Variable opcional]

Valor por defecto: `true`

Controla si los `lambda` sustituidos son aplicados en la simplificación después de invocar a `sublis`, o si se tiene que hacer un `ev` para hacerlo. Si `sublis_apply_lambda` vale `true`, significa que se ejecute la aplicación.

`subnumsimp` [Variable opcional]

Valor por defecto: `false`

Si vale `true`, las funciones `subst` y `psubst` puede sustituir una variable subindicada `f[x]` por un número simplemente utilizando el símbolo `f`.

Véase también `subst`.

```
(%i1) subst(100,g,g[x]+2);

subst: cannot substitute 100 for operator g in expression g
                                                                    x
 -- an error. To debug this try: debugmode(true);

(%i2) subst(100,g,g[x]+2),subnumsimp:true;
(%o2)                          102
```

`subst (a, b, c)` [Función]

Sustituye *a* por *b* en *c*. El argumento *b* debe ser un átomo o una subexpresión completa de *c*. Por ejemplo, `x+y+z` es una subexpresión completa de `2*(x+y+z)/w` mientras que `x+y` no lo es. Cuando *b* no cumple esta característica, se puede utilizar en algunos casos `substpart` o `ratsubst` (ver más abajo). Alternativamente, si *b* no es de la forma `e/f` entonces se puede usar `subst (a*f, e, c)`, pero si *b* es de la forma `e^(1/f)` se debe usar `subst (a^f, e, c)`. La instrucción `subst` también reconoce `x^y` en `x^-y`, de manera que `subst (a, sqrt(x), 1/sqrt(x))` da `1/a`. Los argumentos *a* y *b* también pueden ser operadores de una expresión acotados por comillas dobles `"` o nombres de funciones. Si se quiere sustituir la variable independiente en expresiones con derivadas se debe utilizar la función `at` (ver más abajo).

La función `subst` es sinónimo de `substitute`.

La llamada `subst (`*eq_1*`, `*expr*`)` o `subst ([`*eq_1*`, ..., `*eq_k*`], `*expr*`)` están permitidas. Las *eq_i* son ecuaciones que indican las sustituciones a realizar. Para cada ecuación, el miembro izquierdo será sustituido por la expresión del miembro derecho en *expr*. Las ecuaciones se sustituyen secuencialmente de izquierda a derecha en *expr*. Véanse las funciones `sublis` y `psubst` para sustituciones en paralelo.

Si la variable `exptsubst` vale `true` se permiten ciertas sustituciones de exponentes; por ejemplo, sustituir `y` por `%e^x` en `%e^(a*x)`.

Si `opsubst` vale `false`, `subst` no intentará sustituir un operador de una expresión. Por ejemplo, `(opsubst: false, subst (x^2, r, r+r[0]))` trabajará sin problemas.

Ejemplos:

```
(%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
```

La sustitución se hace secuencialmente según una lista de ecuaciones. Compárese con la sustitución en paralelo.

```
(%i3) subst([a=b, b=c], a+b);
(%o3)                                   2 c
(%i4) sublis([a=b, b=c], a+b);
(%o4)                                  c + b
```

Para más ejemplos, ejecútese `example (subst)`.

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

Es similar a `substinpart`, pero trabaja con la representación interna de *expr*.

Ejemplos:

```
(%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)
```

Si el último argumento pasado a la función `part` es una lista de índices, se obtendrá la lista de subexpresiones correspondientes a cada uno de los índices.

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

La variable `piece` guarda el valor de la última expresión seleccionada al utilizar las funciones `part`. El valor es asignado durante la ejecución de la función y puede ser utilizada tal como se muestra más abajo. Si a `partswitch` se le asigna el valor `true` entonces se devolverá `end` cuando no existe la parte solicitada; con otro valor devuelve un mensaje de error.

```
(%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
```

Además, dándole a `inflag` el valor `true` y llamando a `part` o `substpart` es lo mismo que invocar a `inpart` o `substinpart`.

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

Sustituye por *x* la subexpresión que se obtiene de aplicar el resto de argumentos a la función `part`, devolviendo el nuevo valor de *expr*. *x* puede ser un operador que sustituya otro operador de *expr*. En ciertos casos, *x* necesita estar entrecomillado por comillas dobles (`"`); por ejemplo, de `substpart ("+", a*b, 0)` se obtiene `b + a`.

Ejemplo:

```
(%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
```

Además, dándole a `inflag` el valor `true` y llamando a `part` o `substpart` es lo mismo que invocar a `inpart` o `substinpart`.

`symbolp` (*expr*) [Función]

Devuelve `true` si *expr* es un símbolo y `false` en caso contrario. La llamada `symbolp(x)` equivale al predicado `atom(x) and not numberp(x)`.

Véase también `Identifiers`.

`unorder` () [Función]

Desactiva las asociaciones creadas por la última utilización de los comandos de ordenación `ordergreat` y `orderless`, los cuales no pueden ser utilizados más de una vez sin invocar a `unorder`.

`unorder` no sustituye en expresiones los símbolos originales por los alias introducidos por `ordergreat` y `orderless`. Es por ello que tras la ejecuciÃ³n de `unorder` los alias aparecen en expresiones anteriores.

Véase también `ordergreat` y `orderless`.

Ejemplos:

`ordergreat(a)` introduce un alias para el símbolo `a`, razón por la cual la diferencia de `%o2` y `%o4` no se anula. `unorder` no restablece el símbolo `a` y el alias aparece en el resultado `%o7`.

```
(%i1)
(%o1)                           []
(%i2) b*x+a^2;
                                    2
(%o2)                         b x + a
(%i3) ordergreat(a);
(%o3)                          done
(%i4) b*x+a^2;
                               2
(%o4)                          a  + b x
(%i5) %th(1)-%th(3);
                                 2    2
(%o5)                           a  - a
(%i6) unorder();
(%o6)                             [a]
(%i7) %th(2);
                                   2    2
(%o7)                         _101a  - a
```

`verbify (`*f*`)` [Función]

Devuelve la forma verbal del nombre de función *f*.

Véanse también `verb`, `noun` y `nounify`.

Ejemplos:

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

9 Simplificación

9.1 Introducción a la simplificación

Tras la evaluación de una expresión se procede a su simplificación. Las funciones matemáticas que involucran cálculos simbólicos y las expresiones con operadores aritméticos no son evaluadas, sino simplificadas, para lo cual Maxima las representa internamente en forma nominal; de ahí que el cálculo numérico de una suma o de una multiplicación no se considera una evaluación, sino una simplificación. La evaluación de una expresión puede inhibirse con el operador de comilla simple (') y su simplificación se puede controlar con el valor asignado a la variable opcional `simp`.

En el siguiente ejemplo, se evita la simplificación con el operador de comilla simple, siendo el resultado una expresión nominal. A continuación, se inhibe la simplificación tras la evaluación de la derivada, dejando sin reducir el resultado a `2*x`.

```
(%i1) 'diff(x*x,x);
                              d    2
(%o1)                         -- (x )
                              dx
(%i2) simp:false;
(%o2)                          false
(%i3) diff(x*x,x);
(%o3)                        1 x + 1 x
```

Para cada función u operador matemático dispone Maxima de una rutina interna que será utilizada para su simplificación siempre que se la encuentre en una expresión. Estas rutinas implementan propiedades simétricas, valores especiales de las funciones y otras propiedades y reglas. La gran cantidad de variables opcionales permiten mantener bajo control la simplificación de funciones y operadores.

Veamos un ejemplo. La simplificación de la función exponencial `exp` se controla con las siguientes variables opcionales: `%enumer`, `%emode`, `%e_to_numlog`, `code`, `logsimp` y `demoivre`. En el primer caso la expresión con la función exponencial no se simplifica, pero en el segundo se reduce a `%i*%pi/2`.

```
(%i1) exp(x+%i*%pi/2), %emode:false;
                                  %i %pi
                              x + ------
                                    2
(%o1)                        %e
(%i2) exp(x+%i*%pi/2), %emode:true;
                                    x
(%o2)                            %i %e
```

Junto con la simplificación aislada de funciones y operadores que Maxima realiza de forma automática, existen también funciones como `expand` o `radcan` que realizan sobre las expresiones simplificaciones especiales. Sigue un ejemplo:

```
(%i1) (log(x+x^2)-log(x))^a/log(1+x)^(a/2);
                             2                a
                      (log(x  + x) - log(x))
```

```
(%o1)                 ----------------------
                                        a/2
                              log(x + 1)
(%i2) radcan(%);
                                        a/2
(%o2)                         log(x + 1)
```

A un operador o función se le pueden asignar propiedades tales como la linealidad, la simetría u otras. Maxima tiene en cuenta estas propiedades durante la simplificación. Por ejemplo, la instrucción `declare(f, oddfun)` declara la función como impar, con lo que Maxima sabrá que las formas `f(-x)` y `-f(x)` son equivalentes, llevando a cabo la reducción oportuna.

Las siguientes propiedades están en la lista `opproperties` y controlan la simplificación de funciones y operadores:

```
additive        lassociative     oddfun
antisymmetric   linear           outative
commutative     multiplicative   rassociative
evenfun         nary             symmetric
```

Tanto las propiedades como los hechos (o hipótesis) establecidos por el usuario dentro de un contexto influyen sobre el proceso de simplificación. Para más detalles véase el capítulo sobre la base de datos de Maxima.

La función seno reduce los múltiplos enteros de `%pi` al valor cero. En este ejemplo se muestra cómo al dotar al símbolo `n` de la propiedad de ser entero, la función se simplifica de la forma apropiada.

```
(%i1) sin(n*%pi);
(%o1)                        sin(%pi n)
(%i2) declare(n, integer);
(%o2)                           done
(%i3) sin(n*%pi);
(%o3)                             0
```

Si las técnicas anteriores no devuelven el resultado esperado por el usuario, éste puede extender a voluntad las reglas que pueda aplicar Maxima; para más información al respecto, véase el capítulo dedicado a las reglas y patrones.

9.2 Funciones y variables para simplificación

`additive` [Propiedad]

Si `declare(f,additive)` ha sido ejecutado, entonces:

(1) Si `f` es univariado, cada vez que el simplificador encuentre `f` aplicada a una suma, `f` será distribuida bajo esta suma. Por ejemplo, `f(x+y)` se simplificará a `f(x)+f(y)`.

(2) Si `f` es una función de 2 o más argumentos, aditivamente es definida como aditiva en el primer argumento de `f`, como en el caso de `sum` o `integrate`. Por ejemplo, `f(h(x)+g(x),x)` se simplificará a `f(h(x),x)+f(g(x),x)`. Esta simplificación no ocurre cuando `f` se aplica a expresiones de la forma `sum(x[i],i,lower-limit,upper-limit)`.

Ejemplo:

```
(%i1) F3 (a + b + c);
```

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

`antisymmetric` [Propiedad]

Si `declare(h,antisymmetric)` es ejecutado, esto dice al simplificador que `h` es antisimétrico. E.g. `h(x,z,y)` será simplificado a `-h(x,y,z)`. Que es, el producto de (-1)^n por el resultado dado por `symmetric` o `commutative`, donde n es el número de intercambios necesarios de dos argumentos para convertirle a esta forma.

Ejemplos:

```
(%i1) S (b, a);
(%o1)                         S(b, a)
(%i2) declare (S, symmetric);
(%o2)                          done
(%i3) S (b, a);
(%o3)                         S(a, b)
(%i4) S (a, c, e, d, b);
(%o4)                   S(a, b, c, d, e)
(%i5) T (b, a);
(%o5)                         T(b, a)
(%i6) declare (T, antisymmetric);
(%o6)                          done
(%i7) T (b, a);
(%o7)                        - T(a, b)
(%i8) T (a, c, e, d, b);
(%o8)                   T(a, b, c, d, e)
```

`combine` (*expr*) [Función]

Simplifica la suma *expr* combinando términos de con igual denominador reduciéndolos a un único término.

`commutative` [Propiedad]

Si `declare(h,commutative)` es ejecutado, le dice al simplificador que `h` es una función conmutaiva. Por ejemplo, `h(x,z,y)` se simplificará a `h(x,y,z)`. Esto es lo mismo que `symmetric`.

`demoivre` (*expr*) [Función]

`demoivre` [Variable opcional]

La función `demoivre (expr)` convierte una expresión sin modificar la variable global `demoivre`.

Cuando `demoivre` vale `true`, los exponenciales complejos se convierten en expresiones equivalentes pero en términos de las funciones trigonométricas: `exp (a + b*%i)` se reduce a `%e^a * (cos(b) + %i*sin(b))` si `b` no contiene a `%i`. Las expresiones `a` y `b` no se expanden.

El valor por defecto de `demoivre` es `false`.

La función `exponentialize` convierte funciones trigonométricas e hiperbólicas a la forma exponencial, por lo que `demoivre` y `exponentialize` no pueden valer `true` al mismo tiempo.

`distrib` (*expr*) [Función]

Distribuye sumas sobre productos. Difiere de `expand` en que trabaja sólo al nivel superior de una expresión, siendo más rápida que `expand`. Difiere de `multthru` en que expande todas las sumas del nivel superior.

Ejemplos:

```
(%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
```

`distribute_over` [Variable opcional]

Valor por defecto: `true`

`distribute_over` controla la distribución de funciones sobre estructuras como listas, matrices y ecuaciones. Actualmente, no todas las funciones de Maxima tienen esta propiedad. Es posible consultar si una función tiene esta propiedad con la instrucción `properties`.

La propiedad distributiva se desactiva asignándole a `distribute_over` el valor `false`.

Ejemplos:

La función `sin` se distribuye sobre una lista:

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

`mod` es una función de dos argumentos que se distribuye sobre listas. La distribución sobre listas anidadas también es posible.

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

Distribución de la función `floor` sobre una matriz y una ecuación.

```
(%i4) floor(matrix([a,b],[c,d]));
                             [ floor(a)  floor(b) ]
(%o4)                        [                    ]
                             [ floor(c)  floor(d) ]
(%i5) floor(a=b);
```

```
(%o5)                          floor(a) = floor(b)
```

Funciones con más de un argumento se distribuyen sobre cualquiera de sus argumentos, o sobre todos ellos.

```
(%i6) expintegral_e([1,2],[x,y]);
(%o6) [[expintegral_e(1, x), expintegral_e(1, y)],
       [expintegral_e(2, x), expintegral_e(2, y)]]
```

Comprueba si una función tiene la propiedad `distribute_over`:

```
(%i7) properties(abs);
(%o7) [integral, distributes over bags, noun, rule, gradef]
```

`domain` [Variable opcional]

Valor por defecto: `real`

Si `domain` vale `complex`, `sqrt (x^2)` permanecerá como `sqrt (x^2)` en lugar de devolver `abs(x)`.

`evenfun` [Propiedad]
`oddfun` [Propiedad]

`declare(f, evenfun` o `declare(f, oddfun)` indican a Maxima que reconozca la función `f` como par o impar, respectivamente.

Ejemplos:

```
(%i1) o (- x) + o (x);
(%o1)                      o(x) + o(- x)
(%i2) declare (o, oddfun);
(%o2)                          done
(%i3) o (- x) + o (x);
(%o3)                            0
(%i4) e (- x) - e (x);
(%o4)                      e(- x) - e(x)
(%i5) declare (e, evenfun);
(%o5)                          done
(%i6) e (- x) - e (x);
(%o6)                            0
```

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

Expande la expresión *expr*. Los productos de sumas y de sumas con exponentes se multiplican, los numeradores de las expresiones racionales que son sumas se separan en sus respectivos términos, y las multiplicaciones (tanto las que son conmutativas como las que no) se distribuyen sobre las sumas en todos los niveles de *expr*.

En el caso de los polinomios es más aconsejable utilizar `ratexpand`, que utiliza un algoritmo más eficiente.

Las variables `maxnegex` y `maxposex` controlan los máximos exponentes negativos y positivos que se van a expandir.

La llamada `expand (`*expr*`, `*p*`, `*n*`)` expande *expr* asignando a `maxposex` el valor *p* y a `maxnegex` el *n*. Esto es útil para expandir sólo parte de la expresión.

La variable `expon` guarda el mayor exponente negativo que será expandido automáticamente, independientemente de `expand`. Por ejemplo, si `expon` vale 4 entonces `(x+1)^(-5)` no se expandirá automáticamente.

La variable `expop` guarda el mayor exponente positivo que será expandido automáticamente. Así, `(x+1)^3` se expandirá automáticamente sólo si `expop` es mayor o igual que 3. Si se quiere expandir `(x+1)^n`, siendo `n` mayor que `expop`, entonces `expand ((x+1)^n)` se desarrollará sólo si `maxposex` no es menor que `n`.

`expand(expr, 0, 0)` provoca que se vuelva a simplificar `expr`. `expr` no se vuelve a evaluar. A diferencia de `ev(expr, noeval)`, se elimina la representación canónica de la expresión. Véase también `ev`.

La variable `expand` utilizada con `ev` provocará una expansión.

El fichero `share/simplification/facexp.mac` contiene algunas funciones relacionadas con `expand` (en concreto, `facsum`, `factorfacsum` y `collectterms`, que se cargan automáticamente) y variables (`nextlayerfactor` y `facsum_combine`) que permiten al usuario estructurar las expresiones controlando la expansión. En `simplification/facexp.usg` se pueden encontrar breves descripciones de estas funciones. Se accederá a una demostración con la instrucción `demo("facexp")`.

Ejemplo:

```
(%i1) expr:(x+1)^2*(y+1)^3;
                                      2       3
(%o1)                          (x + 1)  (y + 1)
(%i2) expand(expr);
       2  3       3    3      2  2       2      2      2
(%o2) x  y  + 2 x y  + y  + 3 x  y  + 6 x y  + 3 y  + 3 x  y
                                                    2
                               + 6 x y + 3 y + x  + 2 x + 1

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

(%i4) expr:(x+1)^-2*(y+1)^3;
                                        3
                                   (y + 1)
(%o4)                              -------
                                        2
                                   (x + 1)
(%i5) expand(expr);
                3             2
               y           3 y           3 y           1
(%o5)     ------------ + ------------ + ------------ + ------------
           2             2             2             2
          x  + 2 x + 1   x  + 2 x + 1   x  + 2 x + 1   x  + 2 x + 1

(%i6) expand(expr,2,2);
```

```
                                                3
                                          (y + 1)
(%o6)                                     ------------
                                           2
                                          x  + 2 x + 1
```

Vuelve a simplificar una expresión pero sin expansión:

```
(%i7) expr:(1+x)^2*sin(x);
                                          2
(%o7)                              (x + 1)  sin(x)
(%i8) exponentialize:true;
(%o8)                                   true
(%i9) expand(expr,0,0);
                                    2    %i x     - %i x
                           %i (x + 1)  (%e     - %e      )
(%o9)                    - ------------------------------
                                          2
```

`expandwrt` (*expr*, *x_1*, ..., *x_n*) [Función]

Expande la expresión `expr` con respecto a las variables *x_1*, ..., *x_n*. Todos los productos que contengan a las variables aparecen explícitamente. El resultado que se obtenga no tendr'a productos de sumas de expresiones que contengan a las variables. Los argumentos *x_1*, ..., *x_n* pueden ser variables, operadores o expresiones.

Por defecto, no se expanden los denominadores, pero esto puede cambiarse mediante el uso de la variable `expandwrt_denom`.

Esta función se carga automáticamente de `simplification/stopex.mac`.

`expandwrt_denom` [Variable opcional]

Valor por defecto: `false`

La variable `expandwrt_denom` controla el tratamiento de las expresiones racinales por parte de `expandwrt`. Si vale `true`, se expandirán tanto el numerador como el denominador de la expresión respecto de los argumentos de `expandwrt`, pero si `expandwrt_denom` vale `false`, sólo se expandirá el numerador.

`expandwrt_factored` (*expr*, *x_1*, ..., *x_n*) [Función]

Es similar a `expandwrt`, pero trata a las expresiones que son productos de una forma algo diferente. La función `expandwrt_factored` expande sólo aquellos factores de `expr` que contienen a las variables *x_1*, ..., *x_n*.

Esta función se carga automáticamente de `simplification/stopex.mac`.

`expon` [Variable opcional]

Valor por defecto: 0

La variable `expon` guarda el mayor exponente negativo que será expandido automáticamente, independientemente de `expand`. Por ejemplo, si `expon` vale 4 entonces `(x+1)^(-5)` no se expandirá automáticamente.

`exponentialize` (*expr*) [Función]
`exponentialize` [Variable opcional]

La función `exponentialize(expr)` convierte las funciones trigonométricas e hiperbólicas de *expr* a exponenciales, sin alterar la variable global `exponentialize`.

Cuando la variable `exponentialize` vale `true`, todas las funciones trigonométricas e hiperbólicas se convierten a forma exponencial. El valor por defecto es `false`.

La función `demoivre` convierte funciones trigonométricas e hiperbólicas a la forma exponencial, por lo que `demoivre` y `exponentialize` no pueden valer `true` al mismo tiempo.

`expop` [Variable opcional]

Valor por defecto: 0

La variable `expop` guarda el mayor exponente positivo que será expandido automáticamente. Así, `(x+1)^3` se expandirá automáticamente sólo si `expop` es mayor o igual que 3. Si se quiere expandir `(x+1)^n`, siendo `n` mayor que `expop`, entonces `expand((x+1)^n)` se desarrollará sólo si `maxposex` no es menor que `n`.

`lassociative` [Propiedad]

La instrucción `declare(g, lassociative)` le indica al simplificador de Maxima que `g` es asociativo por la izquierda. Por ejemplo, `g(g(a, b), g(c, d))`se reduce a `g(g(g(a, b), c), d)`.

`linear` [Propiedad]

Es una de las propiedades de operadores de Maxima. Si la función univariante `f` se declara lineal, la expansión de `f(x + y)` produce `f(x) + f(y)`, `f(a*x)` produce `a*f(x)` si `a` es una constante. Si la función tiene dos o más argumentos, la linealidad se interpreta como la de `sum` o `integrate`, esto es, `f(a*x + b, x)` produce `a*f(x,x) + b*f(1,x)` si `a` y `b` no contienen a `x`.

`linear` equivale a `additive` y `outative`. Véase también `opproperties`.

Ejemplo:

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

`maxnegex` [Variable opcional]

Valor por defecto: 1000

La variable `maxnegex` es el mayor exponente negativo que expandirá la función `expand`. Véase también `maxposex`.

`maxposex` [Variable opcional]

Valor por defecto: 1000

La variable `maxposex` es el mayor exponenteque expandirá la función `expand`. Véase también `maxnegex`.

`multiplicative` [Propiedad]

La instrucción `declare (f, multiplicative)` indica al simplificador de Maxima que `f` is multiplicativa.

1. Si `f` es univariante, cada vez que el simplificador encuentre a `f` aplicad a un producto, `f` se distribuirá sobre ese producto. Por ejemplo, `f(x*y)` se reduciría a `f(x)*f(y)`.
2. Si `f` es una función de 2 o más argumentos, la multiplicabilidad se define como multiplicabilidad para el primer argumento de `f`, de modo que `f (g(x) * h(x), x)` se reduciría a `f (g(x) ,x) * f (h(x), x)`.

Esta transformación no se realiza cuando `f` se aplica a expresiones de la forma `product (x[i], i, m, n)`.

Ejemplo:

```
(%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)
```

`multthru` (*expr*) [Función]

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

Multiplica un factor (que debería ser una suma) de *expr* por los otros factores de *expr*. Esto es, *expr* es *f_1 f_2 ... f_n*, donde al menos un factor, por ejemplo *f_i*, es una suma de términos. Cada término en esta suma se multiplica por los otros factores del producto, excepto el propio *f_i*. La función `multthru` no expande sumas elevadas a exponentes, siendo el método más rápido para distribuir productos (sean o no conmutativos) sobre sumas. Puesto que los cocientes se representan como productos, puede utilizarse`multthru` para dividir sumas entre productos.

La llamada `multthru (expr_1, expr_2)` multiplica cada término de *expr_2* (que debería ser una suma o una ecuación) por *expr_1*. Si *expr_1* no es ella misma una suma, entonces la llamada es 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 that this does not expand (b+a)^10 */
                                          10
                        2   a b   (b + a)
(%o5)                   - + --- + ---------
                        s    2       a b
                             s
(%i6) multthru (a.(b+c.(d+e)+f));
(%o6)             a . f + a . c . (e + d) + a . b
(%i7) expand (a.(b+c.(d+e)+f));
(%o7)          a . f + a . c . e + a . c . d + a . b
```

`nary` [Propiedad]

`declare(f, nary)` le indica a Maxima que reconozca la función `f` como n-aria.

La declaración `nary` no equivale a invocar la función `function_nary, nary`. El único efecto de `declare(f, nary)` es indicar al simplificador de Maxima que aplane expresiones anidadas, como simplificar `foo(x, foo(y, z))` a `foo(x, y, z)`. Véase también `declare`.

Ejemplo:

```
(%i1) H (H (a, b), H (c, H (d, e)));
(%o1)                H(H(a, b), H(c, H(d, e)))
(%i2) declare (H, nary);
(%o2)                           done
(%i3) H (H (a, b), H (c, H (d, e)));
(%o3)                    H(a, b, c, d, e)
```

`negdistrib` [Variable opcional]

Valor por defecto: `true`

Si `negdistrib` vale `true`, -1 se distribuye sobre una expresión. Por ejemplo, `-(x + y)` se transforma en `- y - x`. Dándole el valor `false` se mostrará `- (x + y)` tal cual. Esto puede ser útil, pero también peligroso; al igual que el indicador `simp`, no conviene asignarle el valor `false`.

`opproperties` [Variable del sistema]

La variable `opproperties` es la lista con las propiedades especiales de los operadores reconocidas por el simplificador de Maxima: `linear`, `additive`, `multiplicative`,

`outative`, `evenfun`, `oddfun`, `commutative`, `symmetric`, `antisymmetric`, `nary`, `lassociative`, `rassociative`.

`outative` [Propiedad]

La instrucción `declare (f, outative)` le indica al simplificador de Maxima que los factores constantes del argumento de la función `f` pueden ser extraídos.

1. Si `f` es univariante, cada vez que el simplificador se encuentra con `f` aplicada a un producto, éste será particionado en factores que son constantes y factores que no lo son, siendo entonces los constantes extraídos de la función. Por ejemplo, `f(a*x)` se reducirá a `a*f(x)` siendo `a` una constante. Las constantes no atómicas no serán extraídas.
2. Si `f` es una función de 2 o más argumentos, esta propiedad se define como en `sum` o `integrate`, esto es, `f (a*g(x), x)` se reducirá a `a * f(g(x), x)` si `a` no contiene a `x`.

Las funciones `sum`, `integrate` y `limit` han sido todas declaradas con la propiedad `outative`.

Ejemplo:

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

`radcan` (*expr*) [Función]

Simplifica la expresión *expr*, que puede contener logaritmos, exponenciales y radicales, convirtiéndola a una forma canónica, lo que significa que todas las expresiones funcionalmente equivalentes se reducen a una forma única. Ciertas expresiones, sin embargo, son reducidas por `radcan` a una forma regular, lo que significa que dos expresiones equivalentes no tienen necesariamente el mismo aspecto, pero su diferencia puede ser reducida por `radcan` a cero.

Con algunas expresiones `radcan` puede consunir mucho tiempo. Este es el coste por explorar ciertas relaciones entre las componentes de la expresión para simplificaciones basadas en factorizaciones y expansiones parciales de fracciones de exponentes.

Ejemplos:

```
(%i1) radcan((log(x+x^2)-log(x))^a/log(1+x)^(a/2));
                                              a/2
(%o1)                              log(x + 1)

(%i2) radcan((log(1+2*a^x+a^(2*x))/log(1+a^x)));
(%o2)                                      2
```

```
(%i3) radcan((%e^x-1)/(1+%e^(x/2)));
                                          x/2
(%o3)                                   %e    - 1
```

`radexpand` [Variable opcional]

Valor por defecto: `true`

La variable `radexpand` controla algunas simplificaciones de radicales.

Si `radexpand` vale `all`, las raíces *n*-ésimas de los factores de un producto que sean potencias de *n* se extraen del símbolo radical. Por ejemplo, si `radexpand` vale `all`, `sqrt (16*x^2)` se reduce a `4*x`.

Más concretamente, considérese `sqrt (x^2)`.

- Si `radexpand` vale `all` o se ha ejecutado `assume (x > 0)`, `sqrt(x^2)` se reduce a `x`.
- Si `radexpand` vale `true` y `domain` es `real` (su valor por defecto), `sqrt(x^2)` se reduce a `abs(x)`.
- Si `radexpand` vale `false` o `radexpand` vale `true` y `domain` es `complex`, `sqrt(x^2)` no se simplifica.

Nótese que `domain` sólo se tiene en cuenta si `radexpand` vale `true`.

`rassociative` [Propiedad]

La instrucción `declare (g, rassociative)` le indica al simplificador de Maxima que `g` es asociativa por la derecha. Por ejemplo, `g(g(a, b), g(c, d))` se reduce a `g(a, g(b, g(c, d)))`.

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

Es el "Sequential Comparative Simplification" (método debido a Stoute). La función `scsimp` intenta simplificar *expr* de acuerdo con las reglas *rule_1*, ..., *rule_n*. Si se obtiene una expresión más pequeña, el proceso se repite. En caso contrario, después de que se hayan intentado todas las simplificaciones, devuelve la respuesta original.

La instrucción `example (scsimp)` muestra algunos ejemplos.

`simp` [Variable opcional]

Valor por defecto: `true`

La variable `simp` activa y desactiva la simplificación. La simplificación está activada por defecto. La variable `simp` también es reconocida por la función `ev` como variable de entorno. Véase también `ev`.

Cuando `simp` se utiliza en un entorno `ev` con el valor `false`, la simplificación se evita sólo durante la fase de evaluación de una expresión. La variable no evita la simplificación que sigue a la fase de evaluación.

Ejemplos:

La simplificación se suspende globalmente. La expresión `sin(1.0)` no se simplifica a su valor numérico. La variable de entorno `simp` conmuta el estado de la simplificación.

```
(%i1) simp:false;
(%o1)                                  false
```

```
(%i2) sin(1.0);
(%o2)                                sin(1.0)
(%i3) sin(1.0),simp;
(%o3)                            .8414709848078965
```

La simplificación se vuelve a activar. La variable de entorno `simp` no puede suprimir totalmente la simplificación. El resultado muestra una expresión simplificada, pero la variable `x` guarda como valor una expresión sin simplificar, porque la asignación se realizó durante la fase de evaluación de la expresión.

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

`symmetric` [Propiedad]

La instrucción `declare (h, symmetric)` le indica al simplificador de Maxima que `h` es una función simétrica. Por ejemplo, `h (x, z, y)` se reduce a `h (x, y, z)`.

El nombre `commutative` es sinónimo de `symmetric`.

`xthru` (*expr*) [Función]

Combina todos los términos de *expr* (la cual debe ser una suma) sobre un común denominador sin expandir productos ni sumas elevadas a exponentes al modo que lo hace `ratsimp`. La función `xthru` cancela factores comunes en el numerador y denominador de expresiones racionales, pero sólo si los factores son explícitos.

En ocasiones puede ser útil el uso de `xthru` antes de la llamada a `ratsimp` a fin de cancelar factores explícitos del máximo común divisor del numerador y denominador y así simplificar la expresión a la que se va a aplicar `ratsimp`.

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

10 Funciones matemáticas

10.1 Funciones para los números

`abs` (*z*) [Función]

La función `abs` representa el valor absoluto y se puede aplicar tanto a argumentos numéricos como simbólicos. Si el argumento *z* es un número real o complejo, `abs` devuelve el valor absoluto de *z*. Si es posible, las expresiones simbólicas que utilizan la función del valor absoluto también se simplifican.

Maxima puede derivar, integrar y calcular límites de expresiones que contengan a `abs`. El paquete `abs_integrate` extiende las capacidades de Maxima para calcular integrales que contengan llamadas a `abs`. Véase `(%i12)` en el ejemplo de más abajo.

Cuando se aplica a una lista o matriz, `abs` se distribuye automáticamente sobre sus elementos. De forma similar, también se distribuye sobre los dos miembros de una igualdad. Para cambiar este comportamiento por defecto, véase la variable `distribute_over`.

Ejemplos:

Cálculo del valor absoluto de números reales y complejos, incluyendo constantes numéricas e infinitos. El primer ejemplo muestra cómo `abs` se distribuye sobre los elementos de una lista.

```
(%i1) abs([-4, 0, 1, 1+%i]);
(%o1)                  [4, 0, 1, sqrt(2)]

(%i2) abs((1+%i)*(1-%i));
(%o2)                                2
(%i3) abs(%e+%i);
                                     2
(%o3)                      sqrt(%e  + 1)
(%i4) abs([inf, infinity, minf]);
(%o4)                     [inf, inf, inf]
```

Simplificación de expresiones que contienen `abs`:

```
(%i5) abs(x^2);
                                     2
(%o5)                               x
(%i6) abs(x^3);
                                2
(%o6)                          x  abs(x)

(%i7) abs(abs(x));
(%o7)                          abs(x)
(%i8) abs(conjugate(x));
(%o8)                          abs(x)
```

Integrando y derivando con la función `abs`. Nótese que se pueden calcular más integrales que involucren la función `abs` si se carga el paquete `abs_integrate`. El último ejemplo muestra la transformada de Laplace de `abs`. Véase `laplace`.

```
(%i9) diff(x*abs(x),x),expand;
(%o9)                        2 abs(x)

(%i10) integrate(abs(x),x);
                               x abs(x)
(%o10)                         --------
                                  2

(%i11) integrate(x*abs(x),x);
                              /
                              [
(%o11)                        I x abs(x) dx
                              ]
                              /

(%i12) load("abs_integrate")$
(%i13) integrate(x*abs(x),x);
                        2           3
                       x  abs(x)   x  signum(x)
(%o13)                 --------- - ------------
                           2            6

(%i14) integrate(abs(x),x,-2,%pi);
                                2
                               %pi
(%o14)                         ---- + 2
                                2

(%i15) laplace(abs(x),x,s);
                                 1
(%o15)                           --
                                  2
                                 s
```

`ceiling` (*x*) [Función]

Si *x* es un número real, devuelve el menor entero mayor o igual que *x*.

Si *x* es una expresión constante (por ejemplo, `10 * %pi`), `ceiling` evalúa *x* haciendo uso de números grandes en coma flotante (big floats), aplicando a continuación `ceiling` al número decimal obtenido. Puesto que `ceiling` hace evaluaciones en coma flotante, es posible, pero improbable, que esta función devuelva un valor erróneo para entradas constantes. Para evitar estos errores, la evaluación en punto flotante se lleva a cabo utilizando tres valores para `fpprec`.

Para argumentos no constantes, `ceiling` intenta devolver un valor simplificado. Aquí se presentan algunos ejemplos sobre las simplificaciones que `ceiling` es capaz de hacer:

```
(%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
```

La función `ceiling` no se extiende automáticamente a los elementos de listas y matrices. Por último, para todos los argumentos que tengan una forma compleja, `ceiling` devuelve una forma nominal.

Si el rango de una función es subconjunto de los números enteros, entonces puede ser declarada como `integervalued`. Tanto `ceiling` como `floor` son funciones que hacen uso de esta información; por ejemplo:

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

Ejemplo de uso:

```
(%i1) unitfrac(r) := block([uf : [], q],
    if not(ratnump(r)) then
       error("unitfrac: argument must be a rational number"),
    while r # 0 do (
        uf : cons(q : 1/ceiling(1/r), uf),
        r : r - q),
    reverse(uf))$
(%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
```

`entier` (*x*) [Función]

Devuelve el mayor entero menor o igual a *x*, siendo *x* numérico. La función `fix` (como en `fixnum`) es un sinónimo, de modo que `fix(x)` hace justamente lo mismo.

`floor` (*x*) [Función]

Si *x* es un número real, devuelve el mayor entero menor o igual que *x*.

Si *x* es una expresión constante (por ejemplo, `10 * %pi`), `floor` evalúa *x* haciendo uso de números grandes en coma flotante (big floats), aplicando a continuación `floor` al número decimal obtenido. Puesto que `floor` hace evaluaciones en coma flotante, es posible, pero improbable, que esta función devuelva un valor erróneo para entradas constantes. Para evitar estos errores, la evaluación en punto flotante se lleva a cabo utilizando tres valores para `fpprec`.

Para argumentos no constantes, `floor` intenta devolver un valor simplificado. Aquí se presentan algunos ejemplos sobre las simplificaciones que `floor` es capaz de hacer:

```
(%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
```

La función `floor` no se extiende automáticamente a los elementos de listas y matrices. Por último, para todos los argumentos que tengan una forma compleja, `floor` devuelve una forma nominal.

Si el rango de una función es subconjunto de los números enteros, entonces puede ser declarada como `integervalued`. Tanto `ceiling` como `floor` son funciones que hacen uso de esta información; por ejemplo:

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

`fix (x)` [Función]
Es un sinónimo de `entier (x)`.

`lmax (L)` [Función]
Si L es una lista o conjunto, devuelve `apply ('max, args (L))`. Si L no es una lista o conjunto, envía un mensaje de error.

`lmin (L)` [Función]
Si L es una lista o conjunto, devuelve `apply ('min, args (L))`. Si L no es una lista o conjunto, envía un mensaje de error.

`max (x_1, ..., x_n)` [Función]
Devuelve un valor simplificado de la mayor de las expresiones desde *x_1* hasta *x_n*. Si `get (trylevel, maxmin)` es 2 o más, `max` aplica la simplificación `max (e, -e) --> |e|`. Si `get (trylevel, maxmin)` es 3 o más, *max* intenta eliminar las expresiones que estén entre otros dos de los argumentos dados; por ejemplo, `max (x, 2*x, 3*x) --> max (x, 3*x)`. Para asignar el valor 2 a `trylevel` se puede hacer `put (trylevel, 2, maxmin)`.

`min (x_1, ..., x_n)` [Función]
Devuelve un valor simplificado de la menor de las expresiones desde *x_1* hasta *x_n*. Si `get (trylevel, maxmin)` es 2 o más, `min` aplica la simplificación `min (e, -e) --> |e|`. Si `get (trylevel, maxmin)` es 3 o más, *min* intenta eliminar las expresiones que estén entre otros dos de los argumentos dados; por ejemplo, `min (x, 2*x, 3*x) --> min (x, 3*x)`. Para asignar el valor 2 a `trylevel` se puede hacer `put (trylevel, 2, maxmin)`.

`round (x)` [Función]
Si x es un número real, la función devuelve el entero más próximo a x. Los múltiplos de $1/2$ se redondean al entero par más próximo. La evaluación de x es similar a `floor` y `ceiling`.

`signum (x)` [Función]
Tanto sea x real o complejo, la función `signum` devuelve 0 si x es cero. Para un valor no nulo de x, la función devuelve `x/abs(x)`.

Para valores no numéricos de x, Maxima intenta determinar el signo del argumento. Cuando es negativo, cero o positivo, `signum` devuelve -1, 0 o 1, respectivamente. En caso de no poder determinarse, `signum` devuelve una forma simplificada equivalente. Estas simplificaciones incluyen la transformación de `signum(-x)` en `-signum(x)` y la de `signum(x*y)` en `signum(x) * signum(y)`.

La función `signum` es distributiva respecto de listas, matrices o ecuaciones. Véase `distribute_over`.

10.2 Funciones para los números complejos

`cabs (expr)` [Función]
Calcula el valor absoluto de una expresión que representa a un número complejo. Al contrario que `abs`, la función `cabs` siempre descompone su argumento en sus partes

real e imaginaria. Si `x` e `y` representan variables o expresiones reales, la función `cabs` calcula el valor absoluto de `x + %i*y` como

```
                                  2    2
                              sqrt(y  + x )
```

La función `cabs` puede utilizar propiedades como la simetría de funciones complejas para calcular el valor absoluto de una expresión.

`cabs` no es una función apropiada para cálculos simbólicos; en tales casos, que incluyen la integración, diferenciación y límites que contienen valores absolutos, es mejor utilizar `abs`.

El resultado devuelto por `cabs` puede incluir la función de valor absoluto, `abs`, y el arco tangente, `atan2`.

Cuando se aplica a una lista o matriz, `cabs` automáticamente se distribuye sobre sus elementos. También se distribuye sobre los dos miembros de una igualdad.

Para otras formas de operar con números complejos, véanse las funciones `rectform`, `realpart`, `imagpart`, `carg`, `conjugate` y `polarform`.

Ejemplos:

Ejemplos con `sqrt` and `sin`:

```
(%i1) cabs(sqrt(1+%i*x));
                                 2     1/4
(%o1)                          (x  + 1)
(%i2) cabs(sin(x+%i*y));
                       2        2         2        2
(%o2)       sqrt(cos (x) sinh (y) + sin (x) cosh (y))
```

La simetría especular de la función de error `erf` se utiliza para calcular el valor absoluto del argumento complejo:

```
(%i3) cabs(erf(x+%i*y));
                                                2
            (erf(%i y + x) - erf(%i y - x))
(%o3) sqrt(--------------------------------
                          4
                                                                2
                                  (erf(%i y + x) + erf(%i y - x))
                               - --------------------------------)
                                                4
```

Dado que Maxima reconoce algunas identidades complejas de las funciones de Bessel, le permite calcular su valor absoluto cuando tiene argumentos complejos. Un ejemplo para `bessel_j`:

```
(%i4) cabs(bessel_j(1,%i));
(%o4)                  abs(bessel_j(1, %i))
```

`carg` (*z*) [Función]

Devuelve el argumento complejo de *z*. El argumento complejo es un ángulo `theta` en `(-%pi, %pi]` tal que `r exp (theta %i) =` *z* donde `r` es la magnitud de *z*.

La función `carg` es computacional, no simplificativa.

Véanse también `abs` (módulo complejo), `polarform`, `rectform`, `realpart` y `imagpart`.

Ejemplos:

```
(%i1) carg (1);
(%o1)                           0
(%i2) carg (1 + %i);
                                %pi
(%o2)                           ---
                                 4
(%i3) carg (exp (%i));
(%o3)                            1
(%i4) carg (exp (%pi * %i));
(%o4)                           %pi
(%i5) carg (exp (3/2 * %pi * %i));
                                 %pi
(%o5)                          - ---
                                  2
(%i6) carg (17 * exp (2 * %i));
(%o6)                            2
```

`conjugate` (*x*) [Función]

Devuelve el conjugado complejo 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)
```

`imagpart` (*expr*) [Función]

Devuelve la parte imaginaria de la expresión *expr*.

La función `imagpart` es computacional, no simplificativa.

Véanse también `abs`, `carg`, `polarform`, `rectform` y `realpart`.

`polarform` (*expr*) [Función]

Devuelve una expresión de la forma `r %e^(%i theta)` equivalente a *expr*, con `r` y `theta` son reales.

`realpart` (*expr*) [Función]

Devuelve la parte real de *expr*. La funciones `realpart` y `imagpart` operan también con expresiones que contengan funciones trigonométricas e hiperbólicas, raíces cuadradas, logaritmos y exponentes.

`rectform` (*expr*) [Función]

Devuelve una expresión de la forma `a + b %i` equivalente a *expr*, con *a* y *b* reales.

10.3 Funciones combinatorias

`!!` [Operador]

El operador doble factorial.

Para un número entero, de punto flotante o racional `n`, `n!!` se evaluará como el producto de `n (n-2) (n-4) (n-6) ... (n - 2 (k-1))` donde `k` es igual a `entier(n/2)`, que es, el mayor entero menor o igual a `n/2`. Note que esta definición no coincide con otras defincciones publicadas para argumentos, los cuales no son enteros.

Para un entero par (o impar) `n`, `n!` se evalua el producto de todos los enteros pares (o impares) consecutivos desde 2 (o 1) por `n` inclusive.

Para un argumento `n` el cual no es un número entero, punto flotante o racional, `n!!` produce una forma de nombre `genfact (n, n/2, 2)`.

`binomial` (*x*, *y*) [Función]

Es el coeficiente binomial `x!/(y! (x - y)!)`. Si *x* y *y* son enteros, entonces se calcula el valor numérico del coeficiente binomial. Si *y* o *x* - *y* son enteros, el coeficiente binomial se expresa como un polinomio.

Ejemplos:

```
(%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)
```

`factcomb` (*expr*) [Función]

Trata de combinar los coeficientes de los factoriales de *expr* con los mismos factoriales, convirtiendo, por ejemplo, `(n + 1)*n!` en `(n + 1)!`.

Si la variable `sumsplitfact` vale `false` hará que `minfactorial` se aplique después de `factcomb`.

`factorial (x)` [Función]
`!` [Operador]

Representa la función factorial. Maxima considera `factorial (x)` y `x!` como sinónimos.

Para cualquier número complejo `x`, excepto para enteros negativos, `x!` se define como `gamma(x+1)`.

Para un entero `x`, `x!` se reduce al producto de los enteros desde 1 hasta `x` inclusive. `0!` se reduce a 1. Para un número real o complejo en formato de coma flotante `x`, `x!` se reduce al valor de `gamma(x+1)`. Cuando `x` es igual a `n/2`, siendo `n` un entero impar, entonces `x!` se reduce a un factor racional multiplicado por `sqrt(%pi)` (pues `gamma(1/2)`) es igual a `sqrt(%pi)`).

Las variables opcionales `factlim` y `gammalim` controlan la evaluación numérica de factoriales de argumentos enteros y racionales.

Las funciones `minfactorial` y `factcomb` simplifican expresiones que contiene factoriales.

Véanse también `factlim`, `gammalim`, `minfactorial` y `factcomb`.

Las funciones `gamma`, `bffac` y `cbffac` son variaciones de la función matemática gamma. Las funciones `bffac` y `cbffac` son llamadas internamente desde `gamma` para evaluar la función gamma de números reales y complejos decimales con precisión de reales grandes (bigfloats).

Las funciones `makegamma` substituye a `gamma` para factoriales y funciones relacionadas.

Maxima reconoce la derivada de la función factorial y los límites para ciertos valores específicos, tales como los enteros negativos.

La variable opcional `factorial_expand` controla la simplificación de expresiones como `(n+x)!`, para `n` entero.

Véase también `binomial`.

Ejemplos:

El factorial de un entero se reduce a un número exacto, a menos que el argumento sea mayor que `factlim`. Los factoriales de números reales o complejos se evalúan como decimales de coma flotante.

```
(%i1) factlim:10;
(%o1)                                10
(%i2) [0!, (7/2)!, 8!, 20!];
                              105 sqrt(%pi)
(%o2)                  [1, -------------, 40320, 20!]
                                   16
(%i3) [4.77!, (1.0+%i)!];
(%o3)    [81.44668037931197,
          .3430658398165454 %i + .6529654964201665]
(%i4) [2.86b0!, (1.0b0+%i)!];
(%o4) [5.046635586910012b0,
       3.430658398165454b-1 %i + 6.529654964201667b-1]
```

El factorial de una constante conocida o de una expresión general no se calcula. Pero puede ser posible reducir el factorial después de evaluado el argumento.

```
(%i1) [(%i + 1)!, %pi!, %e!, (cos(1) + sin(1))!];
(%o1)      [(%i + 1)!, %pi!, %e!, (sin(1) + cos(1))!]
(%i2) ev (%, numer, %enumer);
(%o2) [.3430658398165454 %i + .6529654964201665,
       7.188082728976031,
       4.260820476357003, 1.227580202486819]
```

Los factoriales son simplificados o reducidos, no evaluados. Así `x!` puede ser reemplazado en una expresión nominal.

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

Maxima reconoce la derivada de la función factorial.

```
(%i1) diff(x!,x);
(%o1)                           x! psi (x + 1)
                                       0
```

La variable opcional `factorial_expand` controla la simplificación de expresiones con la función factorial.

```
(%i1) (n+1)!/n!,factorial_expand:true;
(%o1)                                 n + 1
```

`factlim` [Variable opcional]

Valor por defecto: -1

La variable `factlim` especifica el mayor factorial que será expandido automáticamente. Si su valor es -1, entonces se expandirán todos los enteros.

`factorial_expand` [Variable opcional]

Valor por defecto: false

La variable `factorial_expand` controla la simplificación de expresiones tales como `(n+1)!`, siendo `n` un entero.

Véase `!` para un ejemplo.

`genfact` (*x*, *y*, *z*) [Función]

Devuelve el factorial generalizado, definido como `x (x-z) (x - 2 z) ... (x - (y - 1) z)`. Así, para el entero *x*, `genfact (x, x, 1) = x!` y `genfact (x, x/2, 2) = x!!`.

`minfactorial` (*expr*) [Función]

Busca en *expr* la presencia de dos factoriales que solo se diferencien en una unidad; en tal caso, `minfactorial` devuelve una expresión simplificada.

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

```
(%o2)                          ---------------
                               (n + 1) (n + 2)
```

`sumsplitfact` [Variable opcional]

Valor por defecto: `true`

Si `sumsplitfact` vale `false`, `minfactorial` se aplica después de `factcomb`.

10.4 Funciones radicales, exponenciales y logarítmicas

`%e_to_numlog` [Variable opcional]

Valor por defecto: `false`

Si `%e_to_numlog` vale `true`, `r` es un número racional y `x` una expresión, `%e^(r*log(x))` se reduce a `x^r` . Téngase en cuenta que la instrucción `radcan` también hace este tipo de transformaciones, así como otras más complicadas. La instrucción `logcontract` "contrae" expresiones que contienen algún `log`.

`%emode` [Variable opcional]

Valor por defecto: `true`

Si `%emode` vale `true`, `%e^(%pi %i x)` se simplifica como sigue.

`%e^(%pi %i x)` se simplifica a `cos (%pi x) + %i sin (%pi x)` si `x` es un número decimal de coma flotante, un entero o un múltiplo de 1/2, 1/3, 1/4 o 1/6, y luego se sigue simplificando.

Para otros valores numéricos de `x`, `%e^(%pi %i x)` se simplifica a `%e^(%pi %i y)` donde `y` es `x - 2 k` para algún entero `k` tal que `abs(y) < 1`.

Si `%emode` vale `false`, no se realizan simplificaciones especiales a `%e^(%pi %i x)`.

`%enumer` [Variable opcional]

Valor por defecto: `false`

Si la variable `%enumer` vale `true` hace que `%e` se reemplace por 2.718... siempre que `numer` valga `true`.

Si `%enumer` vale `false`, esta sustitución se realiza sólo si el exponente en `%e^x` tiene un valor numérico.

Véanse también `ev` y `numer`.

`exp` (*x*) [Función]

Representa la función exponencial. La expresión `exp (x)` en la entrada se simplifica en `%e^x`; `exp` no aparece en expresiones simplificadas.

Si la variable `demoivre` vale `true` hace que `%e^(a + b %i)` se simplifique a `%e^(a (cos(b) + %i sin(b)))` si `b` no contiene a `%i`. Véase `demoivre`.

Si la variable `%emode` vale `true`, hace que `%e^(%pi %i x)` se simplifique. Véase `%emode`.

Si la variable `%enumer` vale `true` hace que `%e` se reemplace por 2.718... siempre que `numer` valga `true`. Véase `%enumer`.

`li` [*s*] (*z*) [Función]

Representa la función polilogarítmica de orden s y argumento z, definida por la serie infinita

$$\mathrm{Li}_s\left(z\right) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}$$

`li [1]` es `- log (1 - z)`. `li [2]` y `li [3]` son las funciones di- y trilogaritmo, respectivamente.

Cuando el orden es 1, el polilogaritmo se simplifica a `- log (1 - z)`, el cual a su vez se reduce a un valor numérico si z es un número real o complejo en coma flotante o si está presente el término `numer`.

Cuando el orden es 2 ó 3, el polilogaritmo se reduce a un valor numérico si z es un número real en coma flotante o si está presente el término `numer`.

Ejemplos:

```
(%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)` [Función]

Representa el logaritmo natural (en base e) de x.

Maxima no tiene definida una función para el logaritmo de base 10 u otras bases. El usuario puede hacer uso de la definición `log10(x) := log(x) / log(10)`.

La simplificación y evaluación de logaritmos se controla con ciertas variables globales:

`logexpand`
: hace que `log(a^b)` se convierta en `b*log(a)`. Si toma el valor `all`, `log(a*b)` también se reducirá a `log(a)+log(b)`. Si toma el valor `super`, entonces `log(a/b)` también se reducirá a `log(a)-log(b)`, siendo `a/b` racional con `a#1`, (la expresión `log(1/b)`, para `b` entero, se simplifica siempre). Si toma el valor `false`, se desactivarán todas estas simplificaciones.

`logsimp`
: si vale `false`, entonces no se transforma `%e` a potencias que contengan logaritmos.

`lognegint`
: si vale `true` se aplica la regla `log(-n) -> log(n)+%i*%pi`, siendo `n` un entero positivo.

`%e_to_numlog`
: si vale `true`, `r` es un número racional y `x` una expresión, `%e^(r*log(x))` se reduce a `x^r`. Téngase en cuenta que la instrucción `radcan` también hace este tipo de transformaciones, así como otras más complicadas. La instrucción `logcontract` "contrae" expresiones que contengan algún `log`.

`logabs` [Variable opcional]

Valor por defecto: `false`

Cuando se calculan integrales indefinidas en las que se generan logaritmos, como en `integrate(1/x,x)`, el resultado se devuelve de la forma `log(abs(...))` si `logabs` vale `true`, o de la forma `log(...)` si `logabs` vale `false`. En la integración definida se hace la asignación `logabs:true`, ya que aquí es normalmente necesario evaluar la integral indefinida en los extremos del intervalo de integración.

`logarc` [Variable opcional]

`logarc (`*expr*`)` [Función]

Si la variable global `logarc` toma el valor `true`, las funciones circulares e hiperbólicas inversas se reemplazan por funciones logarítmicas equivalentes. El valor por defecto de `logarc` es `false`.

La función `logarc(`*expr*`)` realiza la anterior transformación en la expresión *expr* sin necesidad de alterar el valor de la variable global `logarc`.

`logconcoeffp` [Variable opcional]

Valor por defecto: `false`

Controla qué coeficientes se contraen cuando se utiliza `logcontract`. Se le puede asignar el nombre de una función de predicado de un argumento; por ejemplo, si se quiere introducir raíces cuadradas, se puede hacer `logconcoeffp:'logconfun$ logconfun(m):=featurep(m,integer) or ratnump(m)$` . Entonces `logcontract(1/2*log(x));` devolverá `log(sqrt(x))`.

logcontract (*expr*) [Función]

Analiza la expresión *expr* recursivamente, transformando subexpresiones de la forma `a1*log(b1) + a2*log(b2) + c` en `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 )
```

Si se hace `declare(n,integer);` entonces `logcontract(2*a*n*log(x));` da `a*log(x^(2*n))`. Los coeficientes que se contraen de esta manera son aquellos que como el 2 y el `n` satisfacen `featurep(coeff,integer)`. El usuario puede controlar qué coeficientes se contraen asignándole a la variable global `logconcoeffp` el nombre de una función de predicado de un argumento; por ejemplo, si se quiere introducir raíces cuadradas, se puede hacer `logconcoeffp:'logconfun$ logconfun(m):=featurep(m,integer) or ratnump(m)$` . Entonces `logcontract(1/2*log(x));` devolverá `log(sqrt(x))`.

logexpand [Variable opcional]

Valor por defecto: `true`

Si `logexpand` vale `true` hace que `log(a^b)` se convierta en `b*log(a)`. Si toma el valor `all`, `log(a*b)` también se reducirá a `log(a)+log(b)`. Si toma el valor `super`, entonces `log(a/b)` también se reducirá a `log(a)-log(b)`, siendo `a/b` racional con `a#1`, (la expresión `log(1/b)`, para `b` entero, se simplifica siempre). Si toma el valor `false`, se desactivarán todas estas simplificaciones.

lognegint [Variable opcional]

Valor por defecto: `false`

Si `lognegint` vale `true` se aplica la regla `log(-n) -> log(n)+%i*%pi` siendo `n` un entero positivo.

logsimp [Variable opcional]

Valor por defecto: `true`

Si `logsimp` vale `false`, entonces no se transforma `%e` a potencias que contengan logaritmos.

plog (*x*) [Función]

Representa la rama principal del logaritmo natural complejo con `-%pi < carg(x) <= +%pi` .

sqrt (*x*) [Función]

Raíz cuadrada de *x*. Se representa internamente por `x^(1/2)`. Véase también `rootscontract`.

Si la variable `radexpand` vale `true` hará que las raíces `n`-ésimas de los factores de un producto que sean potencias de `n` sean extraídas del radical; por ejemplo, `sqrt(16*x^2)` se convertirá en `4*x` sólo si `radexpand` vale `true`.

10.5 Funciones trigonométricas

10.5.1 Introducción a la trigonometría

Maxima reconoce muchas funciones trigonométricas. No están programadas todas las identidades trigonométricas, pero el usuario puede añadir muchas de ellas haciendo uso de las técnicas basadas en patrones. Las funciones trigonométricas definidas en Maxima son: `acos`, `acosh`, `acot`, `acoth`, `acsc`, `acsch`, `asec`, `asech`, `asin`, `asinh`, `atan`, `atanh`, `cos`, `cosh`, `cot`, `coth`, `csc`, `csch`, `sec`, `sech`, `sin`, `sinh`, `tan` y `tanh`. Hay también un determinado número de instrucciones especiales para manipular funciones trigonométricas; véanse a este respecto `trigexpand`, `trigreduce` y la variable `trigsign`. Dos paquetes adicionales amplían las reglas de simplificación de Maxima, `ntrig` y `atrig1`. Ejecútese `describe(`*`command`*`)` para más detalles.

10.5.2 Funciones y variables para trigonometría

`%piargs` [Variable opcional]

Valor por defecto: `true`

Cuando `%piargs` vale `true`, las funciones trigonométricas se simplifican a constantes algebraicas cuando el argumento es múltiplo entero de π, $\pi/2$, $\pi/3$, $\pi/4$ o $\pi/6$.

Maxima conoce algunas identidades aplicables cuando π, etc., se multiplican por una variable entera (esto es, un símbolo declarado como entero).

Ejemplo:

```
(%i1) %piargs : false$
(%i2) [sin (%pi), sin (%pi/2), sin (%pi/3)];
                                  %pi       %pi
(%o2)             [sin(%pi), sin(---), sin(---)]
                                   2         3
(%i3) [sin (%pi/4), sin (%pi/5), sin (%pi/6)];
                        %pi       %pi       %pi
(%o3)             [sin(---), sin(---), sin(---)]
                         4         5         6
(%i4) %piargs : true$
(%i5) [sin (%pi), sin (%pi/2), sin (%pi/3)];
                                  sqrt(3)
(%o5)                      [0, 1, -------]
                                     2
(%i6) [sin (%pi/4), sin (%pi/5), sin (%pi/6)];
                             1         %pi   1
(%o6)                 [-------, sin(---), -]
                       sqrt(2)       5    2
(%i7) [cos (%pi/3), cos (10*%pi/3), tan (10*%pi/3),
       cos (sqrt(2)*%pi/3)];
                  1    1                sqrt(2) %pi
(%o7)            [-, - -, sqrt(3), cos(-----------)]
                  2    2                     3
```

Se aplican ciertas identidades cuando π o $\pi/2$ se multiplican por una variable entera.

```
(%i1) declare (n, integer, m, even)$
(%i2) [sin (%pi * n), cos (%pi * m), sin (%pi/2 * m),
       cos (%pi/2 * m)];
                                              m/2
(%o2)                  [0, 1, 0, (- 1)   ]
```

`%iargs` [Variable opcional]

Valor por defecto: `true`

Cuando `%iargs` vale `true`, las funciones trigonométricas se simplifican a funciones hiperbólicas si el argumento es aparentemente un múltiplo de la unidad imaginaria i.

La simplificación se lleva a cabo incluso cuando el argumento es manifiestamente real; Maxima sólo se fija en si el argumento es un múltiplo literal de i.

Ejemplos:

```
(%i1) %iargs : false$
(%i2) [sin (%i * x), cos (%i * x), tan (%i * x)];
(%o2)           [sin(%i x), cos(%i x), tan(%i x)]
(%i3) %iargs : true$
(%i4) [sin (%i * x), cos (%i * x), tan (%i * x)];
(%o4)           [%i sinh(x), cosh(x), %i tanh(x)]
```

La simplificación se aplica incluso en el caso de que el argumento se reduzca a un número real.

```
(%i1) declare (x, imaginary)$
(%i2) [featurep (x, imaginary), featurep (x, real)];
(%o2)                     [true, false]
(%i3) sin (%i * x);
(%o3)                      %i sinh(x)
```

`acos` (*x*) [Function]

Arco coseno.

`acosh` (*x*) [Función]

Arco coseno hiperbólico.

`acot` (*x*) [Función]

Arco cotangente.

`acoth` (*x*) [Función]

Arco cotangente hiperbólica.

`acsc` (*x*) [Función]

Arco cosecante.

`acsch` (*x*) [Función]

Arco cosecante hiperbólica.

`asec` (*x*) [Función]

Arco secante.

`asech` (*x*) [Función]
Arco secante hiperbólica.

`asin` (*x*) [Función]
Arco seno.

`asinh` (*x*) [Función]
Arco seno hiperbólico.

`atan` (*x*) [Función]
Arco tangente.

`atan2` (*y*, *x*) [Función]
Calcula el valor de `atan(y/x)` en el intervalo de `-%pi` a `%pi`.

`atanh` (*x*) [Función]
Arco tangente hiperbólica.

`atrig1` [Paquete]
El paquete `atrig1` contiene ciertas reglas de simplificación adicionales para las funciones trigonométricas inversas. Junto con las reglas que ya conoce Maxima, los siguientes ángulos están completamente implementados: `0`, `%pi/6`, `%pi/4`, `%pi/3` y `%pi/2`. Los ángulos correspondientes en los otros tres cuadrantes también están disponibles. Para hacer uso de estas reglas, ejecútese `load("atrig1");`.

`cos` (*x*) [Función]
Coseno.

`cosh` (*x*) [Función]
Coseno hiperbólico.

`cot` (*x*) [Función]
Cotangente.

`coth` (*x*) [Función]
Cotangente hiperbólica.

`csc` (*x*) [Función]
Cosecante.

`csch` (*x*) [Función]
Cosecante hiperbólica.

`halfangles` [Variable opcional]
Valor por defecto: `false`

Si `halfangles` vale `true`, las funciones trigonométricas con argumentos del tipo *expr*`/2` se simplifican a funciones con argumentos *expr*.

Para un argumento real x en el intervalo `0 < x < 2*%pi` el seno del semiángulo se simplifica como

```
                          sqrt(1 - cos(x))
                          ----------------
```

```
                                sqrt(2)
```

Se necesita un factor relativamente complicado para que esta fórmula sea también válida para cualquier argumento complejo z:

```
                realpart(z)
           floor(-----------)
                    2 %pi
      (- 1)                    (1 - unit_step(- imagpart(z))

                          realpart(z)            realpart(z)
                    floor(-----------) - ceiling(-----------)
                             2 %pi                  2 %pi
                ((- 1)                                          + 1))
```

Maxima reconoce este factor y otros similares para las funciones `sin`, `cos`, `sinh` y `cosh`. Para valores especiales del argumento z, estos factores se simplifican de forma apropiada.

Ejemplos:

```
(%i1) halfangles:false;
(%o1)                                false
(%i2) sin(x/2);
                                        x
(%o2)                                sin(-)
                                        2
(%i3) halfangles:true;
(%o3)                                true
(%i4) sin(x/2);
                                                      x
                                                floor(-----)
                                                      2 %pi
                         sqrt(1 - cos(x)) (- 1)
(%o4)                    ---------------------------------
                                     sqrt(2)
(%i5) assume(x>0, x<2*%pi)$
(%i6) sin(x/2);
                                  sqrt(1 - cos(x))
(%o6)                             ----------------
                                      sqrt(2)
```

`ntrig` [Paquete]

El paquete `ntrig` contiene un conjunto de reglas de simplificación que se pueden usar para simplificar funciones trigonométricas cuyos argumentos son de la forma `f(n %pi/10)` donde f es cualquiera de las funciones `sin`, `cos`, `tan`, `csc`, `sec` o `cot`.

`sec` (*x*) [Función]

Secante.

`sech` (*x*) [Función]

Secante hiperbólica.

`sin` (*x*) [Función]
Seno.

`sinh` (*x*) [Función]
Seno hiperbólico.

`tan` (*x*) [Función]
Tangente.

`tanh` (*x*) [Función]
Tangente hiperbólica.

`trigexpand` (*expr*) [Función]
Expande funciones trigonométricas e hiperbólicas de sumas de ángulos y de múltiplos de ángulos presentes en *expr*. Para mejorar los resultados, *expr* debería expandirse. Para facilitar el control por parte del usuario de las simplificaciones, esta función tan solo expande un nivel de cada vez, expandiendo sumas de ángulos o de múltiplos de ángulos. A fin de obtener una expansión completa en senos y coseno, se le dará a la variable `trigexpand` el valor `true`.

La función `trigexpand` está controlada por las siguientes variables:

`trigexpand`
Si vale `true`, provoca la expansión de todas las expresiones que contengan senos y cosenos.

`trigexpandplus`
Controla la regla de la suma para `trigexpand`, la expansión de una suma como `sin(x + y)` se llevará a cabo sólo si `trigexpandplus` vale `true`.

`trigexpandtimes`
Controla la regla del producto para `trigexpand`, la expansión de un producto como `sin(2 x)` se llevará a cabo sólo si `trigexpandtimes` vale `true`.

Ejemplos:

```
(%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` [Variable optativa]
Valor por defecto: `true`

La variable `trigexpandplus` controla la regla de la suma para `trigexpand`. Así, si la instrucción `trigexpand` se utiliza o si la variable `trigexpand` vale `true`, se realizará la expansión de sumas como `sin(x+y)` sólo si `trigexpandplus` vale `true`.

`trigexpandtimes` [Variable optativa]
Valor por defecto: `true`

La variable `trigexpandtimes` controla la regla del producto para `trigexpand`. Así, si la instrucción `trigexpand` se utiliza o si la variable `trigexpand` vale `true`, se realizará la expansión de productos como `sin(2*x)` sólo si `trigexpandtimes` vale `true`.

`triginverses` [Variable optativa]

Valor por defecto: `true`

La variable `triginverses` controla la simplificación de la composición de funciones trigonométricas e hiperbólicas con sus funciones inversas.

Si vale `all`, tanto `atan(tan(x))` como `tan(atan(x))` se reducen a *x*.

Si vale `true`, se desactiva la simplificación de *`arcfun`*(*`fun`*(*x*)).

Si vale `false`, se desactivan las simplificaciones de *`arcfun`*(*`fun`*(*x*)) y *`fun`*(*`arcfun`*(*x*)).

`trigreduce` (*expr*, *x*) [Función]

`trigreduce` (*expr*) [Función]

Combina productos y potencias de senos y cosenos trigonométricos e hiperbólicos de *x*, transformándolos en otros que son múltiplos de *x*. También intenta eliminar estas funciones cuando aparecen en los denominadores. Si no se introduce el argumento *x*, entonces se utilizan todas las variables de *expr*.

Véase también `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
```

Las rutinas de simplificación trigonométrica utilizan información declarada en algunos casos sencillos. Las declaraciones sobre variables se utilizan como se indica a continuación:

```
(%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` [Variable optativa]

Valor por defecto: `true`

Si `trigsign` vale `true`, se permite la simplificación de argumentos negativos en funciones trigonométricas, como en `sin(-x)`, que se transformará en `-sin(x)` sólo si `trigsign` vale `true`.

`trigsimp` (*expr*) [Función]

Utiliza las identidades $\sin(x)^2+\cos(x)^2 = 1$ y $\cosh(x)^2-\sinh(x)^2 = 1$ para simplificar expresiones que contienen `tan`, `sec`, etc., en expresiones con `sin`, `cos`, `sinh`, `cosh`.

Las funciones `trigreduce`, `ratsimp` y `radcan` puden seguir siendo útiles para continuar el proceso de simplificación.

La instrucción `demo ("trgsmp.dem")` muestra algunos ejemplos de `trigsimp`.

`trigrat` (*expr*) [Función]

Devuelve una forma canónica simplificada cuasi-lineal de una expresión trigonométrica; *expr* es una fracción racional que contiene `sin`, `cos` o `tan`, cuyos

argumentos son formas lineales respecto de ciertas variables (o kernels) y `%pi/n` (n entero) con coeficientes enteros. El resultado es una fracción simplificada con el numerador y denominador lineales respecto de `sin` y `cos`. Así, `trigrat` devuelve una expresión lineal siempre que sea posible.

```
(%i1) trigrat(sin(3*a)/sin(a+%pi/3));
(%o1)            sqrt(3) sin(2 a) + cos(2 a) - 1
```

El siguiente ejemplo se ha tomado de Davenport, Siret y Tournier, *Calcul Formel*, Masson (o en inglés, Addison-Wesley), sección 1.5.5, teorema de Morley.

```
(%i1) c : %pi/3 - a - b$
(%i2) bc : sin(a)*sin(3*c)/sin(a+b);
                                          %pi
                 sin(a) sin(3 (- b - a + ---))
                                           3
(%o2)            ----------------------------
                         sin(b + a)
(%i3) ba : bc, c=a, a=c;
                                        %pi
                   sin(3 a) sin(b + a - ---)
                                         3
(%o3)              -------------------------
                                 %pi
                        sin(a - ---)
                                 3
(%i4) ac2 : ba^2 + bc^2 - 2*bc*ba*cos(b);
         2         2         %pi
      sin (3 a) sin (b + a - ---)
                             3
(%o4) --------------------------
                2     %pi
             sin (a - ---)
                      3
                                          %pi
 - (2 sin(a) sin(3 a) sin(3 (- b - a + ---)) cos(b)
                                           3
             %pi            %pi
 sin(b + a - ---))/(sin(a - ---) sin(b + a))
              3              3
     2       2              %pi
  sin (a) sin (3 (- b - a + ---))
                             3
 + ------------------------------
                2
             sin (b + a)
```

```
(%i5) trigrat (ac2);
(%o5) - (sqrt(3) sin(4 b + 4 a) - cos(4 b + 4 a)
 - 2 sqrt(3) sin(4 b + 2 a) + 2 cos(4 b + 2 a)
 - 2 sqrt(3) sin(2 b + 4 a) + 2 cos(2 b + 4 a)
 + 4 sqrt(3) sin(2 b + 2 a) - 8 cos(2 b + 2 a) - 4 cos(2 b - 2 a)
 + sqrt(3) sin(4 b) - cos(4 b) - 2 sqrt(3) sin(2 b) + 10 cos(2 b)
 + sqrt(3) sin(4 a) - cos(4 a) - 2 sqrt(3) sin(2 a) + 10 cos(2 a)
 - 9)/4
```

10.6 Números aleatorios

`make_random_state` (*n*) [Función]
`make_random_state` (*s*) [Función]
`make_random_state` (*true*) [Función]
`make_random_state` (*false*) [Función]

Un objeto de estado aleatorio representa el estado del generador de números aleatorios. El estado consiste en 627 cadenas binarias de 32 bits.

La llamada `make_random_state (n)` devuelve un nuevo objeto de estado aleatorio creado a partir de una semilla entera igual a *n* módulo 2^32. El argumento *n* puede ser negativo.

La llamada `make_random_state (s)` devuelve una copia del estado aleatorio *s*.

La llamada `make_random_state (true)` devuelve un nuevo objeto de estado aleatorio, cuya semilla se genera a partir de la hora actual del reloj del sistema como semilla.

La llamada `make_random_state (false)` devuelve una copia del estado actual del generador de números aleatorios.

`set_random_state` (*s*) [Función]

Establece *s* como estado del generador de números aleatorios.

La función `set_random_state` devuelve `done` en todo caso.

`random` (*x*) [Función]

Devuelve un número seudoaleatorio. Si *x* es un entero, `random (x)` devuelve un entero entre 0 y `x - 1`, ambos inclusive. Si *x* es un decimal en punto flotante, `random (x)` devuelve un decimal no negativo en punto flotante menor que *x*. La función `random` emite un mensaje de error si *x* no es ni entero ni de punto flotante, o si *x* no es positivo.

Las funciones `make_random_state` y `set_random_state` permiten controlar el estado del generador de números aleatorios.

El generador de números aleatorios de Maxima implementa el algoritmo de Mersenne twister MT 19937.

Ejemplos:

```
(%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
```

11 Base de datos de Maxima

11.1 Introducción a la base de datos de Maxima

Propiedades

A las variables y funciones se les puede asignar propiedades con la función `declare`. Estas propiedades son almacenadas en un *banco de datos* o registradas en una *lista de propiedades* que proporciona Lisp. Con la función `featurep` se puede comprobar si un símbolo tiene una determinada propiedad y con la función `properties` se pueden obtener todas las propiedades asociadas a él. A su vez, la función `remove` elimina una propiedad de la base de datos o de la lista de propiedades. En caso de utilizar `kill` para borrar el valor asignado a una variable, también serán borradas todas las propiedades asociadas a la misma.

El usuario tiene la facultad de añadirle propiedades a un símbolo con las funciones `put` y `qput`. Con la función `get` podrá leer sus propiedades y borrarlas con `rem`.

Las variables pueden tener las siguientes propiedades a almacenar en el banco de datos:

```
constant
integer        noninteger
even           odd
rational       irrational
real           imaginary      complex
```

Las funciones pueden tener las siguientes propiedades a almacenar en el banco de datos:

```
increasing     decreasing
posfun         integervalued
```

Las siguientes propiedades se pueden aplicar a funciones y se utilizan para su correcta simplificación. Estas propiedades se describen en el capítulo dedicado a la simplificación:

```
linear         additive       multiplicative
outative       commutative    symmetric
antisymmetric  nary           lassociativ
rassociative   evenfun        oddfun
```

Otras propiedades aplicables a variables y funciones, y que se almacenan en la lista de propiedades de Lisp, son:

```
bindtest       feature        alphabetic
scalar         nonscalar      nonarray
```

Contextos

Maxima administra contextos en los que se almacenan tanto las propiedades de las variables y funciones como hechos o hipótesis sobre las mismas. Los hechos se establecen con la función `assume` y se almacenan en el contexto actual. Por ejemplo, con `assume(a>10)` guarda Maxima la información sobre el hecho de que la variable `a` es mayor que `10`. Con la función `forget` se borran los hechos de la base de datos. Cuando Maxima pregunta al usuario sobre las propiedades de una variable, éstas son almacenadas en un contexto.

Cada contexto se identifica por un nombre. Al iniciarse Maxima, el contexto actual recibe el nombre de `initial` y se puede definir un número arbitrario de contextos adicionales que

pueden organizarse de forma jerárquica. Así, el contexto `initial` está incluido en el contexto `global`. Los hechos definidos en un contexto dado están siempre activos en los contextos de nivel inferior. Por ejemplo, el contexto `global` contiene hechos que se inicializan por el propio Maxima y estarán activos, por tanto, en el contexto `initial`.

Los contextos pueden almacenar un número arbitrario de hechos y pueden desactivarse con la función `deactivate`. Desactivar un contexto no implica la pérdida de los hechos almacenados, pudiendo ser posteriormente reactivado con la función `activate`, estando los hechos siempre a disposición del usuario.

11.2 Funciones y variables para las propiedades

`alphabetic` [Propiedad]

`alphabetic` es un tipo de propiedad reconocida por `declare`. La expresión `declare(`*s*`, alphabetic)` le indica a Maxima que reconozca como alfabéticos todos los caracteres que haya en *s*, que debe ser una cadena de texto.

Véase también `Identifiers`.

Ejemplo:

```
(%i1) xx\~yy\‘\@ : 1729;
(%o1)                          1729
(%i2) declare ("~‘@", alphabetic);
(%o2)                          done
(%i3) xx~yy‘@ + @yy‘xx + ‘xx@@yy~;
(%o3)               ‘xx@@yy~ + @yy‘xx + 1729
(%i4) listofvars (%);
(%o4)                  [@yy‘xx, ‘xx@@yy~]
```

`bindtest` [Propiedad]

La sentencia `declare(`*x*`, bindtest` le indica a Maxima que devuelva un mensaje de error cuando el símbolo *x* no tenga asociado valor alguno.

Ejemplo:

```
(%i1) aa + bb;
(%o1)                         bb + aa
(%i2) declare (aa, bindtest);
(%o2)                          done
(%i3) aa + bb;
aa unbound variable
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i4) aa : 1234;
(%o4)                          1234
(%i5) aa + bb;
(%o5)                        bb + 1234
```

`constant` [Propiedad]

`declare(a, constant)` declara *a* como constante. La declaración de un símbolo como constante no impide que se le asigne un valor no constante al símbolo.

Véanse `constantp` y `declare`

Ejemplo:

```
(%i1) declare(c, constant);
(%o1)                         done
(%i2) constantp(c);
(%o2)                         true
(%i3) c : x;
(%o3)                           x
(%i4) constantp(c);
(%o4)                         false
```

`constantp` (*expr*) [Función]

Devuelve `true` si *expr* es una expresión constante y `false` en caso contrario.

Una expresión se considera constante si sus argumentos son números (incluidos los números racionales que se muestran con `/R/`), constantes simbólicas como `%pi`, `%e` o `%i`, variables con valor constante o declarada como constante por `declare`, o funciones cuyos argumentos son constantes.

La función `constantp` evalúa sus argumentos.

Ejemplos:

```
(%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*, *f_1*, *a_2*, *f_2*, ...) [Función]

Asigna al átomo o lista de átomos *a_i* la propiedad o lista de propiedades*p_i*. Si *a_i* y/o *p_i* son listas, cada uno de los átomos adquiere todas las propiedades.

La función `declare` no evalúa sus argumentos y siempre devuelve la expresión `done`.

La llamada `featurep (object, feature)` devuelve `true` si *object* ha sido previamente declarado como poseedor de la propiedad *feature*.

Véase también `features`.

La función `declare` reconoce las siguientes propiedades:

`additive` Hace que Maxima simplifique las expresiones *a_i* haciendo uso de la sustitución `a_i(x + y + z + ...) --> a_i(x) + a_i(y) + a_i(z) + ...`. Tal sustitución se aplica únicamente al primer argumento.

`alphabetic`
Indica a Maxima que reconozca todos los caracteres de la cadena alfanumérica *a_i* como caracteres alfabéticos.

`antisymmetric, commutative, symmetric`
Hace que Maxima reconozca a *a_i* como una función simétrica o antisimétrica. La propiedad `commutative` equivale a `symmetric`.

`bindtest` Hace que Maxima envíe un error si *a_i* es evaluado sin habérsele asignado un valor.

`constant` Hace que Maxima considere a *a_i* como una constante simbólica.

`even, odd` Hace que Maxima reconozca a *a_i* como una variable entera par o impar.

`evenfun, oddfun`
Hace que Maxima reconozca a *a_i* como una función par o impar.

`evflag` Hace que *a_i* sea reconocida por `ev`, de manera que a *a_i* se le asigne el valor `true` durante la ejecución de `ev` cuando *a_i* aparezca como argumento de control de `ev`. Véase también `evflag`.

`evfun` Hace que *a_i* sea reconocida por `ev`, de manera que la función nombrada por *a_i* se aplique cuando *a_i* aparezca como argumento de control de `ev`. Véase también `evfun`.

`feature` Hace que Maxima considere a *a_i* como el nombre de una propiedad. Otros átomos podrán ser declarados entonces como poseedores de la propiedad *a_i*.

`increasing, decreasing`
Hace que Maxima reconozca a *a_i* como una función creciente o decreciente.

`integer, noninteger`
Hace que Maxima reconozca a *a_i* como una variable entera o no entera.

`integervalued`
Hace que Maxima reconozca a *a_i* como una función de valor entero.

`lassociative, rassociative`
Hace que Maxima reconozca a *a_i* como una función asociativa por la derecha o por la izquierda.

`linear` Equivale a declarar *a_i* conjuntamente como `outative` y `additive`.

`mainvar` Hace que Maxima considere a *a_i* como una "variable principal", dándole prioridad frente a cualesquiera otras constantes o variables en la ordenación canónica de expresiones de Maxima, tal como determina `ordergreatp`.

`multiplicative`
Hace que Maxima simplifique las expresiones *a_i* haciendo uso de la sustitución `a_i(x * y * z * ...) --> a_i(x) * a_i(y) * a_i(z) * ...`. Tal sustitución se aplica únicamente al primer argumento.

`nary` Hace que Maxima reconozca a *a_i* como una función n-aria.

La declaración `nary` no es equivalente a la función `nary`. El único efecto de `declare(foo, nary)` consiste en hacer que el simplificador de Maxima reduzca expresiones anidadas; por ejemplo, para transformar `foo(x, foo(y, z))` a `foo(x, y, z)`.

`nonarray` Indica que Maxima no debe considerar *a_i* como un array. Esta declaración evita la evaluación múltiple de variables subindicadas.

`nonscalar`
Hace que Maxima considere a *a_i* como una variable no escalar. Se aplica comúnmente para declarar una variable como un vector simbólico o una matriz simbólica.

`noun` Hace que Maxima considere a *a_i* como un nombre. El efecto que se obtiene es que se reemplazan todas las expresiones *a_i* por `'a_i` o `nounify (a_i)`, dependiendo del contexto.

`outative` Hace que Maxima simplifique las expresiones *a_i* extrayendo los factores constantes del primer argumento.

Cuando *a_i* tenga un único argumento, un factor se considerará constante si es una constante literal o declarada.

Cuando *a_i* tenga dos o más argumentos, un factor se considerará constante si el segundo argumento es un símbolo y el factor no contiene al segundo argumento.

`posfun` Hace que Maxima reconozca a *a_i* como una función positiva.

`rational, irrational`
Hace que Maxima reconozca a *a_i* como una variable real racional o irracional.

`real, imaginary, complex`
Hace que Maxima reconozca a *a_i* como una variable real, imaginaria o compleja.

`scalar` Hace que Maxima considere a *a_i* como una variable escalar.

Ejemplos sobre el uso de estas propiedades están disponibles en la documentación correspondiente a cada propiedad por separado.

`decreasing` [Propiedad]
`increasing` [Propiedad]

Las instrucciones `declare(f, decreasing)` y `declare(f, increasing` le indican a Maxima que reconozca la función *f* como una función decreciente o creciente.

Véase también `declare` para más propiedades.

Ejemplo:

```
(%i1) assume(a > b);
(%o1)                         [a > b]
(%i2) is(f(a) > f(b));
(%o2)                         unknown
```

```
(%i3) declare(f, increasing);
(%o3)                         done
(%i4) is(f(a) > f(b));
(%o4)                         true
```

even [Propiedad]
odd [Propiedad]

`declare(a, even)` y `declare(a, odd)` le indican a Maxima que reconozca el símbolo *a* como entero par o impar. Las propiedades `even` y `odd` no son reconocidas por las funciones `evenp`, `oddp` y `integerp`.

Véanse también `declare` y `askinteger`.

Ejemplo:

```
(%i1) declare(n, even);
(%o1)                         done
(%i2) askinteger(n, even);
(%o2)                          yes
(%i3) askinteger(n);
(%o3)                          yes
(%i4) evenp(n);
(%o4)                         false
```

`feature` [Propiedad]

Maxima interpreta dos tipos diferentes de propiedades, del sistema y las que se aplican a expresiones matemáticas. Véase `status` para obtener información sobre propiedades del sistema, así como `features` y `featurep` para propiedades de las expresiones matemáticas.

`feature` no es el nombre de ninguna función o variable.

`featurep` (*a*, *f*) [Función]

Intenta determinar si el objeto *a* tiene la propiedad *f* en base a los hechos almacenados en la base de datos. En caso afirmativo, devuelve `true`, o `false` en caso contrario.

Nótese que `featurep` devuelve `false` cuando no se puedan verificar ni *f* ni su negación.

`featurep` evalúa su argumento.

Véanse también `declare` y `features`.

Ejemplos:

```
(%i1) declare (j, even)$
(%i2) featurep (j, integer);
(%o2)                          true
```

`features` [Declaración]

Maxima reconoce ciertas propiedades matemáticas sobre funciones y variables.

La llamada `declare (x, foo)` asocia la propiedad *foo* a la función o variable *x*.

La llamada `declare (foo, feature)` declara una nueva propiedad *foo*. Por ejemplo, `declare ([rojo, verde, azul], feature)` declara tres nuevas propiedades, `rojo`, `verde` y `azul`.

El predicado `featurep (x, foo)` devuelve `true` si *x* goza de la propiedad *foo*, y `false` en caso contrario.

La lista `features` contiene las propiedades que reconoce Maxima; a saber,

```
integer        noninteger      even
odd            rational        irrational
real           imaginary       complex
analytic       increasing      decreasing
oddfun         evenfun         posfun
commutative    lassociative    rassociative
symmetric      antisymmetric
```

junto con las definidas por el usuario.

La lista `features` sólo contiene propiedades matemáticas. Hay otra lista con propiedades no matemáticas; Véase `status`.

Ejemplo:

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

`get (a, i)` [Función]

Recupera la propiedad de usuario indicada por *i* asociada al átomo *a* o devuelve `false` si *a* no tiene la propiedad *i*.

La función `get` evalúa sus argumentos.

Véanse también `put` y `qput`.

```
(%i1) put (%e, 'transcendental, 'type);
(%o1)                    transcendental
(%i2) put (%pi, 'transcendental, 'type)$
(%i3) put (%i, 'algebraic, 'type)$
(%i4) typeof (expr) := block ([q],
        if numberp (expr)
        then return ('algebraic),
        if not atom (expr)
        then return (maplist ('typeof, expr)),
        q: get (expr, 'type),
        if q=false
        then errcatch (error(expr,"is not numeric.")) else q)$
(%i5) typeof (2*%e + x*%pi);
x is not numeric.
(%o5)  [[transcendental, []], [algebraic, transcendental]]
(%i6) typeof (2*%e + %pi);
(%o6)     [transcendental, [algebraic, transcendental]]
```

integer [Propiedad]
noninteger [Propiedad]

`declare(a, integer)` o `declare(a, noninteger)` indica a Maxima que reconozca *a* como una variable entera o no entera.

Véase también `declare`.

Ejemplo:

```
(%i1) declare(n, integer, x, noninteger);
(%o1)                          done
(%i2) askinteger(n);
(%o2)                           yes
(%i3) askinteger(x);
(%o3)                           no
```

integervalued [Propiedad]

`declare(`*f*`, integervalued)` indica a MAxima que reconozca *f* como una función que toma valores enteros.

Véase también `declare`.

Ejemplo:

```
(%i1) exp(%i)^f(x);
                                 %i f(x)
(%o1)                          (%e       )
(%i2) declare(f, integervalued);
(%o2)                          done
(%i3) exp(%i)^f(x);
                                 %i f(x)
(%o3)                           %e
```

nonarray [Propiedad]

La instrucción `declare(a, nonarray)` le indica a Maxima que no considere *a* como un array. Esta declaración evita la evaluación múltiple de *a*, si ésta es una variable subindicada.

Véase también `declare`.

Ejemplo:

```
(%i1) a:'b$ b:'c$ c:'d$

(%i4) a[x];
(%o4)                           d
                                 x
(%i5) declare(a, nonarray);
(%o5)                          done
(%i6) a[x];
(%o6)                           a
                                 x
```

nonscalar [Propiedad]

Hace que los átomos se comporten como hace una lista o matriz con respecto del operador `.` del la multiplicación no conmutativa.

Véase también `declare`.

`nonscalarp` (*expr*) [Función]
Devuelve `true` si *expr* no es escalar, es decir, si contiene átomos declarados como no escalares, listas o matrices.

`posfun` [Declaración]
La instrucción `declare (f, posfun)` declara a `f` como función positiva, de forma que `is (f(x) > 0)` devolverá `true`.

Véase también `declare`.

`printprops` (`a`, *i*) [Función]
`printprops` ([`a_1`, ..., `a_n`], *i*) [Función]
`printprops` (*all*, *i*) [Función]
Muestra la propiedad con el indicador *i* asociado con el átomo *a*. *a* puede ser también una lista de átomos o el átomo `all` en cuyo caso todos los átomos a los cuales se les haya dado esa propiedad serán usados. Por ejemplo, `printprops ([f, g], atvalue)`. `printprops` es para propiedades que no pueden ser mostradas de otra manera, i.e. para `atvalue`, `atomgrad`, `gradef`, y `matchdeclare`.

`properties` (`a`) [Función]
Devuelve una lista con los nombres de todas las propiedades asociadas al átomo *a*.

`props` [Variable del sistema]
Valor por defecto: `[]`

`props` son átomos que tienen asociadas otras propiedades además de las explícitamente mencionadas en `infolists`, tales como las especificadas por `atvalue`, `matchdeclare` y la función `declare`.

`propvars` (*prop*) [Función]
Devuelve la lista con los átomos de la lista `props` que tienen la propiedad indicada por *prop*. Así, `propvars (atvalue)` devuelve la lista con los átomos a los que se les ha asociado valores con `atvalue`.

`put` (*átomo*, *valor*, *indicador*) [Función]
Asigna el *valor* a la propiedad (especificada por *indicador*) de *átomo*; *indicador* puede ser el nombre de cualquier propiedad y no solamente de aquellas definidas por el sistema.

`rem` deshace la asignación realizada por `put`.

La función `put` evalúa sus argumentos y devuelve *valor*.

Ejemplos:

```
(%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*) [Función]

Asigna *valor* a la propiedad de *átomo* que especifique *indicador*. Actúa del mismo modeo que `put`, excepto que sus argumentos no son evaluados.

Véase también `get`.

Ejemplo:

```
(%i1) foo: aa$
(%i2) bar: bb$
(%i3) baz: cc$
(%i4) put (foo, bar, baz);
(%o4)                              bb
(%i5) properties (aa);
(%o5)                  [[user properties, cc]]
(%i6) get (aa, cc);
(%o6)                              bb
(%i7) qput (foo, bar, baz);
(%o7)                              bar
(%i8) properties (foo);
(%o8)             [value, [user properties, baz]]
(%i9) get ('foo, 'baz);
(%o9)                              bar
```

`rational` [Propiedad]

`irrational` [Propiedad]

`declare(a, rational)` o `declare(a, irrational)` indica a Maxima que reconozca *a* como una variable real racional o irracional.

Véase también `declare`.

`real` [Propiedad]

`imaginary` [Propiedad]

`complex` [Propiedad]

`declare(a, real)`, `declare(a, imaginary)` o `declare(a, complex)` indican a Maxima que reconozca *a* como variable real, imaginaria puro o compleja, respectivamente.

Véase también `declare`.

`rem` (*átomo*, *indicador*) [Función]

Elimina del *átomo* la propiedad indicada por *indicador*. `rem` deshace la asignación realizada por `put`.

`rem` devuelve `done` si *átomo* tenía la propiedad *indicador* cuando `rem` fue invocado, devolviendo `false` si carecía tal propiedad.

`remove` (`a_1`, `p_1`, ..., `a_n`, `p_n`) [Función]
`remove` ([`a_1`, ..., `a_m`], [`p_1`, ..., `p_n`], ...) [Función]
`remove` (`"a"`, *operator*) [Función]
`remove` (`a`, *transfun*) [Función]
`remove` (*all*, `p`) [Función]

Elimina propiedades asociadas con átomos.

La llamada `remove (a_1, p_1, ..., a_n, p_n)` elimina la propiedad `p_k` del átomo `a_k`.

La llamada `remove ([a_1, ..., a_m], [p_1, ..., p_n], ...)` elimina las propiedades `p_1, ..., p_n` de los átomos *a_1*, ..., *a_m*. Puede tener más de un par de listas.

La llamada `remove (all, p)` elimina la propiedad *p* de todos los átomos que la tengan.

Las propiedades eliminadas pueden ser de las que define el sistema, como `function`, `macro` o `mode_declare`; `remove` no elimina las propiedades definidas por `put`.

La llamada `remove ("a", operator)` o su equivalente `remove ("a", op)` elimina de *a* las propiedades de operador declaradas por `prefix`, `infix`, `nary`, `postfix`, `matchfix` o `nofix`. Nótese que el nombre del operador debe escribirse como cadena precedida de apóstrofo.

La función `remove` devuelve siempre `done` independientemente que haya algún átomo con la propiedad especificada.

La función `remove` no evalúa sus argumentos.

`scalar` [Propiedad]

`declare(a, scalar)` indica a Maxima que considere a *a* como una variable escalar.

Véase también `declare`.

`scalarp` (*expr*) [Función]

Devuelve `true` si *expr* es un número, constante o variable declarada como `scalar` con `declare`, o compuesta completamente de tales números, constantes o variables, pero que no contengan matrices ni listas.

11.3 Funciones y variables para los hechos

`activate` (*context_1*, ..., *context_n*) [Función]

Activa los contextos *context_1*, ..., *context_n*. Los hechos en estos contextos están disponibles para hacer deducciones y extraer información. Los hechos en estos contextos no se listan al invocar `facts ()`.

La variable `activecontexts` es la lista de contextos que se han activado por medio de la función `activate`.

`activecontexts` [Variable del sistema]

Valor por defecto: `[]`

La variable `activecontexts` es la lista de contextos que se han activado por medio de la función `activate`, pero que no se han activado por ser subcontextos del contexto actual.

askinteger (*expr*, *integer*) [Función]
askinteger (*expr*) [Función]
askinteger (*expr*, *even*) [Función]
askinteger (*expr*, *odd*) [Función]

La llamada `askinteger (expr, integer)` intenta determinar a partir de la base de datos de `assume` si *expr* es un entero. La función `askinteger` pide más información al usuario si no encuentra la respuesta, tratando de almacenar la nueva información en la base de datos si es posible. La llamada `askinteger (expr)` equivale a `askinteger (expr, integer)`.

La llamadas `askinteger (expr, even)` ay `askinteger (expr, odd)` intentan determinar si *expr* es un entero par o impar, respectivamente.

asksign (*expr*) [Función]

Primero intenta determinar si la expresión especificada es positiva, negativa o cero. Si no lo consigue, planteará al usuario preguntas que le ayuden a conpletar la deducción. Las respuestas del usuario son almacenadas en la base de datos durante el tiempo que dure este cálculo. El valor que al final devuelva `asksign` será `pos`, `neg` o `zero`.

assume (*pred_1*, ..., *pred_n*) [Función]

Añade los predicados *pred_1*, ..., *pred_n* al contexto actual. Si un predicado es inconsistente o redundante con los otros predicados del contexto actual, entonces no es añadido al contexto. El contexto va acumulando predicados con cada llamada a `assume`.

La función `assume` devuelve una lista cuyos miembros son los predicados que han sido añadidos al contexto, o los átomos `redundant` o `inconsistent` si fuere necesario.

Los predicados *pred_1*, ..., *pred_n* tan solo pueden ser expresiones formadas con los operadores relacionales `<` `<=` `equal` `notequal` `>=` y `>`. Los predicados no pueden estar formados por expresiones que sean del tipo igualdad `=` ni del tipo desigualdad `#`, ni tampoco pueden ser funciones de predicado como `integerp`.

En cambio, sí se reconocen predicados compuestos de la forma *pred_1* `and ...` `and` *pred_n*, pero no *pred_1* `or ...` `or` *pred_n*. También se reconoce `not` *pred_k* si *pred_k* es un predicado relacional. Expresiones de la forma `not (`*pred_1* `and` *pred_2*`)` y `not (`*pred_1* `or` *pred_2*`)` no son reconocidas.

El mecanismo deductivo de Maxima no es muy potente; existen muchas consecuencias que, siendo obvias, no pueden ser obtenidas por `is`. Se trata de una debilidad reconocida.

`assume` no gestiona predicados con números complejos. Si un predicado contiene un número complejo, `assume` devuelve `inconsistent` o `redunant`.

La función `assume` evalúa sus argumentos.

Véanse también `is`, `facts`, `forget`, `context` y `declare`.

Ejemplos:

```
(%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` [Variable opcional]

Valor por defecto: `true`

La variable `assumescalar` ayuda a controlar si una expresión `expr` para la cual `nonscalarp (expr)` es `false` va a tener un comportamiento similar a un escalar bajo ciertas transformaciones.

Sea `expr` cualquier expresión distinta de una lista o matriz, y sea también `[1, 2, 3]` una lista o una matriz. Entonces, `expr . [1, 2, 3]` dará como resultado `[expr, 2 expr, 3 expr]` si `assumescalar` es `true`, o si `scalarp (expr)` es `true`, o si `constantp (expr)` es `true`.

Si `assumescalar` vale `true`, la expresión se comportará como un escalar sólo en operaciones conmutativas, pero no en el caso de la multiplicación no conmutativa o producto matricial `..`

Si `assumescalar` vale `false`, la expresión se comportará como un no escalar.

Si `assumescalar` vale `all`, la expresión se comportará como un escalar para todas las operaciones.

`assume_pos` [Variable opcional]

Valor por defecto: `false`

Si `assume_pos` vale `true` y el signo de un parámetro x no puede ser determinado a partir del contexto actual o de otras consideraciones, `sign` y `asksign (x)` devolverán `true`. Con esto se pueden evitar algunas preguntas al usuario que se generan automáticamente, como las que hacen `integrate` y otras funciones.

By default, a parameter is x such that `symbolp (x)` or `subvarp (x)`.

Por defecto, un parámetro x es aquel para el que `symbolp (x)` o `subvarp (x)` devuelven `true`. La clase de expresiones que se consideran parámetros se puede extender mediante la utilización de la variable `assume_pos_pred`.

Las funciones `sign` y `asksign` intentan deducir el signo de una expresión a partir de los signos de los operandos que contiene. Por ejemplo, si `a` y `b` son ambos positivos, entonces `a + b` también es positivo.

Sin embargo, no es posible obviar todas las preguntas que hace `asksign`. En particular, cuando el argumento de `asksign` es una diferencia `x - y` o un logaritmo `log(x)`, `asksign` siempre solicita una respuesta por parte del usuario, incluso cuando `assume_pos` vale `true` y `assume_pos_pred` es una función que devuelve `true` para todos los argumentos.

`assume_pos_pred` [Variable opcional]

Valor por defecto: `false`

Cuando a `assume_pos_pred` se le asigna el nombre de una función o una expresión lambda de un único argumento *x*, ésta será invocada para determinar si *x* se considera un parámetro por `assume_pos`. La variable `assume_pos_pred` se ignora cuando `assume_pos` vale `false`.

La función `assume_pos_pred` es invocada por `sign` y por `asksign` con un argumento *x*, el cual puede ser un átomo, una variable subindicada o una expresión de llamada a una función. Si la función `assume_pos_pred` devuelve `true`, *x* será considerada como un parámetro por `assume_pos`.

Por defecto, un parámetro *x* es aquel para el que `symbolp (x)` o `subvarp (x)` devuelven `true`.

Véanse también `assume` y `assume_pos`.

Ejemplos:

```
(%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` [Variable opcional]

Valor por defecto: `initial`

La variable `context` da nombre al conjunto de hechos establecidos desde `assume` y `forget`. La función `assume` añade nuevos hechos al conjunto nombrado por `context`, mientras que `forget` los va eliminando. Asignando a `context` un nuevo nombre *foo* cambia el contexto actual a *foo*. Si el contexto *foo* no existe todavía, se crea automáticamente mediante una llamada a `newcontext`.

Véase `contexts` para una descripción general del mecanismo que siguen los contextos.

`contexts` [Variable opcional]

Valor por defecto: `[initial, global]`

La variable `contexts` es una lista que contiene los contextos existentes, incluyendo el actualmente activo.

El mecanismo que siguen los contextos permiten al usuario agrupar y nombrar un conjunto de hechos, que recibe el nombre de contexto. Una vez hecho esto, el usuario puede hacer que Maxima tenga en cuenta o que olvide cualquier número de hechos sin más que activar o desactivar su contexto.

Cualquier átomo simbólico puede ser el nombre de un contexto, y los hechos contenidos en tal contexto pueden ser almacenados hasta que se destruyan uno a uno mediante llamadas a la función `forget`, o que se destruyan conjuntamente invocando a `kill` para eliminar el contexto al que pertenecen.

Los contextos tienen estructura jerárquica, siendo su raíz el contexto `global`, el cual contiene información sobre Maxima que necesitan algunas funciones. Cuando en un contexto todos los hechos están activos (lo que significa que están siendo utilizados en deducciones) lo estarán también en cualquier subcontexto del contexto actual.

Cuando se comienza una sesión de Maxima, el usuario estará trabajando en un contexto llamado `initial`, el cual tiene un subcontexto de nombre `global`.

Véanse también `facts`, `newcontext`, `supcontext`, `killcontext`, `activate`, `deactivate`, `assume` y `forget`.

`deactivate` (*contexto_1*, ..., *contexto_n*) [Función]

Desactiva los contextos especificados *contexto_1*, ..., *contexto_n*.

`facts` (*item*) [Función]

`facts` () [Función]

Si *item* es el nombre de un contexto, `facts (item)` devuelve una lista con los hechos asociados al contexto especificado.

Si *item* no es el nombre de un contexto, `facts (item)` devuelve una lista con los hechos conocidos acerca de *item* en el contexto actual. Los hechos que estén activos en contextos diferentes no aparecen en la lista.

La llamada `facts ()`, sin argumentos, muestra el contexto actual.

`forget` (*pred_1*, ..., *pred_n*) [Función]

`forget` (*L*) [Función]

Borra los predicados establecidos por `assume`. Los predicados pueden ser expresiones equivalentes, pero no necesariamente idénticas, a las establecidas por `assume`.

La llamada `forget (L)`, siendo L una lista de predicados, borra todos los predicados contenidos en ella.

`is` (*expr*) [Función]

Intenta determinar si el predicado *expr* se puede deducir de los hechos almacenados en la base de datos gestionada por `assume`.

Si el predicado se reduce a `true` o `false`, `is` devuelve `true` o `false`, respectivamente. En otro caso, el valor devuelto está controlado por la variable global `prederror`. Si `prederror` vale `true`, `is` emite un mensaje de error; en caso contrario, `is` devuelve `unknown`.

La instrucción `ev(expr, pred)` (que puede escribirse como `expr, pred` en el modo interactivo) equivale a `is(expr)`.

Véanse también `assume`, `facts` y `maybe`.

Ejemplos:

`is` evalúa los predicados,

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

`is` intenta evaluar predicados a partir del conocimiento almacenado en la base de datos de `assume`.

```
(%i1) assume (a > b);
(%o1)                          [a > b]
(%i2) assume (b > c);
```

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

Si `is` no puede evaluar el valor lógico del predicado a partir de la base de datos gestionada por `assume`, la variable global `prederror` controla el comportamiento de `is`.

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

`killcontext` (*contexto_1*, ..., *contexto_n*) [Función]

Elimina los contextos *contexto_1*, ..., *contexto_n*.

Si alguno de estos contextos es el actual, el nuevo contexto activo será el primer subcontexto disponible del actual que no haya sido eliminado. Si el primer contexto no eliminado disponible es `global` entonces `initial` será usado en su lugar. Si el contexto `initial` es eliminado, se creará un nuevo contexto `initial` completamente vacío.

La función `killcontext` no elimina un contexto actualmente activo si es un subcontexto del contexto actual, o si se hace uso de la función `activate`.

La función `killcontext` evalúa sus argumentos y devuelve `done`.

`maybe` (*expr*) [Función]

Intenta determinar si el predicado *expr* se puede deducir de los hechos almacenados en la base de datos gestionada por `assume`.

Si el predicado se reduce a `true` o `false`, `maybe` devuelve `true` o `false`, respectivamente. En otro caso, `maybe` devuelve `unknown`.

La función `maybe` es funcionalmente equivalente a `is` con `prederror: false`, pero el resultado se calcula sin asignar valor alguno a `prederror`.

Véanse también `assume`, `facts` y `is`.

Ejemplos:

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

```
(%i3) maybe (x > 0);
(%o3)                         true
```

newcontext (*nombre*) [Función]

Crea un nuevo contexto vacío *nombre*, el cual tiene a `global` como su único subcontexto. El recién creado contexto pasa a ser el contexto actualmente activo.

La función `newcontext` evalúa sus argumentos y devuelve *nombre*.

sign (*expr*) [Función]

Intenta determinar el signo de *expr* en base a los hechos almacenados en la base de datos. Devuelve una de las siguientes respuestas: `pos` (positivo), `neg` (negativo), `zero` (cero), `pz` (positivo o cero), `nz` (negativo o cero), `pn` (positivo o negativo), o `pnz` (positivo, negativo o cero, lo que significa que el signo es desconocido).

supcontext (*nombre*, *contexto*) [Función]

supcontext (*nombre*) [Función]

Crea un nuevo contexto *nombre*, que tiene a *contexto* como subcontexto. El argumento *contexto* debe existir ya.

Si no se especifica *context*, se tomará como tal el actual.

11.4 Funciones y variables para los predicados

charfun (*p*) [Función]

Devuelve 0 cuando el predicado *p* toma el valor `false`, y devuelve 1 cuando vale `true`. Si el predicado toma un valor diferente de `true` y `false` (desconocido), entonces devuelve una forma nominal.

Ejemplos:

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

compare (*x*, *y*) [Función]

Devuelve un operador de comparación *op* (`<`, `<=`, `>`, `>=`, `=` o `#`) de manera que `is (`*x op y*`)` tome el valor `true`; cuando tanto *x* como *y* dependan de `%i` y *x* `#` *y*, devuelve `notcomparable`; cuando no exista tal operador o Maxima sea incapaz de determinarlo, devolverá `unknown`.

Ejemplos:

```
(%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) <=
```

La función `compare` no intenta determinar si los dominios reales de sus argumentos son conjuntos no vacíos; así,

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

Aquí, el dominio real de `acos (x^2 + 1)` es el conjunto vacío.

`equal` (`a`, *b*) [Función]

Representa la equivalencia, esto es, la igualdad de los valores.

Por sí misma, `equal` no evalúa ni simplifica. La función `is` intenta evaluar `equal` a un resultado booleano. La instrucción `is(equal(a, b))` devuelve `true` (o `false`) si y sólo si *a* y *b* son iguales (o no iguales) para todos los posibles valores de sus variables, tal como lo determina `ratsimp(a - b)`; si `ratsimp` devuelve 0, las dos expresiones se consideran equivalentes. Dos expresiones pueden ser equivalentes sin ser sintácticamente iguales (es decir, idénticas).

Si `is` no consigue reducir `equal` a `true` o `false`, el resultado está controlado por la variable global `prederror`. Si `prederror` vale `true`, `is` emite un mensaje de error; en caso contrario, `is` devuelve `unknown`.

Además de `is`, otros operadores evalúan `equal` y `notequal` a `true` o `false`; a saber, `if`, `and`, `or` y `not`.

La negación de `equal` es `notequal`.

Ejemplos:

Por sí misma, `equal` no evalúa ni 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)
```

La función `is` intenta evaluar `equal` a un resultado booleano. La instrucción `is(equal(a, b))` devuelve `true` si `ratsimp(a - b)` devuelve 0. Dos expresiones pueden ser equivalentes sin ser sintácticamente iguales (es decir, idénticas).

```
(%i1) ratsimp (x^2 - 1 - (x + 1) * (x - 1));
(%o1)                           0
(%i2) is (equal (x^2 - 1, (x + 1) * (x - 1)));
(%o2)                          true
(%i3) is (x^2 - 1 = (x + 1) * (x - 1));
(%o3)                          false
```

```
(%i4) ratsimp (x - (x + 1));
(%o4)                           - 1
(%i5) is (equal (x, x + 1));
(%o5)                          false
(%i6) is (x = x + 1);
(%o6)                          false
(%i7) ratsimp (x - y);
(%o7)                          x - y
(%i8) is (equal (x, y));
(%o8)                         unknown
(%i9) is (x = y);
(%o9)                          false
```

Si `is` no consigue reducir `equal` a `true` o `false`, el resultado está controlado por la variable 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
```

Otros operadores evalúan `equal` y `notequal` a `true` o `false`.

```
(%i1) if equal (y, y - 1) then FOO else BAR;
(%o1)                           BAR
(%i2) eq_1 : equal (x, x + 1);
(%o2)                      equal(x, x + 1)
(%i3) eq_2 : equal (y^2 + 2*y + 1, (y + 1)^2);
                          2                   2
(%o3)              equal(y  + 2 y + 1, (y + 1) )
(%i4) [eq_1 and eq_2, eq_1 or eq_2, not eq_1];
(%o4)                   [false, true, true]
```

Debido a que `not` *expr* obliga a la evaluación previa de *expr*, `not equal(a, b)` equivale 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
```

`notequal` (*a*, *b*) [Función]

Representa la negación de `equal (a, b)`.

Ejemplos:

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

`unknown` (*expr*) [Función]

Devuelve `true` si y sólo si *expr* contiene un operador o función no reconocido por el simplificador de Maxima.

`zeroequiv` (*expr*, *v*) [Función]

Analiza si la expresión *expr* de variable *v* equivale a cero, devolviendo `true`, `false` o `dontknow`.

La función `zeroequiv` tiene estas restricciones:

1. No utilizar funciones que Maxima no sepa derivar y evaluar.
2. Si la expresión tiene polos en la recta real, pueden aparecer errores en el resultado, aunque es poco probable.
3. Si la expresión contiene funciones que no son soluciones de ecuaciones diferenciales ordinarias de primer orden (como las funciones de Bessel) pueden presentarse resultados incorrectos.
4. El algoritmo utiliza evaluaciones en puntos aleatoriamente seleccionados. Esto conlleva un riesgo,aunque el algoritmo intenta minimizar el error.

Por ejemplo, `zeroequiv (sin(2*x) - 2*sin(x)*cos(x), x)` devuelve `true` y `zeroequiv (%e^x + x, x)` devuelve `false`. Por otro lado `zeroequiv (log(a*b) - log(a) - log(b), a)` devuelve `dontknow` debido a la presencia del parámetro `b`.

12 Gráficos

12.1 Introducción a los gráficos

Maxima utiliza un programa gráfico externo para hacer figuras (véase la sección Sección 12.2 Formatos gráficos). Las funciones gráficas calculan un conjunto de puntos y se los pasa al programa gráfico, junto con una serie de instrucciones. Estas instrucciones pueden pasarse al programa gráfico, bien a través de una tubería (*pipe*, en inglés), bien llamando al programa, junto con el nombre del fichero en el que se almacenan los datos. Al fichero de datos se le da el nombre `maxout.interface`, donde `interface` es el nombre del intefaz a ser utilizado (`gnuplot`, `xmaxima`, `mgnuplot` o `gnuplot_pipes`).

El fichero `maxout.interface`, si se utiliza, se almacena en la carpeta especificada por la variable *maxima_tempdir*, cuyo valor se puede cambiar por una cadena de texto que represente la ruta a una carpeta válida, en la que Maxima pueda guardar nuevos ficheros.

Una vez creado el gráfico, el fichero `maxout.interface` puede ejecutarse nuevamente con el programa externo adecuado. Si una instrucción gráfica de Maxima falla, este fichero puede ser inspeccionado a fin de encontrar el origen del problema.

Junto con las funciones gráficas descritas en esta sección, el paquete `draw` añade otras funcionalidades. Nótese que algunas opciones gráficas se llaman igual en ambos contextos gráficos, pero con diferente sintaxis; para acceder a la información de estas opciones en el ámbito de `draw`, es necesario teclear `?? opc`, donde `opc` es el nombre de la opción.

12.2 Formatos gráficos

Actualmente, Maxima utiliza dos programas gráficos externos: Gnuplot y Xmaxima. Existen varios formatos diferentes para estos programas, que pueden seleccionarse con la opción `plot_format` (véase la sección Opciones gráficas).

Los formatos gráficos se listan a continuación:

- **gnuplot** (formato por defecto para Windows)

  Se utiliza para ejecutar el programa externo Gnuplot, el cual debe estar instalado en el sistema. Las instrucciones gráficas y los datos se almacenan en el fichero `maxout.gnuplot`.

- **gnuplot_pipes** (formato por defecto para plataformas distintas de Windows)

  Este formato no está disponible en plataformas Windows. Es similar al formato `gnuplot`, excepto por el hecho de que las instrucciones son enviadas a Gnuplot por una tubería, mientras que los datos se almacenan en el fichero `maxout.gnuplot_pipes`. Mediante esta técnica, un único proceso de Gnuplot se mantiene activo y sucesivos gráficos son enviados al mismo proceso, a menos que la tubería a Gnuplot se cierre con la función `gnuplot_close()`. Cuando se utiliza este formato, se puede utilizar la función `gnuplot_replot` para modificar un gráfico que ya había sido representado previamente en la pantalla (véase `gnuplot_replot`).

  Este formato debería ser utilizado únicamente cuando se representen los gráficos por pantalla; para gráficos almacenados en ficheros, mejor utilizar el formato `gnuplot`.

- **mgnuplot**

  Mgnuplot es una interfaz para Gnuplot basada en Tk. Se incluye en la distribución de Maxima. Mgnuplot ofrece una interface gráfica de usuario rudimentaria para gnuplot, pero tiene algunas mejoras respecto de la interface propia de gnuplot. Mgnuplot requiere de una instalación externa de Gnuplot y de Tcl/Tk.

- **xmaxima**

  Xmaxima es un interfaz gráfico Tcl/Tk de Maxima, que también se puede utilizar para representar gráficos cuando Maxima se ejecuta desde la consola o desde otros interfaces. Para utilizar este formato, debe estar instalado junto con Maxima. Si Maxima se ejecuta desde el propio Xmaxima, las instrucciones gráficas y los datos se envían por el mismo canal de comunicación que se establece entre Maxima y Xmaxima (un *socket*). Cuando Maxima se ejecuta desde una consola o desde otro interfaz, las instrucciones gráficas y los datos se almacenan en un fichero de nombre `maxout.xmaxima`, que le es pasado a Xmaxima como argumento

  En versiones anteriores, este formato se llamaba `openmath`, cuyo nombre se sigue aceptando como sinónimo de `xmaxima`.

12.3 Funciones y variables para gráficos

`contour_plot` (*expr*, *x_range*, *y_range*, *options*, ...) [Función]

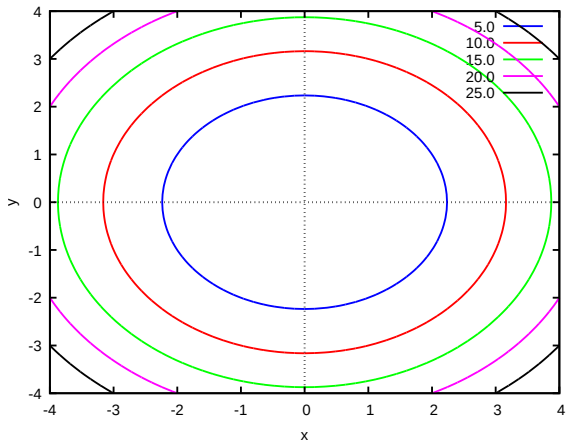
Dibuja las curvas de nivel de *expr* en el rectángulo *x_range* por *y_range*. Cualesquiera otros argumentos adicionales se tratan como en `plot3d`.

`contour_plot` sólo trabaja con los métodos `gnuplot` o `gnuplot_pipes`.

Véase también `implicit_plot`.

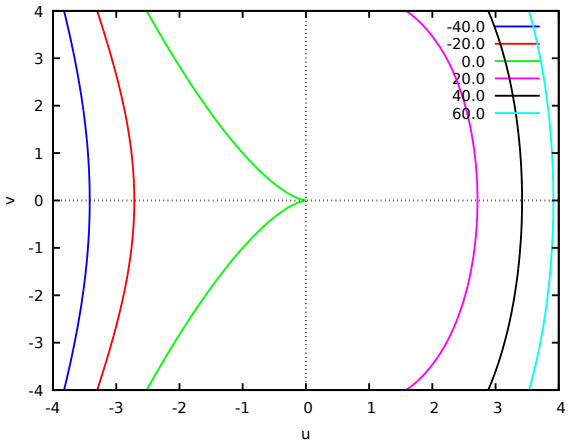
Ejemplos:

```
(%i1) contour_plot (x^2 + y^2, [x, -4, 4], [y, -4, 4])$
```



Se pueden añadir cualesquiera opciones que acepte `plot3d`; por ejemplo, la opción `legend` con un valor `false`, para eliminar la leyenda. Gnuplot muestra por defecto tres líneas de contorno, pero para aumentar el número de niveles es necesario añadir algún código nativo de Gnuplot:

```
(%i1) contour_plot (u^3 + v^2, [u, -4, 4], [v, -4, 4],
                    [legend,false],
                    [gnuplot_preamble, "set cntrparam levels 12"])$
```



`get_plot_option` (*keyword*, *index*) [Función]

Devuelve el valor actual de la opción *keyword* almacenada en la variable global `plot_options`. Si *index* toma el valor 1, devuelve el propio valor de *keyword*; si vale 2 le añade el primer parámetro, y así sucesivamente.

Véanse también `plot_options`, `set_plot_option` y la sección Opciones gráficas.

`make_transform` ([*var1*, *var2*, *var3*], *fx*, *fy*, *fz*) [Función]

Devuelve una función que se puede utilizar con la opción `transform_xy` de `plot3d`. Las tres variables ficticias *var1*, *var2* y *var3* representan las tres variables de la función `plot3d`, las dos primeras independientes y la tercera dependiente. Las tres funciones *fx*, *fy* y *fz* deben depender solo de las tres variables anteriores y retornar las correspondientes x, y, z que se deben dibujar. Hay dos transformaciones predefinidas: `polar_to_xy` y `spherical_to_xyz`.

Véanse `polar_to_xy` y `spherical_to_xyz`.

`polar_to_xy` [Símbolo del sistema]

Cuando a la opción `transform_xy` de `plot3d` se le pasa el valor `polar_to_xy`, se interpretarán las dos primeras variables independientes como polares, transformándolas luego a coordenadas cartesianas.

`plot2d` (*plot*, *x_range*, ..., *options*, ...) [Función]
`plot2d` ([*plot_1*, ..., *plot_n*], ..., *options*, ...) [Función]
`plot2d` ([*plot_1*, ..., *plot_n*], *x_range*, ..., *options*, ...) [Función]

Donde *plot*, *plot_1*, ..., *plot_n* pueden ser expresiones, nombres de funciones o una lista de cualquiera de las siguientes formas: `[discrete, [x1, ..., xn], [y1, ..., yn]]`, `[discrete, [[x1, y1], ..., [xn, ..., yn]]` o `[parametric, x_expr, y_expr, t_range]`.

Muestra un gráfico de una o más expresiones como función de una variable.

La función `plot2d` representa uno o más gráficos en dos dimensiones. Las expresiones o nombres de funciones que se utilicen para definir curvas deben depender todas ellas

de una única variable *var*, siendo obligatorio utilizar *x_range* para nombrar la variable y darle sus valores mínimo y máximo usando la siguiente sintaxis: `[variable, min, max]`.

Un gráfico también se puede definir de forma discreta o paramétrica. La forma discreta se utiliza para dibujar un conjunto de puntos de coordenadas dadas. Un gráfico discreto se define como una lista que empiezan con la palabra clave *discrete* seguida de una o dos listas de valores numéricos. Cuando haya dos listas, ambas deben ser de igual longitud, la primera se interpreta como la de abscisas y la segunda de ordenadas. Cuando haya una lista siguiendo la clave *discrete*, cada uno de sus elementos debe ser a su vez una lista de solo dos valores, correspondientes a las coordenadas x e y.

Un gráfico paramétrico se define como una lista que empieza con la palabra clave *parametric*, seguida de dos expresiones o nombres de funciones y un rango paramétrico. El rango paramétrico debe ser una lista formada con el nombre del parámetro seguido de sus valores mínimo y máximo: `[param, min, max]`. El gráfico se formará con los puntos cuyas coordenadas devuelvan las dos expresiones o funciones, según *param* aumente desde *min* hasta *max*.

La especificación del rango para el eje vertical es opcional y toma la forma `[y, min, max]` (*y* se utiliza siempre para el eje vertical). En caso de utilizar esta opción, el gráfico mostrará exactamente ese rango vertical, independientemente de los valores alcanzados por los elementos gráficos. Si no se especifica el rango vertical, se ajustará a los valores extremos alcanzados por las ordenadas de los puntos que aparezcan en el gráfico.

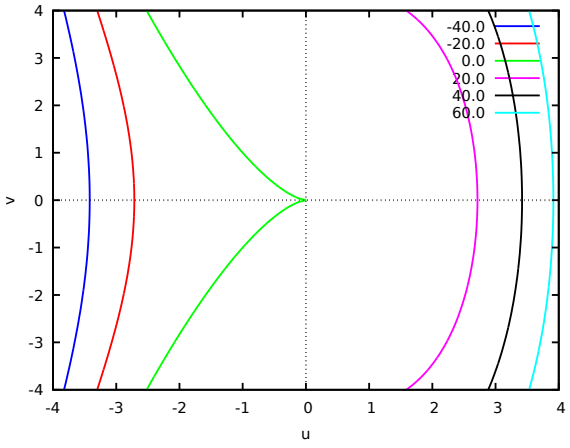
Cualesquiera otras opciones deben ser listas, comenzando con el nombre de la opción seguido de uno o más valores. Véase `plot_options`.

Si hay varias expresiones para ser dibujadas, se mostrará una leyenda que identifique a cada una de ellas. Las etiquetas a utilizar pueden especificarse con la opción *legend*. Si no se utiliza esta opción, Maxima creará etiquetas a partir de las expresiones o nombres de funciones.

Ejemplos:

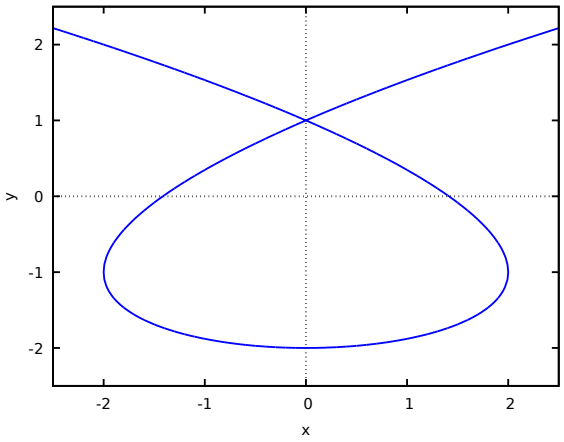
Dibujando la función sinusoidal:

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



Si la función crece rápidamente puede ser necesario limitar los valores del eje vertical:

```
(%i1) plot2d (sec(x), [x, -2, 2], [y, -20, 20])$
plot2d: some values were clipped.
```



El aspecto del gráfico puede ser diferente dependiendo del programa gráfico utilizado. Por ejemplo, cuando se desactiva el marco, Xmaxima dibuja los ejes como flechas:

```
(%i1) plot2d ( x^2-1, [x, -3, 3], [y, -2, 10],
                      [box, false], [plot_format, xmaxima])$
```

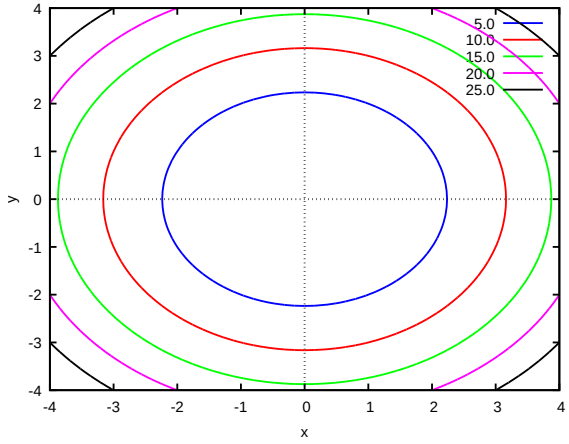
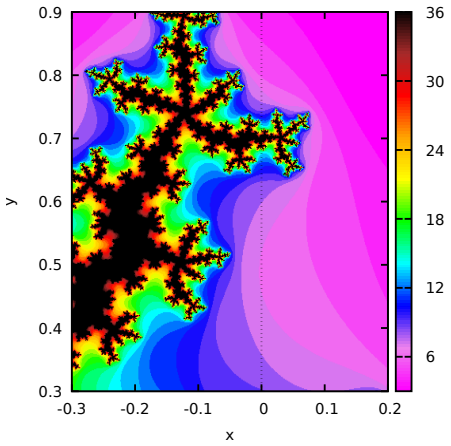


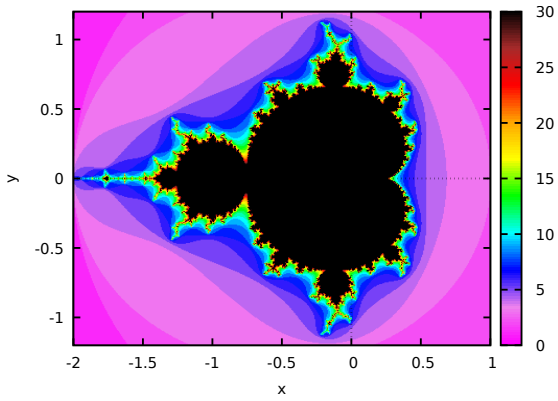
Gráfico con escala logarítmica:

```
(%i1) plot2d (exp(3*s), [s, -2, 2], [logy])$
```



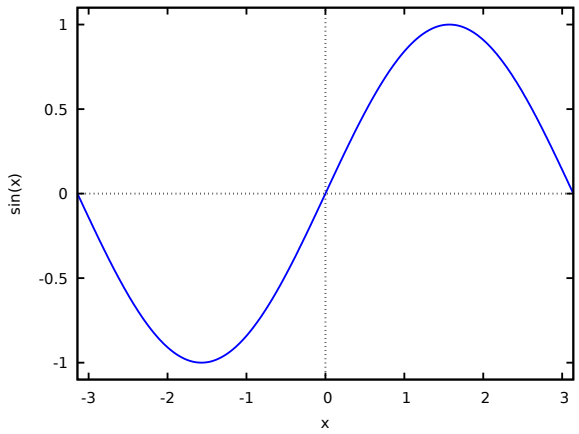
Dibujando funciones especificadas por su nombre:

```
(%i1) F(x) := x^2 $
(%i2) :lisp (defun |$g| (x) (m* x x x))
$g
(%i2) H(x) := if x < 0 then x^4 - 1 else 1 - x^5 $
(%i3) plot2d ([F, G, H], [u, -1, 1], [y, -1.5, 1.5])$
```



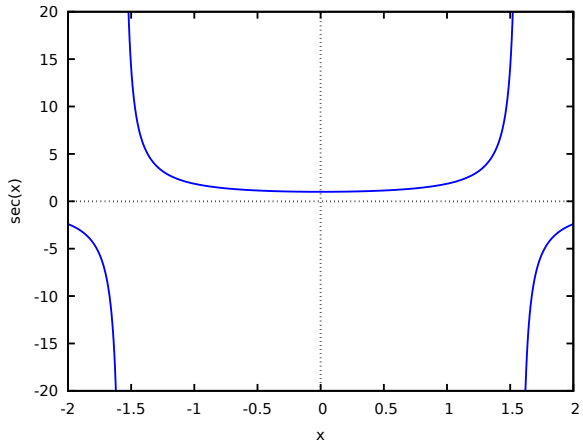
Ejemplo de función paramétrica. Curva de la mariposa:

```
(%i1) r: (exp(cos(t))-2*cos(4*t)-sin(t/12)^5)$
(%i2) plot2d([parametric, r*sin(t), r*cos(t),
              [t, -8*%pi, 8*%pi], [nticks, 2000]])$
```



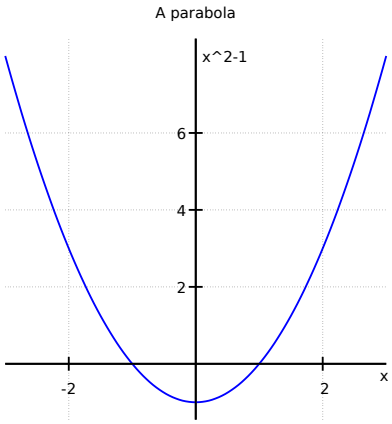
Una circunferencia de dos vueltas y solo siete puntos:

```
(%i1) plot2d ([parametric, cos(t), sin(t),
               [t, -2*%pi, 2*%pi], [nticks, 8]])$
```



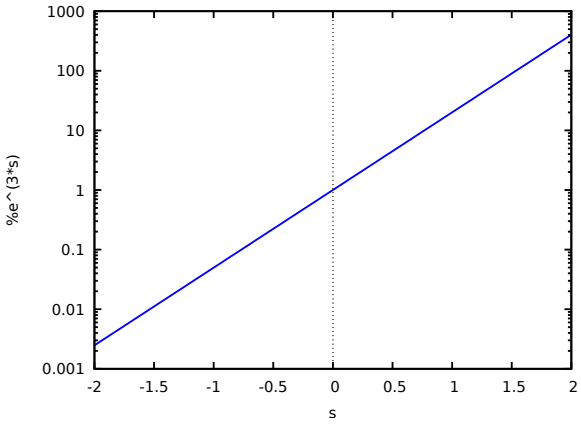
Dibujo de una función junto con la representación paramétrica de una circunferencia. El tamaño del gráfico se ha ajustado con las opciones x e y para que la circunferencia no se vea como una elipse. Estos valores son aceptables para el terminal Postscript utilizado para producir este gráfico, y puede seer necesario adaptar los valores para otros terminales:

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



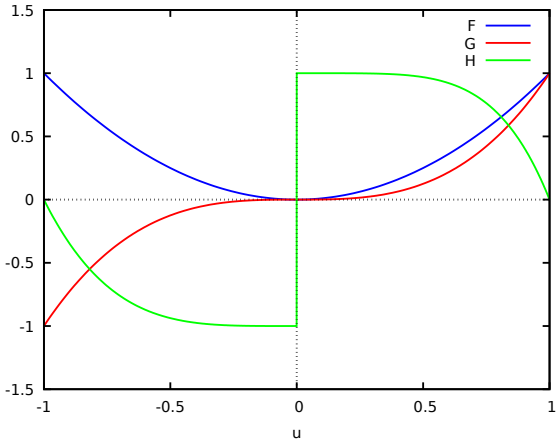
Puntos discretos definiendo separadamente las coordenadas x e y:

```
(%i1) plot2d ([discrete, [10, 20, 30, 40, 50],
                         [.6, .9, 1.1, 1.3, 1.4]])$
```



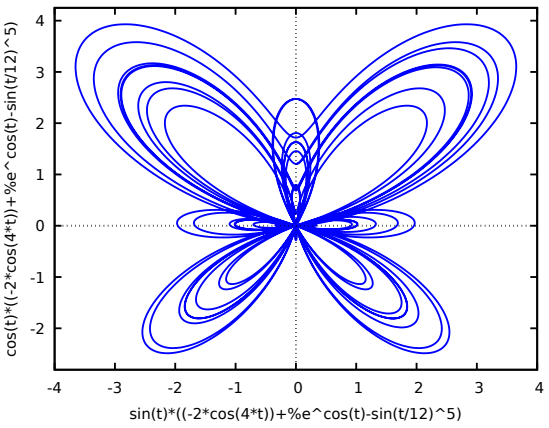
Los mismos puntos del ejemplo anterior, pero definiendo una a una las coordenadas y sin segmentos que unan los puntos:

```
(%i1) plot2d([discrete, [[10, .6], [20, .9], [30, 1.1],
                         [40, 1.3], [50, 1.4]]],
                        [style, points])$
```



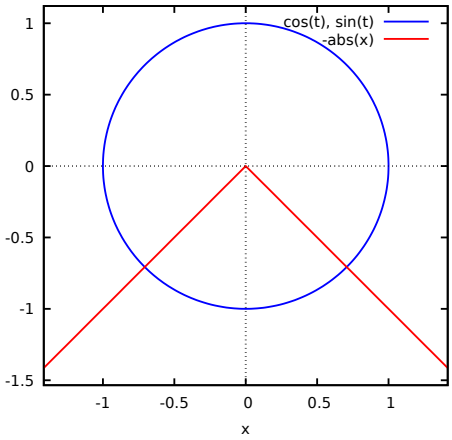
En este ejemplo, se guarda una tabla de tres columnas en el archivo `data.txt`, que luego será leído para representar las dos últimas columnas:

```
(%i1) with_stdout ("data.txt", for x:0 thru 10 do
                                    print (x, x^2, x^3))$
(%i2) data: read_matrix ("data.txt")$
(%i3) plot2d ([discrete, transpose(data)[2], transpose(data)[3]],
  [style,points], [point_type,diamond], [color,red])$
```



Un gráfico de datos empíricos junto con su modelo teórico:

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



`plot3d` (*expr*, *x_range*, *y_range*, ..., *options*, ...) [Función]
`plot3d` ([*expr_1*, ..., *expr_n*], *x_range*, *y_range*, ..., *options*, ...) [Función]

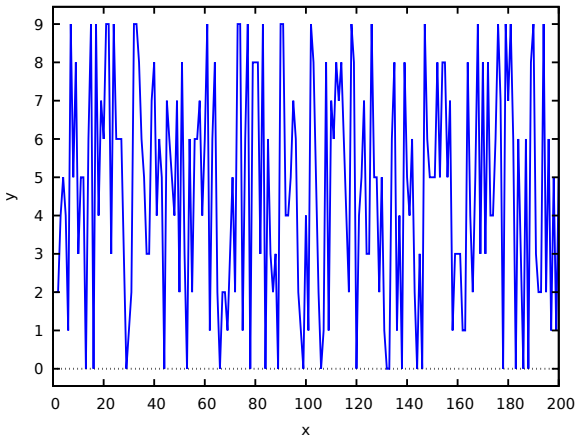
Dibuja una o más superficies definidas como funciones de dos variables o en forma paramétrica

Las funciones a dibujar se pueden especificar como expresiones o nombres de funciones. Puede utilizarse el ratón para hacer girar el gráfico y observarlo desde distintos ángulos.

Ejemplos:

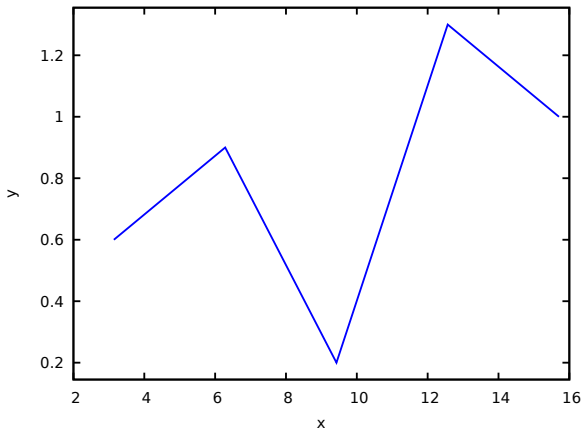
Representación de una función:

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



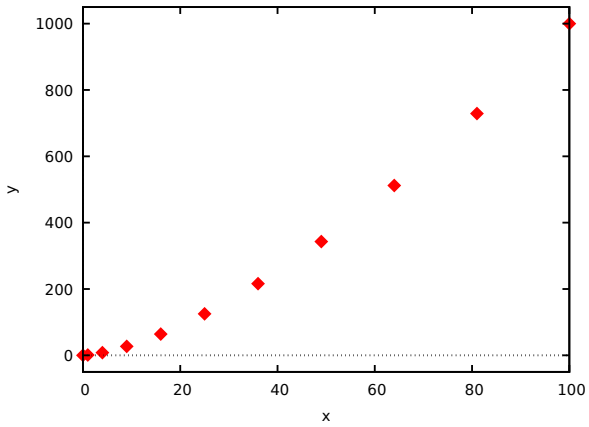
Uso de la opción z para acotar una función que tiende a infinito (en este caso, la función tiende a menos infinito en los ejes x e y):

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



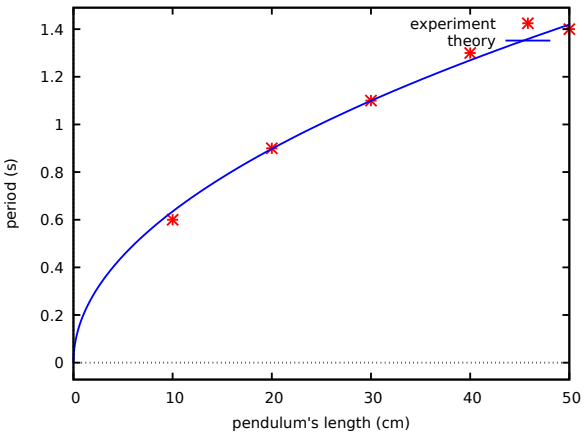
Los valores infinitos de z se pueden obviar eligiendo una retícula que no coincida con las asíntotas; este ejemplo también muestra cómo seleccionar las paletas predefinidas, en este caso la número 4:

```
(%i1) plot3d (log (x^2*y^2), [x, -2, 2], [y, -2, 2],
         [grid, 29, 29],
         [palette, get_plot_option(palette,5)])$
```



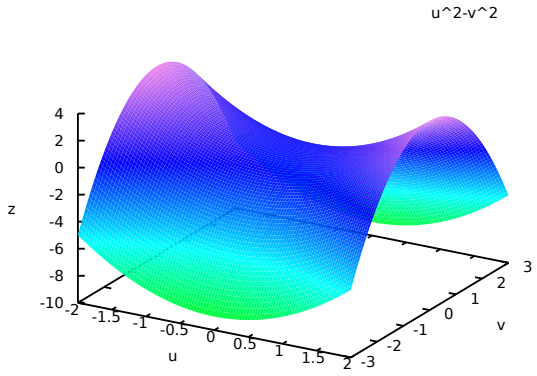
Dos superficies en el mismo gráfico, compartiendo el mismo dominio; en Gnuplot, ambas superfifies comparten la misma paleta:

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



Las mismas superficies, pero con diferentes dominios; en Xmaxima cada superficies usa una paleta diferente, elegida de la lista definida por la opción `palette`:

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



La botella de Klein, definida paramétricamente:

```
(%i1) expr_1:5*cos(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3.0)-10.0$
(%i2) expr_2:-5*sin(x)*(cos(x/2)*cos(y) + sin(x/2)*sin(2*y) + 3.0)$
(%i3) expr_3: 5*(-sin(x/2)*cos(y) + cos(x/2)*sin(2*y))$
(%i4) plot3d ([expr_1, expr_2, expr_3], [x, -%pi, %pi],
        [y, -%pi, %pi], [grid, 40, 40])$
```

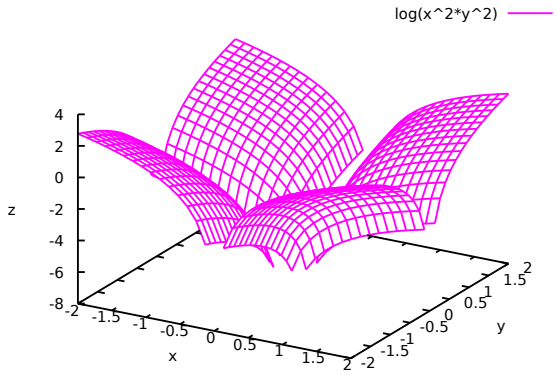
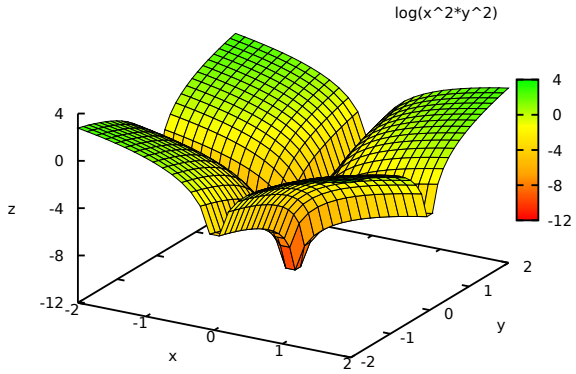


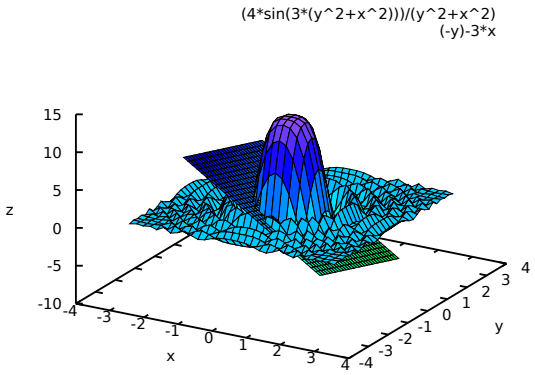
Gráfico de un armónico esférico, utilizando las transformaciones predefinidas `spherical_to_xyz`:

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



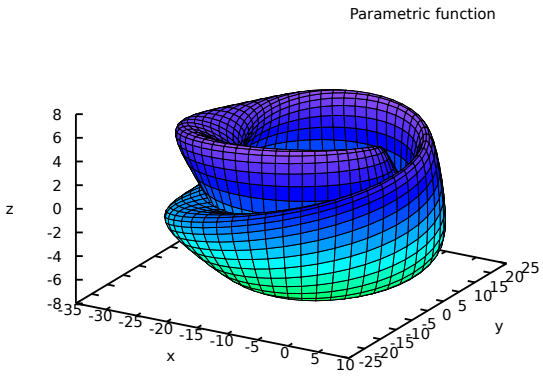
Uso de la transformación predefinida `polar_to_xy`. Este ejemplo también muestra cómo eliminar el marco y la leyenda:

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



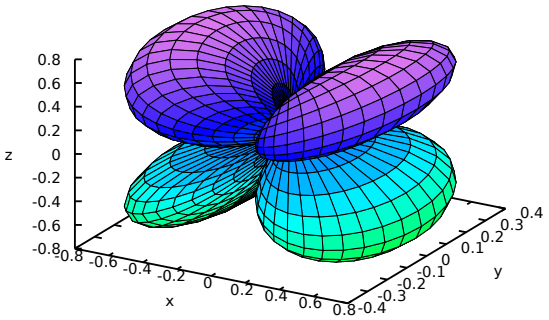
Dibujo de una esfera utilizando la transformación esférica. En Xmaxima, los tres ejes utilizan escalas proporcionales, manteniendo la forma simétrica de la esfera. Se utiliza una paleta con color degradado:

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



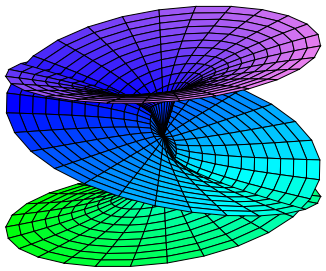
Definición de una función con dos variables utilizando una matriz. Nótese la comilla simple en la definición de la función para prevenir que `plot3d` falle al detectar que la matriz necesita índices enteros:

```
(%i1) M: matrix([1, 2, 3, 4], [1, 2, 3, 2], [1, 2, 3, 4],
                [1, 2, 3, 3])$
(%i2) f(x, y) := float('M [round(x), round(y)])$
(%i3) plot3d (f(x,y), [x, 1, 4], [y, 1, 4], [grid, 4, 4])$
apply: subscript must be an integer; found: round(x)
```



Asignando a la opción `elevation` el valor cero, una superficie puede verse como una aplicación en la que cada color representa un nivel diferente. La opción `colorbox` se utiliza para mostrar la correspondencia entre colores y niveles; las líneas de la retícula se desactivan para facilitar la visualización de los colores:

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



Véase también la sección Opciones gráficas.

`plot_options` [Variable global]

Los elementos de esta lista establecen las opciones por defecto para los gráficos. Si una opción está presente en una llamada a `plot2d` o a `plot3d`, este valor adquiere prevalencia sobre las opciones por defecto. En otro caso se utilizará el valor que tenga en `plot_options`. Las opciones por defecto se asignan mediante la función `set_plot_option`.

Cada elemento de `plot_options` es una lista de dos o más elementos, el primero de los cuales es el nombre de la opción, siendo los siguientes los valores que toma. En algunos casos el valor asignado es a su vez una lista, que puede contener varios elementos.

Véanse también `set_plot_option`, `get_option` y la sección Opciones gráficas.

`set_plot_option` (*option*) [Función]

Acepta la mayor parte de opciones listadas en la sección Opciones gráficas y las almacena en la variable global `plot_options`.

La función `set_plot_option` evalúa su argumento y devuelve la lista completa `plot_options` tal como queda después de la actualización.

Véanse también `plot_options`, `get_option` y la sección Opciones gráficas.

Ejemplos:

Modificación de los valores para `grid`.

```
(%i1) set_plot_option ([grid, 30, 40]);
(%o1) [[t, - 3, 3], [grid, 30, 40], [transform_xy, false],
[run_viewer, true], [axes, true], [plot_format, gnuplot_pipes],
[color, blue, red, green, magenta, black, cyan],
[point_type, bullet, circle, plus, times, asterisk, box, square,
triangle, delta, wedge, nabla, diamond, lozenge],
[palette, [hue, 0.25, 0.7, 0.8, 0.5],
[hue, 0.65, 0.8, 0.9, 0.55], [hue, 0.55, 0.8, 0.9, 0.4],
[hue, 0.95, 0.7, 0.8, 0.5]], [gnuplot_term, default],
[gnuplot_out_file, false], [nticks, 29], [adapt_depth, 5],
[gnuplot_preamble, ], [gnuplot_default_term_command,
set term pop], [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], [plot_realpart, false]]
```

`spherical_to_xyz` [Símbolo del sistema]

Cuando a la opción `transform_xy` de `plot3d` se le pasa el valor `spherical_to_xyz`, se interpretarán las variables independientes como esféricas, transformándolas luego a coordenadas cartesianas.

12.4 Opciones gráficas

Todas las opciones consisten en una lista que comienza con una palabra clave seguida de uno o más valores. La mayor parte de las opciones pueden utilizarse con cualquiera de las funciones gráficas *plot2d*, *plot3d*, *contour_plot* y *implicit_plot*, o en la función *set_plot_option*. Las excepciones se indican en la lista siguiente.

`adapt_depth` [*adapt_depth*, *integer*] [Opción para plot]

Valor por defecto: 5

Número máximo de particiones utilizado por el algoritmo adaptativo de representación gráfica.

`axes` [*axes*, *symbol*] [Opción para plot]

Valor por defecto: `true`

El argumento *symbol* puede valer `true`, `false`, `x` o `y`. Si vale `false`, no se mostrarán los ejes; si es igual a `x` o `y`, solo ese eje será el que se representa; si vale `true`, se mostrarán ambos ejes.

Esta opción solo es relevante para `plot2d` y `implicit_plot`.

`azimuth` [*azimuth*, *number*] [Opción para plot]

Valor por defecto: 30

Un gráfico `plot3d` se puede interpretar como comenzando horizontalmente en el plano xy; a partir de ahí, la coordenada z se eleva perpendicularmente al papel. El eje z gira entonces alrededor del eje x un ángulo igual a `elevation`, luego gira el plano xy alrededor del nuevo eje z un ángulo `azimuth`. Esta opción establece el valor para `azimuth` en grados sexagesimales.

Véase también `elevation`.

`box` [*box*, *symbol*] [Opción para plot]

Valor por defecto: `true`

Si vale `true`, se representará el marco para el gráfico; si vale `false`, no.

`color` [*color*, *color_1*, ..., *color_n*] [Opción para plot]

Valor por defecto: blue, red, green, magenta, black, cyan

Define el color para las curvas en `plot2d` y `implicit_plot`. En `plot3d` define los colores para el enrejado de las superficies si no se utiliza la paleta; una cara de la superficie tendrá *color_1* y la otra *color_2*, o el mismo color si se especifica solo uno.

Si hay más curvas o superficies que caras, los colores se repetirán secuencialmente. Con Gnuplot, los colores pueden ser: azul, rojo, verde, magenta, negro y cián; con Xmaxima, los colores pueden ser esos mismos, o una cadena de texto que comienza con el carácter `#` seguido de seis dígitos hexadecimales: dos para la componente roja, otros dos para la verde y otros dos para la azul. Si se introduce un nombre de color no reconocido, en su lugar se utilizará el negro.

`colorbox` [*colorbox*, *symbol*] [Opción para plot]

Valor por defecto: `false`

El argumento *symbol* puede valer `true` o `false`. Si vale `true` y `plot3d` utiliza la paleta de colores para representar diferentes valores de z, se dibujará un rectángulo en la parte derecha, indicando los colores utilizados para los diferentes valores de z. Esta opción no funciona en Xmaxima.

`elevation` [*elevation*, *number*] [Opción para plot]

Valor por defecto: 60

Un gráfico `plot3d` se puede interpretar como comenzando horizontalmente en el plano xy; a partir de ahí, la coordenada z se eleva perpendicularmente al papel. El eje z gira entonces alrededor del eje x un ángulo igual a `elevation`, luego gira el plano xy alrededor del nuevo eje z un ángulo `azimuth`. Esta opción establece el valor para `elevation` en grados sexagesimales.

Véase también `azimuth`.

`grid` [*grid*, *integer*, *integer*] [Opción para plot]

Valor por defecto: 30, 30

Establece el número de puntos para los puntos de la rejilla en las direcciones x e y en escenas 3D.

`legend` [*legend*, *string_1*, ..., *string_n*] [Opción para plot]
`legend` [*legend*, `false`] [Opción para plot]

Especifica las etiquetas para los gráficos en los que aparecen varios objetos. Si hay más expresiones que etiquetas, éstas se repetirán. Con el valor `false` no se mostrarán etiquetas. Por defecto se pasarán los nombres de las expresiones o funciones, o las palabras `discrete1`, `discrete2`, ..., para gráficos de puntos. Esta opción no se puede utilizar con *set_plot_option*.

`logx` [*logx*] [Opción para plot]

Hace que el eje de abscisas se dibuje en la escala logarítmica. Esta opción no se puede utilizar con *set_plot_option*.

`logy` [*logy*] [Opción para plot]

Hace que el eje de ordenadas se dibuje en la escala logarítmica. Esta opción no se puede utilizar con *set_plot_option*.

`mesh_lines_color` [*mesh_lines_color*, *color*] [Opción para plot]

Valor por defecto: `black`

Establece el color del enrejado en los gráficos creados por `plot3d` cuando se utiliza una paleta. Acepta los mismos colores que la opción `color`. También se le puede dar el valor `false` para eliminar el enrejado.

`nticks` [*nticks*, *integer*] [Opción para plot]

Valor por defecto: 29

Cuando se dibujan funciones con `plot2d`, establece el número inicial de puntos utilizado por la rutina gráfica adaptativa. Cuando se dibujan funciones con `plot3d` o paramétricas con `plot2d`, su valor es igual al número de puntos que se representarán en el gráfico.

`palette` [*palette*, [*palette_1*], ..., [*palette_n*]] [Opción para plot]
`palette` [*palette*, `false`] [Opción para plot]

Valor por defecto: [hue, 0.25, 0.7, 0.8, 0.5], [hue, 0.65, 0.8, 0.9, 0.55], [hue, 0.55, 0.8, 0.9, 0.4], [hue, 0.95, 0.7, 0.8, 0.5]

Puede consistir en una paleta o en una lista de varias paletas. Cada paleta es una lista con una palabra clave seguida de cuatro números. Los tres primeros números, que deben tomar valores entre 0 y 1, definen el matiz, la saturación y el valor de un color básico a asignar al mínimo valor de z. La palabra clave especifica cuál de los tres atributos (`hue`, `saturation` o `value`) incrementará de acuerdo con los valores de z. El último número se corresponde con el incremento del máximo valor de z. Este último número puede ser mayor que 1 o negativo.

Gnuplot solo utiliza la primera paleta de la lista; Xmaxima utilizará las paletas de la lista secuencialmente cuando haya que representar varias superficies conjuntamente; si el número de paletas no es suficiente, se repetirán también de forma secuencial.

El color del enrrejado de la superficie se establece con `mesh_lines_color`. Si `palette` tiene el valor `false`, las superficies se representan solo con el enrrejado; en tal caso, el color de las líneas será el determinado por `color`.

`plot_format` [*plot_format, format*] [Opción para plot]
Valor por defecto: en sistemas Windows, `gnuplot`; en otros, `gnuplot_pipes`

Establece el formato a utlizar por las rutinas gráficas.

Debe tomar uno de los siguientes valores: `gnuplot`, `xmaxima`, `mgnuplot` o `gnuplot_pipes`.

`plot_realpart` [*plot_realpart, symbol*] [Opción para plot]
Valor por defecto: `false`

Cuando vale `true`, se representa gráficamente la parte real de las funciones; equivale a ejecutar `realpart(`*`función`*`)`. Si vale `false`, no se representa nada cuando la función no devuelva un valor real. Por ejemplo, si `x` es negativo, `log(x)` devuelve un valor negativo cuya parte real es `log(abs(x))`; en tal caso, si `plot_realpart` vale `true`, `log(-5)` se representa como `log(5)`, mientras que no se representa nada si `plot_realpart` vale `false`.

`point_type` [*point_type, type_1, ..., type_n*] [Opción para plot]
Valor por defecto: `bullet`, `circle`, `plus`, `times`, `asterisk`, `box`, `square`, `triangle`, `delta`, `wedge`, `nabla`, `diamond`, `lozenge`

En Gnuplot, cada conjunto de puntos que vaya a ser representado con los estilos `points` o `linespoints` se hará con objetos tomados de esta lista en orden secuencial. Si hay más conjuntos de puntos que objetos en la lista, se irán repitiendo de forma secuencial. Los objetos que pueden ser utilizados son: `bullet`, `circle`, `plus`, `times`, `asterisk`, `box`, `square`, `triangle`, `delta`, `wedge`, `nabla`, `diamond` o `lozenge`

`psfile` [*psfile, string*] [Opción para plot]
Guarda el gráfico en formato Postscript con nombre *string* en lugar de mostrarlo en pantalla. Por defecto, el fichero se creará en la carpeta definida en la variable *maxima_tempdir*, cuyo valor se podrá cambiar para almacenar el fichero en otra carpeta.

`run_viewer` [*run_viewer, symbol*] [Opción para plot]
Valor por defecto: `true`

Controla si el visor apropiado para la salida gráfica debe ejecutarse o no.

`style` [*style, type_1, ..., type1_n*] [Opción para plot]
`style` [*style,* [*style_1*]*, ...,* [*style_n*]] [Opción para plot]
Valor por defecto: *lines* (dibuja todos los puntos unidos por líneas de ancho 1 y con el primer color de la lista de la opción `color`).

Estilos a utilizar para las funciones o conjuntos de datos en gráficos 2d. A la palabra `style` debe seguirle uno o más estilos. Si hay más funciones o conjuntos de datos que estilos, éstos se repetirán. Los estilos que se admiten son: *lines* para segmentos lineales, *points* para puntos aislados, *linespoints* para segmentos y puntos, *dots* para pequeños puntos aislados. Gnuplot también acepta el estilo *impulses*.

Los estilos se pueden escribir como elementos de una lista, junto con algunos parámetros adicionales. *lines* acepta uno o dos números: el ancho de la línea y un entero que identifica el color. Los códigos de color por defecto son: 1, azul; 2, rojo; 3, magenta; 4, naranja; 5, marrón; 6, verde lima; 7, aguamarina. En caso de utilizar Gnuplot con un terminal diferente de X11, estos colores pueden cambiar; por

ejemplo, bajo la opción [*gnuplot_term*,*ps*], el índice 4 se corresponde con el negro en lugar del naranja.

points acepta uno, dos o tres parámetros; el primer parámetro es el radio de los puntos, el segundo es un entero para seleccionar el color, con igual codificación que en *lines* y el tercer parámetro sólo es utilizado por Gnuplot y hace referencia a varios objetos para representar los puntos. Los tipos de objetos disponibles son: 1, círculos rellenos; 2, circunferencias; 3, `+`; 4, x; 5, *; 6, cuadrados rellenos; 7, cuadrados huecos; 8, triángulos rellenos; 9, triángulos huecos; 10, triángulos rellenos invertidos; 11, triángulos huecos invertidos; 12, rombos rellenos; 13, rombos huecos.

linesdots acepta hasta cuatro parámetros: ancho de línea, radio de los puntos, color y tipo de objetos para representar puntos.

Véanse también `color` y `point_type`.

`transform_xy` [*transform_xy*, *symbol*] [Opción para plot]
Valor por defecto: `false`

La variable *symbol* puede ser `false` o el resultado devuelto por la función `transform_xy`. Si es distinto de `false`, se utiliza para transformar las tres coordenadas en `plot3d`.

Véanse `make_transform`, `polar_to_xy` y `spherical_to_xyz`.

`x` [*x*, *min*, *max*] [Opción para plot]
Cuando se utiliza como primera opción en una instrucción para un gráfico 2D (o cualquiera de las dos primeras en un gráfico 3D), indica que su primera variable independiente es x y ajusta su rango. También se puede utilizar después de la primera opción (o después de la segunda opción en un gráfico 3D) para definir el dominio horizontal que se representará en el gráfico.

`xlabel` [*xlabel*, *string*] [Opción para plot]
Especifica la etiqueta para el primer eje. Si no se utiliza esta opción, la etquieta será el nombre de la variable independiente, cuando se utilicen `plot2d` o `implicit_plot`, o el nombre de la primera variable cuando se utilicen `plot3d` o `contour_plot`, o la primera expresión en el caso de una curva paramétrica. No puede utilizarse con *set_plot_option*.

`y` [*y*, *min*, *max*] [Opción para plot]
Cuando se utiliza como una de las dos primeras opciones en `plot3d`, indica que una de las variables independientes es "y" y ajusta su rango. En otro caso, define el dominio de la segunda variable que se mostrará en el gráfico.

`ylabel` [*ylabel*, *string*] [Opción para plot]
Especifica la etiqueta para el segundo eje. Si no se utiliza esta opción, la etiqueta será "y", cuando se utilicen `plot2d` o `implicit_plot`, o el nombre de la segunda variable cuando se utilicen `plot3d` o `contour_plot`, o la segunda expresión en el caso de una curva paramétrica. No puede utilizarse con *set_plot_option*.

`z` [*z*, *min*, *max*] [Opción para plot]
Se utiliza en `plot3d` para ajustar el rango de valores de z que se mostrará en el gáfico.

`zlabel` [*zlabel, string*] [Opción para plot]
Especifica la etiqueta para el tercer eje cuando se utiliza `plot3d`. Si no se utiliza esta opción, la etiqueta será "z" en el caso de superficies, o la tercera expresión en el caso de una curva paramétrica. No puede utilizarse con *set_plot_option* y se ignora en `plot2d` y `implicit_plot`.

12.5 Opciones para Gnuplot

Hay varias opciones gráficas que son específicas de Gnuplot. Algunas de ellas son comandos propios de Gnuplot que se especifican como cadenas de texto. Consúltese la documentación de Gnuplot para más detalles.

`gnuplot_term` [Opción para plot]
Establece el terminal de salida para Gnuplot.

- **default** (valor por defecto)

  Gnuplot muestra el gráfico en una ventana gráfica.

- **dumb**

  Gnuplot muestra el gráfico en la consola de Maxima en estilo ASCII artístico.

- **ps**

  Gnuplot genera código en lenguaje PostScript. Si a la opción `gnuplot_out_file` se le da el valor *filename*, Gnuplot escribe el código PostScript en *filename*. En caso contrario, se guarda en el archivo `maxplot.ps`.

- Cualquier otro terminal admitido por Gnuplot.

  Gnuplot puede generar gráficos en otros muchos formatos, tales como png, jpeg, svg etc. Para crear gráficos en cualquera de estos formatos, a la opción `gnuplot_term` se le puede asignar cualquiera de los terminales admitidos por Gnuplot, bien por su nombre (símbolo) bien con la especificación completa del terminal (cadena). Por ejemplo, `[gnuplot_term,png]` guarda el gráfico en formato PNG (Portable Network Graphics), mientras que `[gnuplot_term,"png size 1000,1000"]` lo hace con dimensiones 1000x1000 píxeles. Si a la opción `gnuplot_out_file` se le da el valor *filename*, Gnuplot escribe el código PostScript en *filename*. En caso contrario, se guarda en el archivo `maxplot.`*term*, siendo *term* el nombre del terminal.

`gnuplot_out_file` [Opción para plot]
Cuando se utiliza conjuntamente con la opción `gnuplot_term`, puede utilizarse para almacenar el gráfico en un fichero en uno de los formatos aceptados por Gnuplot. Si se quiere crear un fichero Postscript se puede utilizar la opción `psfile`, que también funciona con Openmath.

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

`gnuplot_pm3d` [Opción para plot]
Controla la utilización del modo PM3D, que tiene capacidades avanzadas para gráficos tridimensionales. PM3D sólo está disponible en versiones de Gnuplot posteriores a la 3.7. El valor por defecto de `gnuplot_pm3d` es `false`.

`gnuplot_preamble` [Opción para plot]
Introduce instrucciones de Gnuplot antes de que se haga el gráfico. Puede utilizarse cualquier comando válido de Gnuplot. Si interesa introducir varios comandos se separarán con punto y coma. El valor por defecto de `gnuplot_preamble` es la cadena vacía `""`.

`gnuplot_curve_titles` [Opción para plot]
Opción obsoleta que ha sido sustituida por `legend`.

`gnuplot_curve_styles` [Opción para plot]
Opción obsoleta que ha sido sustituida por `style`.

`gnuplot_default_term_command` [Opción para plot]
Comando de Gnuplot para establecer el tipo de terminal por defecto. El valor por defecto es `set term pop`.

`gnuplot_dumb_term_command` [Opción para plot]
Comando de Gnuplot para establecer el tipo de terminal para el terminal oculto. El valor por defecto es `"set term dumb 79 22"`, que da una salida de texto de 79 por 22 caracteres.

`gnuplot_ps_term_command` [Opción para plot]
Comando de Gnuplot para establecer el tipo de terminal para el terminal PostScript. El valor por defecto es `"set size 1.5, 1.5;set term postscript eps enhanced color solid 24"`, que establece un tamaño de 1.5 veces el valor por defecto de gnuplot, junto con un tamaño de fuente de 24, entre otras cosas. Consúltese la documentación de gnuplot para más información sobre `set term postscript`.

12.6 Funciones para el formato Gnuplot_pipes

`gnuplot_start` () [Función]
Inicializa una tubería hacia Gnuplot, con el fin de ser utilizada para utilizar el formato `gnuplot_pipes`. No es necesario inicializarla manualmente antes de hacer gráficos.

`gnuplot_close` () [Función]
Cierra la tubería hacia Gnuplot que haya sido utilizada para hacer gráficos.

`gnuplot_restart` () [Función]
Cierra la tubería hacia Gnuplot que haya sido utilizada para hacer gráficos e inicializa una nueva.

`gnuplot_replot` () [Función]
`gnuplot_replot` (*s*) [Función]
Actualiza la ventana de Gnuplot. Si `gnuplot_replot` es invocada con un comando de Gnuplot en la cadena *s*, entonces *s* es enviada a Gnuplot antes de redibujar la ventana.

`gnuplot_reset` () [Función]
Resetea Gnuplot cuando se utiliza el formato `gnuplot_pipes`. Para actualizar la ventana de Gnuplot invóquese a `gnuplot_replot` después de `gnuplot_reset`.

13 Lectura y escritura

13.1 Comentarios

En Maxima, un comentario es cualquier texto encerrado entre las marcas `/*` y `*/`.

El analizador sintáctico de Maxima trata los comentarios como espacios en blanco a efectos de encontrar *tokens* en el flujo de entrada. Una entrada tal como `a/* foo */b` contiene dos *tokens*, `a` y `b`, no un único *token* `ab`. En cualquier otro contexto, los comentarios son ignorados por Maxima; no se almacenan ni sus contenidos ni sus localizaciones.

Los comentarios pueden anidarse hasta una profundidad arbitraria. Las marcas `/*` y `*/` deben emparejarse y debe haber igual número de ambos.

Ejemplos:

```
(%i1) /* aa is a variable of interest */  aa : 1234;
(%o1)                          1234
(%i2) /* Value of bb depends on aa */  bb : aa^2;
(%o2)                         1522756
(%i3) /* User-defined infix operator */  infix ("b");
(%o3)                            b
(%i4) /* Parses same as a b c, not abc */  a/* foo */b/* bar */c;
(%o4)                          a b c
(%i5) /* Comments /* can be nested /* to any depth */ */ */  1 + xyz;
(%o5)                         xyz + 1
```

13.2 Archivos

Un archivo no es más que una área de un cierto dispositivo de almacenamiento que contiene datos o texto. Los archivos se agrupan en los discos en "directorios", que son listas de archivos. Instrucciones que operan con archivos son:

`appendfile`	`batch`	`batchload`
`closefile`	`file_output_append`	`filename_merge`
`file_search`	`file_search_maxima`	`file_search_lisp`
`file_search_demo`	`file_search_usage`	`file_search_tests`
`file_type`	`file_type_lisp`	`file_type_maxima`
`load`	`load_pathname`	`loadfile`
`loadprint`	`pathname_directory`	`pathname_name`
`pathname_type`	`printfile`	`save`
`stringout`	`with_stdout`	`writefile`

Cuando el nombre de un fichero se pasa a funciones como `plot2d`, `save` o `writefile` y en él no se incluye la ruta de acceso, Maxima almacena el fichero en la carpeta de trabajo actual. La ubicación de la carpeta de trabajo depende del sistema operativo y de la instalación.

13.3 Funciones y variables para lectura y escritura

`appendfile` (*filename*) [Función]

Añade información de la consola a *filename*, de igual manera que lo hace `writefile`, pero con la salvedad de que si el archivo ya existe la información queda añadida al final de su contenido.

La función `closefile` cierra los archivos abiertos por `appendfile` o `writefile`.

`batch` (*filename*) [Función]
`batch` (*filename*, `option`) [Function]

`batch(`*filename*`)` lee expresiones de Maxima desde *filename* y las evalúa. La función `batch` busca *filename* en la lista `file_search_maxima`. Véase `file_search`.

`batch(`*filename*`, test)` es como `run_testsuite` con la opción `display_all=true`. En este caso `batch` busca *filename* en la lista `file_search_maxima` y no en `file_search_tests` como hace `run_testsuite`. Además, `run_testsuite` ejecuta tests que están en la lista `testsuite_files`. Con `batch` es posible ejecutar cualquier fichero que se encuentre en `file_search_maxima` en modo de prueba.

El contenido de *filename* debe ser una secuencia de expresiones de Maxima, cada una de las cuales termina en `;` o `$`. La variable especial `%` y la función `%th` se refieren a resultados previos dentro del archivo. El archivo puede incluir construcciones del tipo `:lisp`. Espacios, tabulaciones y saltos de línea en el archivo se ignoran. Un archivo de entrada válido puede crearse con un editor de texto o con la función `stringout`.

La función `batch` lee las expresiones del archivo *filename*, muestra las entradas en la consola, realiza los cálculos solicitados y muestra las expresiones de los resultados. A las expresiones de entrada se les asignan etiquetas, así como a las de salida. La función `batch` evalúa todas las expresiones de entrada del archivo a menos que se produzca un error. Si se le solicita información al usuario (con `asksign` o `askinteger`, por ejemplo) `batch` se detiene para leer la nueva información para luego continuar.

Es posible detener `batch` tecleando `control-C` desde la consola. El efecto de `control-C` depende del entorno Lisp instalado.

La función `batch` tiene diversas aplicaciones, tales como servir de almacén de código escrito por el usuario, suministrar demostraciones libres de errores o ayudar a organizar el trabajo del usuario en la resolución de problemas complejos.

La función `batch` evalúa su argumento y devuelve la ruta hacia *filename* en formato cadena cuando es invocada sin segundo argumento o con la opción `demo`. Cuando es llamada con la opción `test`, devuelve la lista vacía `[]` o una lista con *filename* y los números de tests que han fallado.

Véanse también `load`, `batchload` y `demo`.

`batchload` (*filename*) [Función]

Lee expresiones de Maxima desde *filename* y las evalúa sin mostrar las entradas ni las salidas y sin asignarles etiquetas. Sin embargo, las salidas producidas por `print` o `describe` sí se muestran.

La variable especial `%` y la función `%th` se refieren a resultados previos del intérprete interactivo, no a los del propio archivo. El archivo no puede incluir construcciones del tipo `:lisp`.

La función `batchload` devuelve la ruta de *filename* en formato de cadena.

La función `batchload` evalúa sus argumentos.

Véanse también `batch` y `load`.

`closefile` () [Función]

La función `closefile` cierra los archivos abiertos por `appendfile` o `writefile`.

`file_output_append` [Variable opcional]

Valor por defecto: `false`

La variable `file_output_append` controla si las funciones de escritura de ficheros añaden información o sustituyen el fichero de salida. Cuando `file_output_append` toma el valor `true`, estas funciones amplían el contenido de sus ficheros de salida; en otro caso, sustituyen el fichero anterior de igual nombre por otro con el nuevo contenido.

Las funciones `save`, `stringout` y `with_stdout` se ven afectadas por el valor que tome la variable `file_output_append`. Otras funciones que también escriben en ficheros de salida no tienen en cuenta este valor; en concreto, las funciones para la representación de gráficos y las de traducción siempre sustituyen el fichero anterior por uno nuevo de igual nombre, mientras que las funciones `tex` y `appendfile` siempre añaden información al fichero de salida sin eliminar la información anterior.

`filename_merge` (*path*, *filename*) [Función]

Construye una ruta modificada a partir de *path* y *filename*. Si la componente final de *path* es de la forma `###.something`, la componente se reemplaza con `filename.something`. En otro caso, la componente final se reemplaza simplemente por *filename*.

El resultado es un objeto Lisp de tipo *pathname*.

`file_search` (*filename*) [Función]
`file_search` (*filename*, *pathlist*) [Función]

La función `file_search` busca el archivo *filename* y devuelve su ruta como una cadena; si no lo encuentra, `file_search` devuelve `false`. La llamada `file_search (filename)` busca en los directorios de búsqueda por defecto, que son los especificados por las variables `file_search_maxima`, `file_search_lisp` y `file_search_demo`.

La función `file_search` analiza primero si el nombre del argumento existe antes de hacerlo coincidir con los comodines de los patrones de búsqueda de archivos. Véase `file_search_maxima` para más información sobre patrones de búsqueda de archivos.

El argumento *filename* puede ser una ruta con nombre de archivo, o simplemente el nombre del archivo, o, si el directorio de búsqueda de archivo incluye un patrón de búsqueda, es suficiente con el nombre de archivo sin extensión. Por ejemplo,

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

todos buscan el mismo archivo, dando por hecho que el archivo existe y que `/home/wfs/special/###.mac` está en `file_search_maxima`.

La llamada `file_search (filename, pathlist)` busca solamente en los directorios especificados por *pathlist*, que es una lista de cadenas. El argumento *pathlist* ignora los directorios de búsqueda por defecto, de manera que si se da la lista de rutas, `file_search` busca solamente en ellas y no en los directorios por defecto. Incluso si hay un único directorio en *pathlist*, debe ser suministrado como una lista de un único elemento.

El usuario puede modificar los directorios de búsqueda por defecto; véase para ello See `file_search_maxima`.

La función `file_search` es llamada por `load` con los directorios de búsqueda `file_search_maxima` y `file_search_lisp`.

`file_search_maxima` [Variable opcional]
`file_search_lisp` [Variable opcional]
`file_search_demo` [Variable opcional]
`file_search_usage` [Variable opcional]
`file_search_tests` [Variable opcional]

Estas variables especifican listas de directorios en los que deben buscar la funciones `load`, `demo` y algunas otras. Los valores por defecto de estas variables nombran directorios de la instalación de Maxima.

El usuario puede modificar estas variables, bien reemplazando los valores por defecto, bien añadiendo nuevos directorios. Por ejemplo,

```
file_search_maxima: ["/usr/local/foo/###.mac",
    "/usr/local/bar/###.mac"]$
```

reemplaza el valor por defecto de `file_search_maxima`, mintras que

```
file_search_maxima: append (file_search_maxima,
    ["/usr/local/foo/###.mac", "/usr/local/bar/###.mac"])$
```

añade dos directorios más. Puede ser conveniente colocar una expresión como esta en el archivo `maxima-init.mac`, de manera que la ruta de búsqueda de ficheros se asigne automáticamente cada vez que arranca Maxima.

Se pueden especificar varias extensiones de archivos y rutas con comodines especiales. La cadena `###` representa el nombre del archivo buscado y una lista separada de comas y encerrada entre llaves, `{foo,bar,baz}` representa múltiples cadenas. Por ejemplo, suponiendo que se busca el nombre `neumann`,

```
"/home/{wfs,gcj}/###.{lisp,mac}"
```

se interpreta como `/home/wfs/neumann.lisp`, `/home/gcj/neumann.lisp`, `/home/wfs/neumann.mac` y `/home/gcj/neumann.mac`.

`file_type (filename)` [Función]

Devuelve una descripción del contenido de *filename* basada en la extensión, sin intentar abrir el archivo para inspeccionar su contenido.

El valor devuelto es un símbolo `object`, `lisp` o `maxima`. Si la extensión es "mac", "mc", "demo", "dem", "dm1", "dm2", "dm3" o "dmt", `file_type` devuelve `maxima`. Si la extensión es "l", "lsp" o "lisp", `file_type` devuelve `lisp`. Si la extensión no es ninguna de las anteriores, `file_type` devuelve `object`.

Véase también `pathname_type`.

Ejemplos:

```
(%i2) map('file_type,["test.lisp", "test.mac", "test.dem", "test.txt"]);
(%o2)            [lisp, maxima, maxima, object]
```

`file_type_lisp` [Variable opcional]

Valor por defecto: `[l, lsp, lisp]`

`file_type_lisp` es una lista con extensiones de ficheros que Maxima reconoce como fuente de Lisp.

Véase también `file_type`

`file_type_maxima` [Variable opcional]

Valor por defecto: `[mac, mc, demo, dem, dm1, dm2, dm3, dmt]`

`file_type_maxima` es una lista con extensiones de ficheros que Maxima reconoce como fuente de Maxima.

Véase también `file_type`

`load` (*`filename`*) [Función]

Evalúa las expresiones del archivo *filename*, trayendo variables, funciones y otros objetos a Maxima. Una asignación hecha previamente a una variable en Maxima será destruida por otra asignación que se le haga en *filename*. Para encontrar el fichero, `load` llama a `file_search` con `file_search_maxima` y `file_search_lisp` como directorios de búsqueda. Si la llamada a `load` funciona correctamente, devuelve el nombre del fichero; en caso contrario, `load` muestra un mensaje de error.

La función `load` trabaja indistintamente con código Lisp y Maxima. Los ficheros creados con `save`, `translate_file` y `compile_file`, que crea código Lisp, y `stringout`, que crea código Maxima, todos ellos pueden ser procesados por `load`. La función `load` llama a `loadfile` para cargar archivos en Lisp y a `batchload` para cargar archivos en Maxima.

La función `load` no reconoce las construcciones de tipo `:lisp` en ficheros de Maxima. Además, mientras se está procesando *filename*, las variables globales `_`, `__`, `%` y `%th` mantienen los valores que tenían cuando se realizó la llamada a `load`.

Véanse también `loadfile`, `batch`, `batchload` y `demo`; `loadfile` procesa archivos en Lisp; `batch`, `batchload` y `demo` procesan archivos en Maxima.

Véase `file_search` para más detalles sobre el mecanismo de búsqueda de archivos.

La función `load` evalúa sus argumentos.

`load_pathname` [Variable del sistema]

Valor por defecto: `false`

Cuando se carga un fichero con las funciones `load`, `loadfile` o `batchload`, a la variable `load_pathname` se le asigna la ruta al fichero en cuestión.

Se puede acceder a la variable `load_pathname` mientras se está cargando el fichero.

Ejemplo:

Supóngase que se tiene el fichero `test.mac` en la carpeta `"/home/usuario/workspace/mymaxima/temp/"` con las siguientes instrucciones:

```
print("The value of load_pathname is: ", load_pathname)$
```

```
print("End of batchfile")$
```

Entonces se obtiene el siguiente resultado:

```
(%i1) load("/home/usuario/workspace/mymaxima/temp/test.mac")$
The value of load_pathname is:
                    /home/usuario/workspace/mymaxima/temp/test.mac
End of batchfile
```

`loadfile` (*filename*) [Función]

Evalúa las expresiones Lisp del archivo *filename*. La función `loadfile` no llama a `file_search`, de manera que `filename` debe incluir la extensión del archivo y su ruta completa.

La función `loadfile` puede procesar ficheros creados por `save`, `translate_file` y `compile_file`. Puede ser más conveniente utilizar `load` en lugar de `loadfile`.

`loadprint` [Variable opcional]

Valor por defecto: `true`

La variable `loadprint` indica si mostrar un mensaje cuando se carga un archivo.

- Si `loadprint` vale `true`, se muestra siempre un mensaje.
- Si `loadprint` vale `'loadfile`, muestra un mensaje sólo si el archivo es cargado con la función `loadfile`.
- Si `loadprint` vale `'autoload`, muestra un mensaje sólo cuandi un archivo se carga automáticamente. Véase `setup_autoload`.
- Si `loadprint` vale `false`, nunca mostrará mensajes.

`packagefile` [Variable opcional]

Valor por defecto: `false`

Los desarrolladores de paquetes que utilizan `save` o `translate` para crear paquetes (ficheros) que van a ser utilizados por terceros pueden hacer `packagefile: true` para evitar que se añada información a la listas de información de Maxima, como `values` o `functions`.

`pathname_directory` (*pathname*) [Función]
`pathname_name` (*pathname*) [Función]
`pathname_type` (*pathname*) [Función]

Estas funciones devuelven las componentes de *pathname*.

Ejemplos:

```
(%i1) pathname_directory("/home/usuario/maxima/changelog.txt");
(%o1)                  /home/usuario/maxima/
(%i2) pathname_name("/home/usuario/maxima/changelog.txt");
(%o2)                         changelog
(%i3) pathname_type("/home/usuario/maxima/changelog.txt");
(%o3)                             txt
```

`printfile` (*path*) [Función]

Envía el fichero al que hace referncia la ruta *path* a la consola. *path* puede ser una cadena o un símbolo, en cuyo caso se convertirá en una cadena.

Si *path* hace referencia a un fichero accesible desde el directorio actual de trabajo, entonces se enviará a la consola; en caso contrario, `printfile` intentará localizar el fichero añadiéndole *path* a cada uno de los elementos de `file_search_usage` a través de `filename_merge`.

`printfile` devuelve la ruta del fichero encontado.

`save` (*filename*, *name_1*, *name_2*, *name_3*, ...) [Función]
`save` (*filename*, *values*, *functions*, *labels*, ...) [Función]
`save` (*filename*, [*m*, *n*]) [Función]
`save` (*filename*, *name_1*=*expr_1*, ...) [Función]
`save` (*filename*, *all*) [Función]
`save` (*filename*, *name_1*=*expr_1*, *name_2*=*expr_2*, ...) [Función]

Alamacena los valores actuales de *name_1*, *name_2*, *name_3*, ..., en el archivo *filename*. Los argumentos son nombres de variables, funciones u otros objetos. Si un nombre no tiene un valor o una función asociado a él, entonces se ignora.

La función `save` devuelve *filename*.

La función `save` almacena datos en forma de expresiones Lisp. Los datos almacenados por `save` pueden recuperarse con `load (`*filename*`)`. El resultado de ejecutar `save` cuando *filename* ya existe depende del soporte Lisp implementado; el archivo puede ser sobreescrito o que `save` envíe un mesaje de error.

La llamada `save (`*filename*`, values, functions, labels, ...)` almacena los elementos cuyos nombres son `values`, `functions`, `labels`, etc. Los nombres pueden ser cualesquiera de los especificados por la variable `infolists`; `values` incluye todas las variables definidas por el usuario.

La llamada `save (`*filename*`, [`*m*`, `*n*`])` almacena los valores de las etiquetas de entrada y salida desde *m* hasta *n*. Nótese que *m* y *n* deben ser números. Las etiquetas de entrada y salida también se pueden almacenar una a una, por ejemplo, `save ("foo.1", %i42, %o42)`. La llamada `save (`*filename*`, labels)` almacena todas las etiquetas de entrada y salida. Cuando las etiquetas almacenadas en el archivo sean posteriormente recuperadas, se sobreescribirán las activas en ese momento.

La llamada `save (`*filename*`, `*name_1*`=`*expr_1*`, `*name_2*`=`*expr_2*`, ...)` almacena los valores de *expr_1*, *expr_2*, ..., con los nombres *name_1*, *name_2*, ... Es útil hacer este tipo de llamada para con etiquetas de entrada y salida, por ejemplo, `save ("foo.1", aa=%o88)`. El miembro derecho de la igualdad puede ser cualquier expresión, que será evaluada. Esta llamada a la función `save` no incorpora nuevos nombres a la sesión actual de Maxima, simplemente los almacena en el archivo *filename*.

Todas estas formas de llamar a la función `save` se pueden combinar a voluntad. Por ejemplo, `save (`*filename*`, aa, bb, cc=42, functions, [11, 17])`.

La llamada `save (`*filename*`, all)` almacena el estado actual de Maxima, lo que incluye todas las variables definidas por el usuario, funciones, arreglos, etc., así como algunos objetos definidos automáticamente. Los elementos almacenados incluyen variables del sistema, como `file_search_maxima` o `showtime`, si han sido modificadas por el usuario. Véase `myoptions`.

`save` evalúa *filename* pero no el resto de argumentos.

`stringout` (*filename*, *expr_1*, *expr_2*, *expr_3*, ...) [Función]
`stringout` (*filename*, [*m*, *n*]) [Función]
`stringout` (*filename*, *input*) [Función]
`stringout` (*filename*, *functions*) [Función]
`stringout` (*filename*, *values*) [Función]

La función `stringout` escribe expresiones en un archivo de la misma forma en que se escribirían como expresiones de entrada. El archivo puede ser utilizado entonces como entrada a las funciones `batch` o `demo`, y puede ser editado para cualquier otro propósito.

La forma general de `stringout` escribe los valores de una o más expresiones en el archivo de salida. Nótese que si una expresión es una variable, solamente se escribirá el valor de la variable y no el nombre de ésta. Como caso especial, y muy útil en algunas ocasiones, las expresiones pueden ser etiquetas de entrada (`%i1`, `%i2`, `%i3`, ...) o de salida (`%o1`, `%o2`, `%o3`, ...).

Si `grind` vale `true`, `stringout` formatea la salida utilizando `grind`. En caso contrario, se utilizará el formato `string`. Véanse `grind` y `string`.

La forma especial `stringout (filename, [m, n])` escribe los valores de las etiquetas de entrada desde la m hasta la n, ambas inclusive.

La forma especial `stringout (filename, input)` escribe todas las etiquetas de entrada en el archivo.

La forma especial `stringout (filename, functions)` escribe todas las funciones definidas por el usuario, contenidas en la lista global `functions`, en el archivo.

La forma especial `stringout (filename, values)` escribe todas las variables asignadas por el usuario, contenidas en la lista global `values`, en el archivo. Cada variable se escribe como una sentencia de asignación, con el nombre de la variable seguida de dos puntos y a continuación su valor. Nótese que la forma general de `stringout` no escribe las variables como sentencias de asignación.

`with_stdout` (*f*, *expr_1*, *expr_2*, *expr_3*, ...) [Función]
`with_stdout` (*s*, *expr_1*, *expr_2*, *expr_3*, ...) [Función]

Evalúa *expr_1*, *expr_2*, *expr_3*, ... y escribe los resultados en el fichero *f* o flujo de salida *s*. Las expresiones que se evalúan no se escriben. La salida puede generarse por medio de `print`, `display`, `grind` entre otras funciones.

La variable global `file_output_append` controla si `with_stdout` añade o reinicia el contenido del fichero de salida *f*. Si `file_output_append` vale `true`, `with_stdout` añade contenido al fichero de salida. En cualquier caso, `with_stdout` crea el fichero si éste no existe.

La función `with_stdout` devuelve el valor de su último argumento.

Véase también `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*) [Función]

Comienza escribiendo una transcripción de la sesión de Maxima en el archivo *filename*. Cualquier interacción entre Maxima y el usuario se almacena también en este archivo, tal como aparece en la consola.

Puesto que la transcripción se escribe en el formato de salida a la consola, su contenido no es interpretable por Maxima. Para hacer un archivo que contenga expresiones que puedan ser nuevamente cargadas en Maxima, véanse `save` y `stringout`; la función `save` almacena expresiones en formato Lisp, mientras que `stringout` lo hace en formato Maxima.

El resultado de ejecutar `writefile` cuando el archivo *filename* ya existe depende del entorno Lisp operativo; el contenido anterior puede ser sobreescrito o ampliado con la sesión actual. La función `appendfile` siempre añade la sesión al contenido actual.

Puede ser útil ejecutar `playback` después de `writefile` para guardar las interacciones previas de la sesión. Puesto que `playback` muestra solamente las variables de entrada y salida (`%i1`, `%o1`, etc.), cualquier salida generada por una sentencia de impresión desde dentro de una función no es mostrada por `playback`.

La función `closefile` cierra los archivos abiertos por `writefile` o `appendfile`.

13.4 Funciones y variables para salida TeX

`tex` (*expr*) [Función]
`tex` (*expr*, *destination*) [Función]
`tex` (*expr*, *false*) [Función]
`tex` (*label*) [Función]
`tex` (*label*, *destination*) [Función]
`tex` (*label*, *false*) [Función]

Devuelve la expresión en un formato apropiado para para ser incorporado a un documento basado en TeX. El resultado que se obtiene es un fragmento de código que puede incluirse en un documento mayor, pero que no puede ser procesado aisladamente.

La instrucción `tex (`*expr*`)` imprime en la consola la representación en TeX de *expr*.

La instrucción `tex (`*label*`)` imprime en la consola la representación en TeX de la expresión a la que hace referencia la etiqueta *label*, asignándole a su vez una etiqueta de ecuación que será mostrada al lado izquierdo de la misma. La etiqueta de la expresión en TeX es la misma que la de Maxima.

destination puede ser tanto un flujo de salida como el nombre de un fichero.

Si *destination* es el nombre de un fichero, `tex` añade la salida al fichero. Las funciones `openw` y `opena` crean flujos de salida.

Las instrucciones `tex (`*expr*`, false)` y `tex (`*label*`, false)` devuelven el código TeX en formato de cadena.

La función `tex` evalúa su primer argumento tras comprobar si se trata de una etiqueta. La doble comilla simple `''` fuerza la evaluación del argumento, anulando la comprobación sobre la etiqueta.

Véase también `texput`.

Ejemplos:

```
(%i1) integrate (1/(1+x^3), x);
                                      2 x - 1
                    2            atan(-------)
                log(x  - x + 1)        sqrt(3)    log(x + 1)
(%o1)       - --------------- + ------------- + ----------
                      6             sqrt(3)          3
(%i2) tex (%o1);
$$-{{\log \left(x^2-x+1\right)}\over{6}}+{{\arctan \left({{2\,x-1
 }\over{\sqrt{3}}}\right)}\over{\sqrt{3}}}+{{\log \left(x+1\right)
 }\over{3}}\leqno{\tt (\%o1)}$$
(%o2)                          (\%o1)
(%i3) tex (integrate (sin(x), x));
$$-\cos x$$
(%o3)                           false
(%i4) tex (%o1, "foo.tex");
(%o4)                          (\%o1)
```

`tex (`*expr*`, false)` devuelve el código TeX en formato de cadena.

```
(%i1) S : tex (x * y * z, false);
(%o1) $$x\,y\,z$$
(%i2) S;
(%o2) $$x\,y\,z$$
```

`tex1 (`*e*`)` [Función]

Devuelve una cadena con el código TeX de la expresión *e*. El código TeX no se encierra entre delimitadores para una ecuación ni cualesquiera otros entornos.

Ejemplo:

```
(%i1) tex1 (sin(x) + cos(x));
(%o1)                     \sin x+\cos x
```

`texput (a, `*s*`)` [Función]
`texput (a, `*f*`)` [Función]
`texput (a, `*s*`, `*operator_type*`)` [Función]
`texput (a, [`*s_1*`, `*s_2*`], `*matchfix*`)` [Función]
`texput (a, [`*s_1*`, `*s_2*`, `*s_3*`], `*matchfix*`)` [Función]

Establece el formato en TeX del átomo *a*, el cual puede ser un símbolo o el nombre de un operador.

La instrucción `texput (a, `*s*`)` hace que la función `tex` introduzca *s* en la salida TeX en el lugar de *a*.

La instrucción `texput (a, `*f*`)` hace que `tex` llame a la función *f* para que genere código TeX. La función *f* debe aceptar un único argumento, el cual es una expresión que tenga como operador *a* y que devuelva una cadena con el código TeX. Esta función puede llamar a `tex1` para generar el código TeX para los argumentos de la expresión de entrada.

La instrucción `texput (a, s, operator_type)`, en la que *operator_type* es `prefix`, `infix` o `postfix`, `nary` o `nofix`,hace que la función `tex` introduzca *s* en la salida TeX en el lugar de *a*, colocándolo en el lugar correcto.

La instrucción `texput (a, [s_1, s_2], matchfix)` hace que la función `tex` introduzca *s_1* y *s_2* en la salida TeX a los lados de los argumentos de *a*. Si son más de uno, los argumentos se separan por comas.

La instrucción `texput (a, [s_1, s_2, s_3], matchfix)` hace que la función `tex` introduzca *s_1* y *s_2* en la salida TeX a los lados de los argumentos de *a*, con *s_3* separando los argumentos.

Ejemplos:

Asigna código TeX para una variable.

Llama a una función que genera código TeX.

```
(%i1) texfoo (e) := block ([a, b], [a, b] : args (e),
  concat ("\\left[\\stackrel{", tex1 (b),
          "}{", tex1 (a), "}\\right]"))$
(%i2) texput (foo, texfoo);
(%o2)                          texfoo
(%i3) tex (foo (2^x, %pi));
$$\left[\stackrel{\pi}{2^{x}}\right]$$
(%o3)                           false
(%i1) texput (me,"\\mu_e");
(%o1)                           \mu_e
(%i2) tex (me);
$$\mu_e$$
(%o2)                           false
```

Asigna código TeX para una función ordinaria (no para un operador).

```
(%i1) texput (lcm, "\\mathrm{lcm}");
(%o1)                       \mathrm{lcm}
(%i2) tex (lcm (a, b));
$$\mathrm{lcm}\left(a , b\right)$$
(%o2)                           false
```

Asigna código TeX para un operador prefijo.

```
(%i1) prefix ("grad");
(%o1)                           grad
(%i2) texput ("grad", " \\nabla ", prefix);
(%o2)                          \nabla
(%i3) tex (grad f);
$$ \nabla f$$
(%o3)                           false
```

Asigna código TeX para un operador infijo.

```
(%i1) infix ("~");
(%o1)                             ~
(%i2) texput ("~", " \\times ", infix);
(%o2)                          \times
```

```
(%i3) tex (a ~ b);
$$a \times b$$
(%o3)                          false
```

Asigna código TeX para un operador postfijo..

```
(%i1) postfix ("##");
(%o1)                           ##
(%i2) texput ("##", "!!", postfix);
(%o2)                           !!
(%i3) tex (x ##);
$$x!!$$
(%o3)                          false
```

Asigna código TeX para un operador n-ario.

```
(%i1) nary ("@@");
(%o1)                           @@
(%i2) texput ("@@", " \\circ ", nary);
(%o2)                          \circ
(%i3) tex (a @@ b @@ c @@ d);
$$a \circ b \circ c \circ d$$
(%o3)                          false
```

Asigna código TeX para un operador "no-fijo".

```
(%i1) nofix ("foo");
(%o1)                           foo
(%i2) texput ("foo", "\\mathsc{foo}", nofix);
(%o2)                      \mathsc{foo}
(%i3) tex (foo);
$$\mathsc{foo}$$
(%o3)                          false
```

Asigna código TeX para un operador "bi-fijo" (matchfix).

```
(%i1) matchfix ("<<", ">>");
(%o1)                           <<
(%i2) texput ("<<", [" \\langle ", " \\rangle "], matchfix);
(%o2)                 [ \langle ,  \rangle ]
(%i3) tex (<<a>>);
$$ \langle a \rangle $$
(%o3)                          false
(%i4) tex (<<a, b>>);
$$ \langle a , b \rangle $$
(%o4)                          false
(%i5) texput ("<<", [" \\langle ", " \\rangle ", " \\, | \\,"],
      matchfix);
(%o5)           [ \langle ,  \rangle ,  \, | \,]
(%i6) tex (<<a>>);
$$ \langle a \rangle $$
(%o6)                          false
(%i7) tex (<<a, b>>);
```

```
$$ \langle a \, | \,b \rangle $$
(%o7)                          false
```

`get_tex_environment` (*op*) [Función]
`set_tex_environment` (*op*, *before*, *after*) [Función]

Gestiona el entorno de las salidas TeX que se obtienen de la función `tex`. El entorno TeX está formado por dos cadenas: una que se escribe antes que cualquier salida en TeX, y otra que se escribe después.

`get_tex_environment` devuelve el entorno TeX que se aplica al operador *op*. Si no se ha asignado ningún entorno, devolverá el que tenga por defecto.

`set_tex_environment` asigna el entorno TeX al operador *op*.

Ejemplos:

```
(%i1) get_tex_environment (":=");
(%o1) [
\begin{verbatim}
, ;
\end{verbatim}
]
(%i2) tex (f (x) := 1 - x);

\begin{verbatim}
f(x):=1-x;
\end{verbatim}

(%o2)                          false
(%i3) set_tex_environment (":=", "$$", "$$");
(%o3)                        [$$, $$]
(%i4) tex (f (x) := 1 - x);
$$f(x):=1-x$$
(%o4)                          false
```

`get_tex_environment_default` () [Función]
`set_tex_environment_default` (*before*, *after*) [Función]

Gestiona el entorno de las salidas TeX que se obtienen de la función `tex`. El entorno TeX está formado por dos cadenas: una que se escribe antes que cualquier salida en TeX, y otra que se escribe después.

`get_tex_environment_default` devuelve el entorno TeX que se aplica a expresiones para las cuales el operador de mayor rango no tiene entorno TeX asignado (mediante `set_tex_environment`).

`set_tex_environment_default` asigna el entorno TeX por defecto.

Ejemplos:

```
(%i1) get_tex_environment_default ();
(%o1)                        [$$, $$]
(%i2) tex (f(x) + g(x));
$$g\left(x\right)+f\left(x\right)$$
(%o2)                          false
```

```
(%i3) set_tex_environment_default ("\\begin{equation}
", "
\\end{equation}");
(%o3) [\begin{equation}
,
\end{equation}]
(%i4) tex (f(x) + g(x));
\begin{equation}
g\left(x\right)+f\left(x\right)
\end{equation}
(%o4)                          false
```

13.5 Funciones y variables para salida Fortran

`fortindent` [Variable opcional]

Valor por defecto: 0

La variable `fortindent` controla el margen izquierdo de las expresiones que escribe la instrucción `fortran`. El valor 0 escribe con un margen normal de 6 espacios; valores positivos harán que las expresiones se escriban más a la derecha.

`fortran` (*expr*) [Función]

Escribe *expr* en código Fortran. La salida se escribe con márgenes, y si ésta es demasiado larga `fortran` sigue escribiendo en líneas sucesivas. La función `fortran` escribe el operador de exponenciación `^` como `**`, e imprime un número complejo `a + b %i` como `(a,b)`.

El argumento *expr* puede ser una ecuación. En tal caso, `fortran` escribe una sentencia de asignación, dándole el valor del miembro derecho de la expresión al miembro izquierdo. En particular, si el miembro derecho de *expr* es el nombre de una matriz, entonces `fortran` escribe una sentencia de asignación para cada elemento de la matriz.

Si *expr* no es reconozida por `fortran`, la expresión se escribe en formato `grind` sin avisos. La función `fortran` no reconoce listas, arreglos ni funciones.

La variable `fortindent` controla el margen izquierdo de las expresiones que escribe la instrucción `fortran`. El valor 0 escribe con un margen normal de 6 espacios; valores positivos harán que las expresiones se escriban más a la derecha.

Si `fortspaces` vale `true`, `fortran` rellena las líneas con espacios de 80 columnas.

La función `fortran` evalúa sus argumentos; un argumento precedido de apóstrofo previene de la evaluación. La función `fortran` siempre devuelve `done`.

Ejemplos:

```
(%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` [Variable opcional]

Valor por defecto: `false`

Si `fortspaces` vale `true`, `fortran` rellena las líneas con espacios de 80 columnas.

14 Polinomios

14.1 Introducción a los polinomios

Los polinomios se almacenan en Maxima, bien en un formato general, bien en una forma conocida como canónica (Cannonical Rational Expressions, CRE). La última corresponde al formato estándar y se utiliza internamente para realizar operaciones como `factor`, `ratsimp` y demás.

Las Expresiones Racionales Canónicas (CRE) constituyen un tipo de representación que es especialmente apropiado para expandir polinomios y funciones racionales (así como para polinomios parcialmente factorizados y funciones racionales cuando a la variable `ratfac` se le asigna el valor `true`). En esta forma CRE las variables se ordenan de mayor a menor. Los polinomios se representan recursivamente como una lista compuesta por la variable principal seguida por una serie de pares de expresiones, una por cada término del polinomio. El primer miembro de cada par es el exponente de la variable principal en ese término y el segundo miembro es el coeficiente de ese término, el cual puede ser un número o un polinomio en otra variable representado también de esta forma. Así, la parte principal de la forma CRE de 3*X^2-1 es (X 2 3 0 -1) y la de 2*X*Y+X-3 es (Y 1 (X 1 2) 0 (X 1 1 0 -3)) asumiendo que Y es la variable principal, y será (X 1 (Y 1 2 0 1) 0 -3) si se asume que la variable principal es X. Qué variable se considera "principal" se determinada en orden alfabético inverso. Las "variables" de la expresión CRE no son necesariamente atómicas. De hecho cualquier subexpresión cuyo operador principal no es + - * / ni ^ con potencia entera puede ser considerada como una "variable" de la expresión (en forma CRE) en el cual aparezca. Por ejemplo las variables CRE de la expresión X+SIN(X+1)+2*SQRT(X)+1 son X, SQRT(X) y SIN(X+1). Si el usuario no especifica una ordenación de las variables mediante la función `ratvars` Maxima escogerá una alfabéticamente. En general, laa CRE representan expresiones racionales, esto es, fracciones de polinomios, donde el numerador y el denominador no tienen factores comunes, siendo el denominador es positivo. La forma interna es esencialmente un par de polinomios (el numerador y el denominador) precedida por la lista de variables ordenadas. Si una expresión a ser mostrada está en la forma CRE o contiene alguna subexpresión en forma de CRE, el simbolo /R/ será seguido por la etiqueta de la línea de comando. Véase la función `rat` para convertir una expresión a la forma CRE. Una extensión de la forma CRE se utiliza para la representación de las series de Taylor. La noción de una expresión racional se extiende de manera que los exponentes de las variables pueden ser números racionales positivos o negativos y no sólo enteros positivos y los coeficientes pueden ser también expresiones racionales y no sólo polinomios. Estas expresiones se representan internamente por una forma polinomial recursiva que es similar a la forma CRE, pero que la generaliza, aportando información adicional como el grado de truncamiento. Como con la forma CRE, el símbolo /T/ sigue la etiqueta de línea de comando en la que se encuentra dicha expresión.

14.2 Funciones y variables para polinomios

`algebraic` [Variable opcional]

Valor por defecto: `false`

La variable `algebraic` debe valer `true` para que se pueda hacer la simplificación de enteros algebraicos.

`berlefact` [Variable opcional]

Valor por defecto: `true`

Si `berlefact` vale `false` entonces se utiliza el algoritmo de factorización de Kronecker, en caso contrario se utilizará el algoritmo de Berlekamp, que es el que se aplica por defecto.

`bezout` (*p1*, *p2*, *x*) [Función]

Es una alternativa a la función `resultant`. Devuelve una matriz.

```
(%i1) bezout(a*x+b, c*x^2+d, x);
                           [ b c  - a d ]
(%o1)                      [            ]
                           [  a     b   ]
(%i2) determinant(%);
                             2      2
(%o2)                      a  d + b  c
(%i3) resultant(a*x+b, c*x^2+d, x);
                             2      2
(%o3)                      a  d + b  c
```

`bothcoef` (*expr*, *x*) [Función]

Devuelve una lista cuyo primer miembro es el coeficiente de *x* en *expr* (que coincide con el que devuelve `ratcoef` si *expr* está en formato CRE, o el que devuelve `coeff` si no está en este formato) y cuyo segundo miembro es la parte restante de *expr*. Esto es, `[A, B]` donde *expr* = `A*`*x* + `B`.

Ejemplo:

```
(%i1) islinear (expr, x) := block ([c],
        c: bothcoef (rat (expr, x), x),
        is (freeof (x, c) and c[1] # 0))$
(%i2) islinear ((r^2 - (x - r)^2)/x, x);
(%o2)                          true
```

`coeff` (*expr*, *x*, *n*) [Función]

`coeff` (*expr*, *x*) [Función]

Devuelve el coeficiente de *x*`^`*n* en *expr*, donde *expr* es un polinomio o monomio en *x*.

`coeff(`*expr*`,` *x*`^`*n*`)` es equivalente a `coeff(`*expr*`,` *x*`,` *n*`)`. `coeff(`*expr*`,` *x*`, 0)` devuelve el resto de *expr*, el cual no contiene a *x*. En caso de omisión, se entiende que *n* es igual a 1.

x puede ser tanto el nombre de una variable simple como el de una variable con subíndice, o también una subexpresión de *expr* que contenga un operador junto con todos sus argumentos.

En ocasiones, es posible calcular los coeficientes de expresiones equivalentes a *expr* aplicando `expand` o `factor`. `coeff` no aplica ni `expand`, ni `factor`, ni ninguna otra función.

`coeff` se distribuye sobre listas, matrices y ecuaciones.

Ejemplos:

`coeff` devuelve el coeficiente de *x^n* en *expr*.

```
(%i1) coeff (b^3*a^3 + b^2*a^2 + b*a + 1, a^3);
                                3
(%o1)                           b
```

`coeff(`*expr*`, `*x^n*`)` es equivalente a `coeff(`*expr*`, `*x*`, `*n*`)`.

```
(%i1) coeff (c[4]*z^4 - c[3]*z^3 - c[2]*z^2 + c[1]*z, z, 3);
(%o1)                          - c
                                 3
(%i2) coeff (c[4]*z^4 - c[3]*z^3 - c[2]*z^2 + c[1]*z, z^3);
(%o2)                          - c
                                 3
```

`coeff(`*expr*`, `*x*`, 0)` devuelve el resto de *expr*, el cual no contiene a *x*.

```
(%i1) coeff (a*u + b^2*u^2 + c^3*u^3, b, 0);
                             3  3
(%o1)                       c  u  + a u
```

x puede ser tanto el nombre de una variable simple como el de una variable con subíndice, o también una subexpresión de *expr* que contenga un operador junto con todos sus argumentos.

```
(%i1) coeff (h^4 - 2*%pi*h^2 + 1, h, 2);
(%o1)                          - 2 %pi
(%i2) coeff (v[1]^4 - 2*%pi*v[1]^2 + 1, v[1], 2);
(%o2)                          - 2 %pi
(%i3) coeff (sin(1 + x)*sin(x) + sin(1 + x)^3*sin(x)^3, sin(1 + x)^3);
                                3
(%o3)                          sin (x)
(%i4) coeff ((d - a)^2*(b + c)^3 + (a + b)^4*(c - d), a + b, 4);
(%o4)                           c - d
```

`coeff` no aplica ni `expand`, ni `factor`, ni ninguna otra función.

```
(%i1) coeff (c*(a + b)^3, a);
(%o1)                           0
(%i2) expand (c*(a + b)^3);
                3          2        2        3
(%o2)          b  c + 3 a b  c + 3 a  b c + a  c
(%i3) coeff (%, a);
                                2
(%o3)                          3 b  c
(%i4) coeff (b^3*c + 3*a*b^2*c + 3*a^2*b*c + a^3*c, (a + b)^3);
(%o4)                           0
(%i5) factor (b^3*c + 3*a*b^2*c + 3*a^2*b*c + a^3*c);
                                  3
(%o5)                        (b + a)  c
(%i6) coeff (%, (a + b)^3);
```

```
(%o6)                              c
```

`coeff` se distribuye sobre listas, matrices y ecuaciones.

```
(%i1) coeff ([4*a, -3*a, 2*a], a);
(%o1)                        [4, - 3, 2]
(%i2) coeff (matrix ([a*x, b*x], [-c*x, -d*x]), x);
                             [  a    b  ]
(%o2)                        [          ]
                             [ - c  - d ]
(%i3) coeff (a*u - b*v = 7*u + 3*v, u);
(%o3)                            a = 7
```

`content` (*p_1*, *x_1*, ..., *x_n*) [Función]

Devuelve una lista cuyo primer miembro es el máximo común divisor de los coeficientes de los términos del polinomio *p_1* de variable *x_n* (este es el contenido) y cuyo segundo miembro es el polinomio *p_1* dividido por el contenido.

Ejemplos:

```
(%i1) content (2*x*y + 4*x^2*y^2, y);
                                      2
(%o1)                    [2 x, 2 x y  + y]
```

`denom` (*expr*) [Función]

Devuelve el denominador de la expresión racional *expr*.

`divide` (*p_1*, *p_2*, *x_1*, ..., *x_n*) [Función]

Calcula el cociente y el resto del polinomio *p_1* dividido por el polinomio *p_2*, siendo la variable principal *x_n*. Las otras funciones son como en la función `ratvars`. El resultado es una lista cuyo primer miembro es el cociente y el segundo miembro el resto.

Ejemplos:

```
(%i1) divide (x + y, x - y, x);
(%o1)                         [1, 2 y]
(%i2) divide (x + y, x - y);
(%o2)                        [- 1, 2 x]
```

Nótese que `y` es la variable principal en el segundo ejemplo.

`eliminate` ([*eqn_1*, ..., *eqn_n*], [*x_1*, ..., *x_k*]) [Función]

Elimina variables de ecuaciones (o de expresiones que se supone valen cero) tomando resultantes sucesivas. Devuelve una lista con *n* - *k* expresiones y *k* variables *x_1*, ..., *x_k* eliminadas. Primero se elimina *x_1* dando *n* - 1 expresiones, después se elimina `x_2`, etc. Si *k* = *n* entonces se devuelve una lista con una única expresión, libre de las variables *x_1*, ..., *x_k*. En este caso se llama a `solve` para resolver la última resultante para la última variable.

Ejemplo:

```
(%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*, ...) [Función]

Devuelve una lista cuyo primer elemento es el máximo común divisor (mcd) de los polinomios *p_1*, *p_2*, *p_3*, ..., siendo los miembros restantes los mismos polinomios divididos por el mcd. Se utiliza siempre el algoritmo `ezgcd`.

Véanse también `gcd`, `gcdex`, `gcdivide` y `poly_gcd`.

Ejemplos:

Los tres polinomios tiene como máximo común divisor `2*x-3`, el cual se calcula primero con la función `gcd` y luego con `ezgcd`.

```
(%i1) p1 : 6*x^3-17*x^2+14*x-3;
                         3       2
(%o1)                 6 x  - 17 x  + 14 x - 3
(%i2) p2 : 4*x^4-14*x^3+12*x^2+2*x-3;
                       4       3       2
(%o2)             4 x  - 14 x  + 12 x  + 2 x - 3
(%i3) p3 : -8*x^3+14*x^2-x-3;
                          3       2
(%o3)                - 8 x  + 14 x  - x - 3

(%i4) gcd(p1, gcd(p2, p3));
(%o4)                         2 x - 3

(%i5) ezgcd(p1, p2, p3);
                     2               3      2           2
(%o5) [2 x - 3, 3 x  - 4 x + 1, 2 x  - 4 x  + 1, - 4 x  + x + 1]
```

`facexpand` [Variable opcional]

Valor por defecto: `true`

La variable `facexpand` controla si los factores irreducibles devueltos por `factor` están en formato expandido (por defecto) o recursivo (CRE normal).

`factor` (*expr*) [Función]

`factor` (*expr*, *p*) [Función]

Factoriza la expresión *expr*, que puede contener cualquier número de variables o funciones, en factores irreducibles respecto de los enteros. La llamada `factor (expr, p)`

factoriza *expr* en el campo de los racionales con un elemento añadido cuyo polinomio mínimo es *p*.

La función `factor` utiliza a `ifactors` para factorizar enteros.

Si la variable `factorflag` vale `false` suprime la factorización de los factores enteros en las expresiones racionales.

La variable `dontfactor` puede contener una lista de variables con respecto a las cuales no se factorizará (inicialmente está vacía). Tampoco se factorizará respecto de cualesquiera otra variables que sean menos importantes (según la ordenación que se sigue en el formato CRE) que aquellas que se encuentran en la lista `dontfactor`.

Si la variable `savefactors` vale `true`, los factores de una expresión en forma de producto se guardarán por ciertas funciones a fin de acelerar posteriores factorizaciones de expresiones que contengan algunos de estos mismos factores.

Si `berlefact` vale `false` entonces se utiliza el algoritmo de factorización de Kronecker, en caso contrario se utilizará el algoritmo de Berlekamp, que es el que se aplica por defecto.

Si la variable `intfaclim` vale `true`, Maxima desistirá de factorizar enteros si no encuentra ningún factor después de las divisiones tentativas y de aplicar el método rho de Pollard. Si vale `false` (este es el caso cuando el usuario invoca explícitamente a `factor`), se intentará la factorización completa del entero. El valor asignado a `intfaclim` se utiliza en llamadas internas a `factor`. Así, se puede cambiar el valor de `intfaclim` para evitar que Maxima dedique un tiempo prohibitivo a factorizar números enteros grandes.

Ejemplos:

```
(%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` [Variable opcional]

Valor por defecto: `false`

Si `factorflag` vale `false` se evita la factorización de factores enteros de expresiones racionales.

`factorout` (*expr*, *x_1*, *x_2*, ...) [Función]

Reorganiza la suma *expr* como una suma de términos de la forma `f (x_1, x_2, ...)*g`, donde `g` es un producto de expresiones que no contienen ningún *x_i* y `f` se factoriza.

Nótese que `factorout` ignora la variable opcional `keepfloat`.

Ejemplo:

```
(%i1) expand (a*(x+1)*(x-1)*(u+1)^2);
             2  2          2      2      2
(%o1)     a u  x  + 2 a u x  + a x  - a u  - 2 a u - a
(%i2) factorout(%,x);
          2
(%o2) a u  (x - 1) (x + 1) + 2 a u (x - 1) (x + 1)
                                              + a (x - 1) (x + 1)
```

`factorsum` (*expr*) [Función]

Intenta agrupar términos en los factores de *expr* que son sumas en grupos de términos tales que su suma sea factorizable. La función `factorsum` puede restablecer el recuperar de `expand ((x + y)^2 + (z + w)^2)` pero no puede recuperar `expand ((x + 1)^2 + (x + y)^2)` porque los términos tienen variables comunes.

Ejemplo:

```
(%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*) [Función]

Calcula el producto de los polinomios *p_1* y *p_2* utilizando un algoritmo especial. Los polinomios `p_1` y `p_2` deben ser multivariantes, densos y aproximadamente del mismo tamaño. La multiplicación clásica es de orden `n_1 n_2` donde `n_1` es el grado de `p_1` y `n_2` el grado de `p_2`. La función `fasttimes` es de orden `max (n_1, n_2)^1.585`.

`fullratsimp` (*expr*) [Función]

Aplica repetidamente `ratsimp` a una expresión, seguida de simplificaciones no racionales, hasta que no se obtienen más transformaciones; entonces devuelve el resultado.

En presencia de expresiones no racionales, una llamada a `ratsimp` seguida de una simplificación no racional (`"general"`) puede no ser suficiente para conseguir un resultado simplificado. En ocasiones serán necesarias más de una llamada a `ratsimp`, que es lo que hace precisamente `fullratsimp`.

Ejemplo:

```
(%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*) [Función]

Similar a `ratsubst` excepto por el hecho de que se llama a í misma recursivamente hasta que el resultado deja de cambiar. Esta función es útil cuando la expresión a sustituir y la que la sustituye tienen variables comunes.

La función `fullratsubst` también acepta sus argumentos en el formato de `lratsubst`.

Es necesario ejecutar `load ("lrats")` para cargar `fullratsubst` y `lratsubst`.

Ejemplos:

```
(%i1) load ("lrats")$
```

- `subst` puede hacer sustituciones múltiples; `lratsubst` es análoga 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
```

- Si sólo se quiere una sustitución, entonces se puede dar una única ecuación como primer argumento.

```
(%i4) lratsubst (a^2 = b, a^3);
(%o4)                           a b
```

- `fullratsubst` equivale a `ratsubst`, excepto por el hecho de que se llama a í misma recursivamente hasta que el resultado deja de cambiar.

```
(%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` también acepta una lista de ecuaciones o una sóla ecuación como primer 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` puede caer en una recursión infinita.

```
(%i9) errcatch (fullratsubst (b*a^2, a^2, a^3));

*** - Lisp stack overflow. RESET
```

`gcd` (*p_1*, *p_2*, *x_1*, ...) [Función]

Devuelve el máximo común divisor de *p_1* y *p_2*. La variable `gcd` determiona qué algoritmo se va a utilizar. Asignándole a `gcd` los valores `ez`, `subres`, `red` o `spmod`, se seleccionan los algoritmos `ezgcd`, subresultante `prs`, reducido o modular, respectivamente. Si `gcd` vale `false` entonces `gcd(p_1, p_2, x)` devolverá siempre 1 para cualquier *x*. Muchas funciones (por ejemplo, `ratsimp`, `factor`, etc.) hacen uso de `gcd`

implícitamente. En caso de polinomios homogéneos se recomienda darle a `gcd` el valor `subres`. Para calcular un máximo común divisor en presencia de raíces, como en `gcd (x^2 - 2*sqrt(2)*x + 2, x - sqrt(2))`, la variable `algebraic` debe igualarse a `true` y `gcd` no puede ser `ez`.

Se recomienda utilizar el algoritmo `subres` en lugar de `red`, por ser aquél más moderno.

Si la variable `gcd`, cuyo valor por defecto es `spmod`, vale `false`, no se calculará el máximo común divisor cuando las expresiones se conviertan a su forma canónica (CRE), lo que redundará en ocasiones en mayor rapidez de cálculo.

`gcdex` (*f*, *g*) [Función]
`gcdex` (*f*, *g*, *x*) [Función]

Devuelve una lista `[a, b, u]` en la que *u* es el máximo común divisor (mcd) de *f* y *g*, e igual a `a f + b g`. Los argumentos *f* y *g* deben ser polinomios univariantes, o indicarles la variable principal *x* en caso de ser multivariantes.

La función `gcdex` implementa el algoritmo de Euclides, en el que tenemos una secuencia de `L[i]: [a[i], b[i], r[i]]` todos ellos ortogonales a `[f, g, -1]` siendo el siguiente calculado a partir de `q = quotient(r[i]/r[i+1])` y `L[i+2]: L[i] - q L[i+1]`; el proceso termina en `L[i+1]` cuando el resto `r[i+2]` se anula.

```
(%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
```

`gcfactor` (*n*) [Función]

Factoriza el entero gaussiano *n* como producto, a su vez, de enteros gaussianos, (un entero gaussiano es de la forma`a + b %i` donde *a* y *b* son números enteros). Los factores se normalizan de manera que tanto la parte real como imaginaria sean no negativas.

`gfactor` (*expr*) [Función]

Factoriza el polinomio *expr* sobre los enteros gaussianos (un entero gaussiano es de la forma`a + b %i` donde *a* y *b* son números enteros). Es como `factor (expr, a^2+1)` donde *a* vale `%i`.

Ejemplo:

```
(%i1) gfactor (x^4 - 1);
(%o1)           (x - 1) (x + 1) (x - %i) (x + %i)
```

`gfactorsum` (*expr*) [Función]

Esta función es similar a `factorsum` pero aplica `gfactor` en lugar de `factor`.

`hipow` (*expr*, *x*) [Función]

Devuelve el mayor exponente explícito de *x* en *expr*. El argumento*x* puede ser una variable o una expresión general. Si *x* no aparece en *expr*, `hipow` devuelve `0`.

La función `hipow` no tiene en cuenta expresiones equivalentes a `expr`. En particular, `hipow` no expande `expr`, de manera que `hipow (`*`expr`*`, `*`x`*`)` y `hipow (expand (`*`expr`*`, `*`x`*`))` pueden dar resultados diferentes.

Ejemplos:

```
(%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` [Variable opcional]

Valor por defecto: `true`

Si vale `true`, Maxima desistirá de factorizar enteros si no encuentra ningún factor después de las divisiones tentativas y de aplicar el método rho de Pollard, por lo que la factorización puede quedar incompleta.

Si vale `false` (este es el caso cuando el usuario invoca explícitamente a `factor`), se intentará la factorización completa del entero. El valor asignado a `intfaclim` se utiliza en llamadas internas a `factor`. A la variable `intfaclim` se le asigna el valor `false` cuando se calculan factores desde las funciones `divisors`, `divsum` y `totient`.

Las llamadas internas a `factor` respetan el valor dado por el usuario a `intfaclim`. Asignando a `intfaclim` el valor `true` se puede reducir el tiempo que Maxima dedica a factorizar enteros grandes.

`keepfloat` [Variable opcional]

Valor por defecto: `false`

Si `keepfloat` vale `true`, los números decimales en coma flotante no se racionalizan cuando las expresiones que los contienen se convierten al formato canónico racional (CRE).

Nótese que la función `solve` y todas aquellas otras que la invocan (por ejemplo, `eigenvalues`) ignoran esta variable, por lo que hacen la conversión de los números decimales.

Ejemplos:

```
(%i1) rat(x/2.0);

'rat' replaced 0.5 by 1/2 = 0.5
                                        x
(%o1)/R/                                -
                                        2
(%i2) rat(x/2.0), keepfloat;
```

```
(%o2)/R/                                0.5 x
```

`solve` ignora `keepfloat`:

```
(%i3) solve(1.0-x,x), keepfloat;

'rat' replaced 1.0 by 1/1 = 1.0
(%o3)                                [x = 1]
```

`lopow` (*expr*, *x*) [Función]

Devuelve el menor exponente de *x* que aparece explícitamente en *expr*.

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

`lratsubst` (*L*, *expr*) [Función]

Esta función es similar a `subst` (*L*, *expr*), excepto por el hecho de que utiliza `ratsubst` en lugar de `subst`.

El primer argumento de `lratsubst` es una ecuación o lista de ecuaciones idénticas en formato a las aceptadas por `subst`. Las sustituciones se hacen en el orden dado por la lista de ecuaciones, esto es, de izquierda a derecha.

La instrucción `load ("lrats")` carga `fullratsubst` y `lratsubst`.

Ejemplos:

```
(%i1) load ("lrats")$
```

- `subst` can carry out multiple substitutions. `lratsubst` is analogous to `subst`.

```
(%i2) subst ([a = b, c = d], a + c);
(%o2)                           d + b
(%i3) lratsubst ([a^2 = b, c^2 = d], (a + e)*c*(a + c));
(%o3)                (d + a c) e + a d + b c
```

- If only one substitution is desired, then a single equation may be given as first argument.

```
(%i4) lratsubst (a^2 = b, a^3);
(%o4)                            a b
```

`modulus` [Variable opcional]

Valor por defecto: `false`

Si `modulus` es un número positivo *p*, las operaciones con números racionales (como los devueltos por `rat` y funciones relacionadas) se realizan módulo *p*, utilizando el llamado sistema de módulo balanceado, en el que *n* `módulo` *p* se define como un entero *k* de `[-(`*p*`-1)/2, ..., 0, ..., (`*p*`-1)/2]` si *p* es impar, o de `[-(`*p*`/2 - 1), ..., 0, ...,` *p*`/2]` si *p* es par, de tal manera que `a` *p* `+` *k* es igual a *n* para algún entero *a*.

Normalmente a `modulus` se le asigna un número primo. Se acepta que a `modulus` se le asigne un entero positivo no primo, pero se obtendrá un mensaje de aviso. Maxima responderá con un mensaje de error cuando se le asigne a `modulus` cero o un número negativo.

Ejemplos:

```
(%i1) modulus:7;
```

```
(%o1)                           7
(%i2) polymod([0,1,2,3,4,5,6,7]);
(%o2)            [0, 1, 2, 3, - 3, - 2, - 1, 0]
(%i3) modulus:false;
(%o3)                         false
(%i4) poly:x^6+x^2+1;
                               6    2
(%o4)                         x  + x  + 1
(%i5) factor(poly);
                               6    2
(%o5)                         x  + x  + 1
(%i6) modulus:13;
(%o6)                            13
(%i7) factor(poly);
                         2        4      2
(%o7)                 (x  + 6) (x  - 6 x  - 2)
(%i8) polymod(%);
                               6    2
(%o8)                         x  + x  + 1
```

num (*expr*) [Función]

Devuelve el numerador de *expr* si se trata de una fracción. Si *expr* no es una fracción, se devuelve *expr*.

La función `num` evalúa su argumento.

polydecomp (*p*, *x*) [Función]

Descompone el polinomio *p* de variable *x* en una composición funcional de polinomios en *x*. La función `polydecomp` devuelve una lista `[`*p_1*`, ..., `*p_n*`]` tal que

```
lambda([x],p_1)(lambda ([x],p_2)(...(lambda([x],p_n)(x))...))
```

es igual a *p*. El grado de *p_i* es mayor que 1 para *i* menor que *n*.

Esta descomposición no es única.

Ejemplos:

```
(%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]
```

La siguiente función compone `L = [e_1, ..., e_n]` como funciones de `x`; se trata de la inversa de `polydecomp`:

```
compose (L, x) :=
  block ([r : x], for e in L do r : subst (e, x, r), r) $
```

Se vuelve a obtener el resultado del ejemplo de más arriba haciendo uso de `compose`:

```
(%i3) polydecomp (compose ([x^2 - a, x^3 - x - 1], x), x);
                          2       3
(%o3)                   [x  - a, x  - x - 1]
```

Nótese que aunque `compose (polydecomp (p, x), x)` devuelve siempre *p* (sin expandir), `polydecomp (compose ([p_1, ..., p_n], x), x)` *no* devuelve necesariamente `[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
```

`polymod (p)` [Función]
`polymod (p, m)` [Función]

Convierte el polinomio *p* a una representación modular respecto del módulo actual, que es el valor almacenado en la variable `modulus`.

La llamada `polymod (p, m)` especifica un módulo *m* para ser utilizado en lugar de valor almacenado en `modulus`.

Véase `modulus`.

`powers (expr, x)` [Función]

Devuelve las potencias de *x* dentro de *expr*.

La instrucción `load ("powers")` carga esta función.

`quotient (p_1, p_2)` [Función]
`quotient (p_1, p_2, x_1, ..., x_n)` [Función]

Devuelve el polinomio *p_1* dividido por el polinomio *p_2*. Los argumentos *x_1*, ..., *x_n* se interpretan como en la función `ratvars`.

La función `quotient` devuelve el primer elemento de la lista devuelta por `divide`.

`rat (expr)` [Función]
`rat (expr, x_1, ..., x_n)` [Función]

Convierte *expr* al formato canónico racional (canonical rational expression o CRE) expandiendo y combinando todos los términos sobre un denominador común y cancelando el máximo común divisor del numerador y denominador, así como convirtiendo números decimales en coma flotante a números racionales dentro de la tolerancia indicada por `ratepsilon`. Las variables se ordenan de acuerdo a *x_1*, ..., *x_n* si se han especificado, como en la función `ratvars`.

En general, `rat` no simplifica otras funciones que no sean la suma `+`, resta `-`, multiplicación `*`, división `/` y exponenciación de exponente entero, mientras que `ratsimp` sí lo hace. Nótese que los átomos (números y variables) en expresiones en formato CRE no son los mismos que en el formato general. Por ejemplo, `rat(x)- x` devuelve `rat(0)`, que tiene una representación interna diferente de 0.

Si `ratprint` vale `false` no aparecerán mensajes informando al usuario sobre la conversión de números decimales en coma flotante a números racionales.

Si `keepfloat` vale `true` no se convertirán números decimales en coma flotante a números racionales.

Véanse también `ratexpand` y `ratsimp`.

Ejemplos:

```
(%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` [Variable opcional]

Valor por defecto: `true`

Si `ratalgdenom` vale `true`, se permite la racionalización de denominadores eliminando radicales. La variable `ratalgdenom` sólo tiene efecto cuando expresiones en formato canónico (CRE) están siendo utilizadas en modo algebraico.

`ratcoef` (*expr*, *x*, *n*) [Función]

`ratcoef` (*expr*, *x*) [Función]

Devuelve el coeficiente de la expresión `x^n` dentro de la expresión *expr*. Si se omite, *n* se considera igual a 1.

El valor devuelto está libre de las variables en *x*, excepto quizás en un sentido no racional. Si no existe un coeficiente de este tipo se devuelve 0.

La función `ratcoef` expande y simplifica racionalmente su primer argumento, por lo que puede dar una respuesta diferente a la dada por la función `coeff`, la cual tiene un carácter puramente sintáctico. Así, `ratcoef ((x + 1)/y + x, x)` devuelve `(y + 1)/y`, mientras que `coeff` devuelve 1.

La llamada `ratcoef (expr, x, 0)`, siendo *expr* una suma, devuelve una suma formada por los términos que no contienen *x*.

Puesto que *expr* se simplifica racionalmente antes de ser examinada, algunos coeficientes puede que no aparezcan como en la expresión original.

Ejemplo:

```
(%i1) s: a*x + b*x + 5$
(%i2) ratcoef (s, a + b);
(%o2)                           x
```

`ratdenom` (*expr*) [Función]

Devuelve el denominador de *expr*, después de transformar *expr* al formato canónico (CRE). El valor retornado está también en formato CRE.

El argumento *expr* se transforma al formato CRE por la función `rat`, a menos que ya esté en este formato. Esta conversión puede cambiar la forma de *expr* colocando todos sus términos sobre un denominador común.

La función `denom` es parecida, pero devuelve una expresión general en lugar de una CRE. Tampoco `denom` intenta colocar todos sus términos sobre un denominador común, de manera que algunas expresiones que son consideradas como divisiones por `ratdenom`, no son tales para `denom`.

`ratdenomdivide` [Variable opcional]

Valor por defecto: `true`

Si `ratdenomdivide` vale `true`, la función `ratexpand` expande una fracción en la que el numerador es una suma en una suma de divisiones. En otro caso, `ratexpand` reduce una suma de divisiones a una única fracción, cuyo numerador es la suma de los denominadores de cada fracción.

Ejemplos:

```
(%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*) [Función]

Deriva la expresión racional *expr* con respecto a *x*. El argumento *expr* debe ser una fracción algebraica o un polinomio en *x*. El argumento *x* puede ser una variable o una subexpresión de *expr*.

El resultado equivale al devuelto por `diff`, aunque es posible que se obtenga en una forma diferente. La función `ratdiff` puede ser más rápida que `diff` en expresiones racionales.

La función `ratdiff` devuelve una expresión en formato canónico o CRE si `expr` es también una expresión CRE. En otro caso, `ratdiff` devuelve una expresión general.

La función `ratdiff` considera únicamente la dependencia de *expr* respecto de *x*, ignorando cualquier dependencia establecida por `depends`.

Ejemplo:

```
(%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*) [Función]

Devuelve su argumento como una expresión general. Si *expr* es una expresión general, se devuelve sin cambios.

Normalmente se invoca a `ratdisrep` a fin de convertir una expresión en formato canónico (CRE) al formato general, lo que puede ser utilizado si se quiere parar el

contagio que produce el formato CRE, o para utilizar funciones racionales en contextos no racionales.

Véase también `totaldisrep`.

`ratexpand` (*expr*) [Función]
`ratexpand` [Variable opcional]

Expande *expr* multiplicando productos de sumas y sumas con exponentes, combinando fracciones con común denominador, cancelando el máximo común divisor del numerador y del denominador y luego dividiendo los sumandos del numerador por el denominador.

El valor que devuelve `ratexpand` es una expresión general, incluso cuando *expr* está en formato canónico o CRE.

Si la variable `ratexpand` vale `true` hará que las expresiones CRE se expandan completamente cuando se conviertan al formato general o se muestren en el terminal, mientras que si vale `false` se mostrarán de forma recursiva. Véase también `ratsimp`.

Si `ratdenomdivide` vale `true`, `ratexpand` expande una fracción en la que el numerador es una suma en una suma de fracciones, todas ellas con denominador común. En otro caso, `ratexpand` reduce una suma de fracciones en una única fracción, cuyo numerador es la suma de los numeradores de cada fracción.

Si `keepfloat` vale `true`, los números decimales en coma flotante no se racionalizan cuando las expresiones que los contienen se convierten al formato canónico racional (CRE).

Ejemplos:

```
(%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` [Variable opcional]

Valor por defecto: `false`

Si `ratfac` vale `true`, las expresiones canónicas (CRE) se manipulan en una forma parcialmente factorizada.

Durante las operaciones racionales, las expresiones se mantienen completamente factorizadas tanto como sea posible sin llamar a `factor`. Esto debería ahorrar espacio y tiempo en algunos cálculos. El numerador y denominador se hacen primos relativos, por ejemplo `rat ((x^2 - 1)^4/(x + 1)^2)` devuelve `(x - 1)^4 (x + 1)^2)`, pero los factores dentro de cada parte pueden no ser primos relativos.

En el paquete `ctensr` sobre manipulación de tensores por componentes, los tensores de Ricci, Einstein, Riemann y Weyl y la curvatura escalar se factorizan automáticamente si `ratfac` vale `true`; `ratfac` *debe activarse únicamente en aquellos casos en los que se sabe que el número de términos de las componentes tensoriales es pequño.*

Nota: Los esquemas de comportamiento basados en `ratfac` y `ratweight` son incompatibles y no se debe pretender usarlos al mismo tiempo.

`ratnumer (`*expr*`)` [Función]

Devuelve el numerador de *expr*, después de reducir *expr* a su forma canónica (CRE). El valor retornado está también en formato CRE.

El argumento *expr* se transforma al formato CRE por la función `rat`, a menos que ya esté en este formato. Esta conversión puede cambiar la forma de *expr* colocando todos sus términos sobre un denominador común.

Es parecida a la función `num`, pero devuelve una expresión general en lugar de una CRE. Además, `num` no intenta colocar todos los términos sobre un denominador común, de manera que algunas expresiones que son consideradas fracciones por `ratnumer` no se consideran como tales por `num`.

`ratp (`*expr*`)` [Función]

Devuelve `true` si *expr* es una expresión canónica racional (canonical rational expression o CRE) o una CRE extendida, en caso contrario devuelve `false`.

Las expresiones CRE son creadas por `rat` y funciones asociadas. Las CRE extendidas son creadas por `taylor` y funciones asociadas.

`ratprint` [Variable opcional]

Valor por defecto: `true`

Si `ratprint` vale `true`, se muestra al usuario un mensaje dando cuenta de la conversión de números decimales en coma flotante a formato racional.

`ratsimp (`*expr*`)` [Función]
`ratsimp (`*expr*`, `*x_1*`, ..., `*x_n*`)` [Función]

Simplifica la expresión *expr* y todas sus subexpresiones, incluyendo los argumentos de funciones no racionales. El resultado es un cociente de dos polinomios en una forma recursiva, esto es, los coeficientes de la variable principal son polinomios respecto de las otras variables. Las variables pueden incluir funciones no racionales, como `sin (x^2 + 1)`, y los argumentos de tales funciones son también racionalmente simplificados.

La llamada `ratsimp (`*expr*`, `*x_1*`, ..., `*x_n*`)` permite la simplificación racional con la especificación del orden de las variables, como en `ratvars`.

Si `ratsimpexpons` vale `true`, `ratsimp` se aplica a los exponentes de las expresiones durante la simplificación.

Véase también `ratexpand`. Nótese que `ratsimp` se ve afectada por algunas de las variables globales que controlan a `ratexpand`.

Ejemplos:

```
(%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` [Variable opcional]

Valor por defecto: `false`

Si `ratsimpexpons` vale `true`, `ratsimp` se aplica a los exponentes de las expresiones durante la simplificación.

`radsubstflag` [Variable opcional]

Valor por defecto: `false`

Si `radsubstflag` vale `true` se permite a `ratsubst` hacer la sustitución `u` por `sqrt (x)` in `x`.

`ratsubst` (*a*, *b*, *c*) [Función]

Sustituye *b* por *a* en *c* y devuelve la expresión resultante. El argumento *b* puede ser una suma, un producto, una potencia, etc.

La función `ratsubst` reconoce el significado de las expresiones, mientras que `subst` tan solo realiza sustituciones sintácticas. Así por ejemplo, `subst (a, x + y, x + y + z)` devuelve `x + y + z` cuando `ratsubst` devuelve `z + a`.

Si `radsubstflag` vale `true`, `ratsubst` sustituye radicales en expresiones que no los contienen explícitamente.

`ratsubst` ignora el valor `true` de la variable opcional `keepfloat`.

Ejemplos:

```
(%i1) ratsubst (a, x*y^2, x^4*y^3 + x^4*y^8);
                                3      4
(%o1)                        a x  y + a
(%i2) cos(x)^4 + cos(x)^3 + cos(x)^2 + cos(x) + 1;
                4         3         2
(%o2)       cos (x) + cos (x) + cos (x) + cos(x) + 1
(%i3) ratsubst (1 - sin(x)^2, cos(x)^2, %);
            4           2                      2
(%o3)    sin (x) - 3 sin (x) + cos(x) (2 - sin (x)) + 3
(%i4) ratsubst (1 - cos(x)^2, sin(x)^2, sin(x)^4);
                          4           2
(%o4)                cos (x) - 2 cos (x) + 1
(%i5) radsubstflag: false$
(%i6) ratsubst (u, sqrt(x), x);
(%o6)                             x
(%i7) radsubstflag: true$
(%i8) ratsubst (u, sqrt(x), x);
                                   2
(%o8)                             u
```

`ratvars` (*x_1*, ..., *x_n*) [Función]
`ratvars` () [Función]
`ratvars` [Variable del sistema]

Declara como variables principales *x_1*, ..., *x_n* en expresiones racionales. Si *x_n* está presente en una expresión racional, se considerará como variable principal. Si no está presente, entonces se considerará principal a la variable *x_[n-1]* si aparece en la expresión, se continúa así hasta *x_1*, que se considerará como variable principal sólo si ninguna de las variables que le siguen está presente en la expresión.

Si una variable de la expresión racional no está presente en la lista `ratvars`, se le dará una prioridad inferior a la de *x_1*.

Los argumentos de `ratvars` pueden ser tanto variables como funciones no racionales como `sin(x)`.

La variable `ratvars` es una lista que contiene los argumentos pasados a la función `ratvars` la última vez que fue invocada. Cada llamada a la función `ratvars` reinicializa la lista. La llamada `ratvars ()` vacía la lista.

`ratvarswitch` [Variable opcional]

Valor por defecto: `true`

Maxima almacena una lista interna en la variable Lisp `VARLIST` cuyo contenido son las variables principales de las expresiones racionales. Cuando `ratvarswitch` vale `true`, su valor por defecto, cada evaluación comienza con la lista `VARLIST` vacía. En caso

contrario, las variables principales de las expresiones anteriores se mantienen en la lista `VARLIST`.

Las variables principales declaradas con la función `ratvars` no se ven afectadas por la opción `ratvarswitch`.

Ejemplos:

Cuando `ratvarswitch` vale `true`, su valor por defecto, cada evaluación comienza con la lista `VARLIST` vacía.

```
(%i1) ratvarswitch:true$

(%i2) rat(2*x+y^2);
                                2
(%o2)/R/                       y  + 2 x
(%i3) :lisp varlist
($X $Y)

(%i3) rat(2*a+b^2);
                                2
(%o3)/R/                       b  + 2 a

(%i4) :lisp varlist
($A $B)
```

Cuando `ratvarswitch` vale `false`, las variables principales de las expresiones anteriores se mantienen en lista `VARLIST`.

```
(%i4) ratvarswitch:false$

(%i5) rat(2*x+y^2);
                                2
(%o5)/R/                       y  + 2 x
(%i6) :lisp varlist
($X $Y)

(%i6) rat(2*a+b^2);
                                2
(%o6)/R/                       b  + 2 a

(%i7) :lisp varlist
($A $B $X $Y)
```

`ratweight` (*x_1*, *w_1*, ..., *x_n*, *w_n*) [Función]

`ratweight` () [Función]

Asigna un peso *w_i* a la variable *x_i*. Un término será reemplazado por 0 si su peso excede el valor de la variable `ratwtlvl` (por defecto no se realiza el truncamiento). El peso de un término es la suma de los productos de los pesos de las variables que lo forman multiplicados por sus exponentes. Por ejemplo, el peso de `3 x_1^2 x_2` es `2 w_1 + w_2`. El truncamiento basado en `ratwtlvl` solamente se lleva a cabo cuando se multiplican o se elevan a potencias expresiones canónicas (CRE).

La llamada `ratweight ()` devuelve la lista acumulada de asignaciones de pesos.

Nota: Los esquemas de comportamiento basados en `ratfac` y `ratweight` son incompatibles y no se debe pretender usarlos al mismo tiempo.

Ejemplos:

```
(%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` [Variable del sistema]

Valor por defecto: `[]`

La variable `ratweights` es una lista que contiene los pesos asignados por `ratweight`. Las lista es acumulativa, en el sentido de que cada llamada a `ratweight` añade nuevos elementos a la lista.

`ratwtlvl` [Variable opcional]

Valor por defecto: `false`

La variable `ratwtlvl` se utiliza en combinación con la función `ratweight` para controlar el truncamiento de expresiones racionales canónicas (CRE). Con el valor por defecto, `false`, no se produce truncamiento alguno.

`remainder` (*p_1*, *p_2*) [Función]
`remainder` (*p_1*, *p_2*, *x_1*, ..., *x_n*) [Función]

Devuelve el resto de la división del polinomio *p_1* entre *p_2*. Los argumentos *x_1*, ..., *x_n* se interpretan como en `ratvars`.

La función `remainder` devuelve el segundo elemento de la lista retornada por `divide`.

`resultant` (*p_1*, *p_2*, *x*) [Función]

Calcula la resultante de los dos polinomios *p_1* y *p_2*, eliminando la variable *x*. La resultante es un determinante de los coeficientes de *x* en *p_1* y *p_2*, que es igual a cero si sólo si *p_1* y *p_2* tienen un factor común no constante.

Si *p_1* o *p_2* pueden ser factorizados, puede ser necesario llamar a `factor` antes que invocar a `resultant`.

La variable opcional `resultant` controla qué algoritmo será utilizado para calcular la resultante. Véanse `option_resultant` y `resultant`.

La función `bezout` toma los mismos argumentos que `resultant` y devuelve una matriz. El determinante del valor retornado es la resultante buscada.

Ejemplos:

```
(%i1) resultant(2*x^2+3*x+1, 2*x^2+x+1, x);
(%o1)                           8
(%i2) resultant(x+1, x+1, x);
```

```
(%o2)                          0
(%i3) resultant((x+1)*x, (x+1), x);
(%o3)                          0
(%i4) resultant(a*x^2+b*x+1, c*x + 2, x);
                         2
(%o4)                   c  - 2 b c + 4 a

(%i5) bezout(a*x^2+b*x+1, c*x+2, x);
                        [ 2 a  2 b - c ]
(%o5)                   [              ]
                        [  c      2    ]
(%i6) determinant(%);
(%o6)                   4 a - (2 b - c) c
```

`resultant` [Variable opcional]

Valor por defecto: `subres`

La variable opcional `resultant` controla qué algoritmo será utilizado para calcular la resultante con la función `resultant`. Los valores posibles son:

`subres` para el algoritmo PRS (*polynomial remainder sequence*) subresultante,

`mod` para el algoritmo resultante modular y

`red` para el algoritmo PRS (*polynomial remainder sequence*) reducido.

En la mayor parte de problemas, el valor por defecto, `subres`, es el más apropiado. Pero en el caso de problemas bivariantes o univariantes de grado alto, puede ser mejor utilizar `mod`.

`savefactors` [Variable opcional]

Valor por defecto: `false`

Si `savefactors` vale `true`, los factores de una expresión producto se almacenan por ciertas funciones a fin de acelerar posteriores factorizaciones de expresiones que contengan algunos de estos factores.

`showratvars` (*expr*) [Función]

Devuelve una lista de las variables de expresiones canónicas racionales (CRE) en la expresión `expr`.

Véase también `ratvars`.

`tellrat` (*p_1*, ..., *p_n*) [Función]

`tellrat` () [Función]

Añade al anillo de enteros algebraicos conocidos por Maxima los elementos que son soluciones de los polinomios *p_1*, ..., *p_n*. Cada argumento *p_i* es un polinomio de coeficientes enteros.

La llamada `tellrat (x)` hace que se sustituya 0 por *x* en las funciones racionales.

La llamada `tellrat ()` devuelve una lista con las sustituciones actuales.

A la variable `algebraic` se le debe asignar el valor `true` a fin de poder realizar la simplificación de enteros algebraicos.

Maxima reconoce la unidad imaginaria `%i` y todas las raíces de los enteros.

La instrucción `untellrat` borra todas las propiedades de `tellrat`.

Es ambiguo aplicar `tellrat` a un polinomio multivariante tal como `tellrat (x^2 - y^2)`, pues no se sabe si sustituir *y*`^2` por *x*`^2` o al revés. Maxima sigue un cierto orden, pero si el usuario quiere especificar uno en concreto, puede hacerlo mediante la sintaxis `tellrat (y^2 = x^2)`, que indica que se ponga *x*`^2` en lugar de *y*`^2`.

Ejemplos:

```
(%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*) [Función]

Convierte cada subexpresión de *expr* del formato canónico (CRE) al general y devuelve el resultado. Si *expr* está en formato CRE entonces `totaldisrep` es idéntico a `ratdisrep`.

La función `totaldisrep` puede ser útil para modificar expresiones como las ecuaciones, listas, matrices, etc., que tienen algunas subexpresiones en formato CRE.

`untellrat` (*x_1*, ..., *x_n*) [Función]

Elimina de *x_1*, ..., *x_n* las propiedades relacionadas con `tellrat`.

15 Funciones Especiales

15.1 Introducción a las funciones especiales

A continuación se especifican las notaciones correspondientes a las funciones especiales:

```
bessel_j (index, expr)    Función de Bessel de primera especie
bessel_y (index, expr)    Función de Bessel de segunda especie
bessel_i (index, expr)    Función de Bessel modificada de primera especie
bessel_k (index, expr)    Función de Bessel modificada de segunda especie
hankel_1 (v,z)            Función de Hankel de primera especie
hankel_2 (v,z)            Función de Hankel de segunda especie
struve_h (v,z)            Función H de Struve
struve_l (v,z)            Función L de Struve
%p[u,v] (z)               Función de Legendre de primera especie
%q[u,v] (z)               Función de Legendre de segunda especie
%f[p,q] ([], [], expr)    Función hipergeométrica generalizada
gamma(z)                  Función Gamma
gamma_incomplete_lower(a,z) Función Gamma incompleta inferior
gamma_incomplete (a,z)    Extremo de la función Gamma incompleta
hypergeometric(l1, l2, z) Función hipergeométrica
slommel
%m[u,k] (z)               Función de Whittaker de primera especie
%w[u,k] (z)               Función de Whittaker de segunda especie
erfc (z)                  Complemento de la función de error, erf
expintegral_e (v,z)       Integral exponencial E
expintegral_e1 (z)        Integral exponencial E1
expintegral_ei (z)        Integral exponencial Ei
expintegral_li (z)        Integral logarítmica Li
expintegral_si (z)        Integral exponencial Si
expintegral_ci (z)        Integral exponencial Ci
expintegral_shi (z)       Integral exponencial Shi
expintegral_chi (z)       Integral exponencial Chi
kelliptic (z)             Integral elíptica completa
                                 de primera especie (K)
parabolic_cylinder_d(v,z) Función D de cilindro parabólico
```

15.2 Funciones de Bessel

`bessel_j` (*v*, *z*) [Función]

Función de Bessel de primera especie de orden v y argumento z.

La función `bessel_j` se define como

$$\sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{z}{2}\right)^{v+2\,k}}{k!\,\Gamma\left(v+k+1\right)}$$

aunque la serie infinita no se utiliza en los cálculos.

`bessel_y (v, z)` [Función]

Función de Bessel de segunda especie de orden v y argumento z.

La función `bessel_y` se define como

$$\frac{\cos(\pi\, v)\ J_v(z) - J_{-v}(z)}{\sin(\pi\, v)}$$

si v no es un entero. En caso de que v sea un entero n, se calcula el límite cuando v se aproxima a n.

`bessel_i (v, z)` [Función]

Función modificada de Bessel de primera especie de orden v y argumento z.

La función `bessel_i` se define como

$$\sum_{k=0}^{\infty} \frac{1}{k!\,\Gamma(v+k+1)} \left(\frac{z}{2}\right)^{v+2\,k}$$

aunque la serie infinita no se utiliza en los cálculos.

`bessel_k (v, z)` [Función]

Función modificada de Bessel de segunda especie de orden v y argumento z.

La función `bessel_k` se define como

$$\frac{\pi\ \csc(\pi\, v)\ (I_{-v}(z) - I_v(z))}{2}$$

si v no es un entero. Si v es igual al entero n, entonces se calcula el límite cuando v tiende a n.

`hankel_1 (v, z)` [Función]

Función de Hankel de primera especie de orden v y argumento z (A&S 9.1.3). La función `hankel_1` se define como

```
bessel_j(v,z) + %i * bessel_y(v,z)
```

Maxima evalúa `hankel_1` numéricamente para el orden real v y el argumento complejo z en doble precisión (float). La evaluación numérica en gran precisión (bigfloat) y para órdenes complejos no está implementada.

Si `besselexpand` vale `true`, `hankel_1` se expande en términos de funciones elementales cuando el orden v es la mitad de un entero impar. Véase al respecto `besselexpand`.

Maxima reconoce la derivada de `hankel_1` con respecto del argumento z.

Ejemplos:

Evaluación numérica:

```
(%i1) hankel_1(1,0.5);
(%o1)              .2422684576748738 - 1.471472392670243 %i
(%i2) hankel_1(1,0.5+%i);
(%o2)             - .2558287994862166 %i - 0.239575601883016
```

No se soportan órdenes complejos. Maxima devuelve una forma nominal:

```
(%i3) hankel_1(%i,0.5+%i);
(%o3)                       hankel_1(%i, %i + 0.5)
```

Expansión de `hankel_1` cuando `besselexpand` vale `true`:

```
(%i4) hankel_1(1/2,z),besselexpand:true;
                        sqrt(2) sin(z) - sqrt(2) %i cos(z)
(%o4)                   ----------------------------------
                                 sqrt(%pi) sqrt(z)
```

Derivada de `hankel_1` respecto del argumento z. No está soportada la derivada respecto del orden v. Maxima devuelve una forma nominal:

```
(%i5) diff(hankel_1(v,z),z);
                     hankel_1(v - 1, z) - hankel_1(v + 1, z)
(%o5)                ---------------------------------------
                                          2
(%i6) diff(hankel_1(v,z),v);
                               d
(%o6)                          -- (hankel_1(v, z))
                               dv
```

`hankel_2` (*v*, *z*) [Función]

Función de Hankel de segunda especie de orden v y argumento z (A&S 9.1.4). La función `hankel_2` se define como

```
bessel_j(v,z) - %i * bessel_y(v,z)
```

Maxima evalúa `hankel_2` numéricamente para el orden real v y el argumento complejo z en doble precisión (float). La evaluación numérica en gran precisión (bigfloat) y para órdenes complejos no está implementada.

Si `besselexpand` vale `true`, `hankel_2` se expande en términos de funciones elementales cuando el orden v es la mitad de un entero impar. Véase al respecto `besselexpand`.

Maxima reconoce la derivada de `hankel_2` con respecto del argumento z.

Véanse ejemplos en `hankel_1`.

`besselexpand` [Variable optativa]

Valor por defecto: `false`

Controla la expansión de las funciones de Bessel cuando el orden es la mitad de un entero impar. En tal caso, las funciones de Bessel se pueden expandir en términos de otras funciones elementales. Si `besselexpand` vale `true`, se expande la función de Bessel.

```
(%i1) besselexpand: false$
(%i2) bessel_j (3/2, z);
                                      3
(%o2)                       bessel_j(-, z)
                                      2
(%i3) besselexpand: true$
(%i4) bessel_j (3/2, z);
```

```
                                                 sin(z)   cos(z)
                                        sqrt(2) sqrt(z) (------ - ------)
                                                          2        z
                                                         z
(%o4)                                   ---------------------------------
                                                   sqrt(%pi)
```

`scaled_bessel_i` (*v*, *z*) [Función]

Es la función de Bessel modificada de primera especie de orden v y argumento z, es decir $scaled_besseli(v,z) = exp(-abs(z)) * bessel_i(v,z)$. Esta función es especialmente útil para calcular $bessel_i$ cuando z es grande. Sin embargo, Maxima no sabe mucho más sobre esta función. En cálculos simbólicos, quizás sea preferible trabajar directamente con la expresión `exp(-abs(z))*bessel_i(v, z)`.

`scaled_bessel_i0` (*z*) [Función]

Idéntica a `scaled_bessel_i(0,z)`.

`scaled_bessel_i1` (*z*) [Función]

Idéntica a `scaled_bessel_i(1,z)`.

`%s` [*u*,*v*] (*z*) [Función]

Función s[u,v](z) de Lommel. Gradshteyn & Ryzhik 8.570.1.

15.3 Funciones de Airy

Las funciones de Airy Ai(x) y Bi(x) se definen en la sección 10.4. de Abramowitz and Stegun, *Handbook of Mathematical Functions*.

`y = Ai(x)` y `y = Bi(x)` son dos soluciones linealmente independientes de la ecuación diferencia de Airy `diff (y(x), x, 2) - x y(x) = 0`.

Si el argumento `x` es un número decimal en coma flotante real o complejo, se devolverá el valor numérico de la función.

`airy_ai` (*x*) [Función]

Función de Airy Ai(x). (A&S 10.4.2)

La derivada `diff (airy_ai(x), x)` es `airy_dai(x)`.

Véanse `airy_bi`, `airy_dai` y `airy_dbi`.

`airy_dai` (*x*) [Función]

Es la derivada de la función Ai de Airy, `airy_ai(x)`.

Véase `airy_ai`.

`airy_bi` (*x*) [Función]

Es la función Bi de Airy, tal como la definen Abramowitz y Stegun, *Handbook of Mathematical Functions*, Sección 10.4. Se trata de la segunda solución de la ecuación de Airy `diff (y(x), x, 2) - x y(x) = 0`.

Si el argumento `x` es un número decimal real o complejo, se devolverá el valor numérico de `airy_bi` siempre que sea posible. En los otros casos, se devuelve la expresión sin evaluar.

La derivada `diff (airy_bi(x), x)` es `airy_dbi(x)`.

Véanse `airy_ai` y `airy_dbi`.

`airy_dbi (x)` [Función]

Es la derivada de la función Bi de Airy, `airy_bi(x)`.

Véanse `airy_ai` y `airy_bi`.

15.4 Funciones Gamma y factorial

Las funciones gamma, beta, psi y gamma incompleta están definidas en el capítulo 6 de Abramowitz y Stegun, *Handbook of Mathematical Functions*.

`bffac (expr, n)` [Función]

Versión para "bigfloat" de la función factorial (Gamma desplazada). El segundo argumento indica cuántos dígitos se conservan y devuelven, pudiendo utilizarse para obtener algunas cifras extra.

`algepsilon` [Variable optativa]

Valor por defecto: 10^8

El valor de `algepsilon` es usado por `algsys`.

`bfpsi (n, z, fpprec)` [Función]
`bfpsi0 (z, fpprec)` [Función]

La función `bfpsi` es la poligamma de argumento real z y de orden el entero n. La función `bfpsi0` es la digamma. La llamada `bfpsi0 (z, fpprec)` equivale a `bfpsi (0, z, fpprec)`.

Estas funciones devuelven valores "bigfloat". La variable *fpprec* es la precisión "bigfloat" del valor de retorno.

`cbffac (z, fpprec)` [Función]

Calcula el factorial de números complejos de punto flotante grandes.

La instrucción `load ("bffac")` carga esta función.

`gamma (x)` [Función]

La definición básica de la función gamma (A&S 6.1.1) es

$$\Gamma(z) = \int_0^\infty t^{z-1}\, e^{-t}\, dt$$

Maxima simplifica `gamma` para enteros positivos y para fracciones positivas o negativas. Para fracciones de denominador dos, el resultado es un número racional multiplicado por `sqrt(%pi)`. La simplificación para valores enteros la controla `factlim`. Para enteros mayores que `factlim` el resultado numérico de la función factorial, la cual se utiliza para calcular `gamma`, producirá un desbordamiento. La simplificación para números racionales la controla `gammalim` para evitar desbordamientos. Véanse también `factlim` y `gammalim`.

Para enteros negativos, `gamma` no está definida.

Maxima puede evaluar `gamma` numéricamente para valores reales y complejos, tanto en formato float (doble precisión) como big float (precisión arbitraria).

La función `gamma` tiene simetría especular.

Si `gamma_expand` vale `true`, Maxima expande `gamma` para argumentos del tipo `z+n` y `z-n`, siendo `n` un entero.

Maxima conoce la derivada de `gamma`.

Ejemplos:

Simplificación para enteros, fracciones de denominador dos y números racionales:

```
(%i1) map('gamma,[1,2,3,4,5,6,7,8,9]);
(%o1)               [1, 1, 2, 6, 24, 120, 720, 5040, 40320]

(%i2) map('gamma,[1/2,3/2,5/2,7/2]);
                             sqrt(%pi)  3 sqrt(%pi)  15 sqrt(%pi)
(%o2)          [sqrt(%pi), ---------, -----------, ------------]
                                2           4            8

(%i3) map('gamma,[2/3,5/3,7/3]);
                                          2           1
                                   2 gamma(-)  4 gamma(-)
                              2           3           3
(%o3)                [gamma(-), ----------, ----------]
                              3       3           9
```

Evaluación numérica para valores reales y complejos:

```
(%i4) map('gamma,[2.5,2.5b0]);
(%o4)             [1.329340388179137, 1.329340388179137b0]

(%i5) map('gamma,[1.0+%i,1.0b0+%i]);
(%o5) [.4980156681183558 - .1549498283018108 %i,
                      4.980156681183561b-1 - 1.549498283018107b-1 %i]
```

Simetría especular:

```
(%i6) declare(z,complex)$
(%i7) conjugate(gamma(z));
(%o7)                         gamma(conjugate(z))
```

Maxima expande `gamma(z+n)` y `gamma(z-n)` si `gamma_expand` vale `true`:

```
(%i8) gamma_expand:true$
(%i9) [gamma(z+1),gamma(z-1),gamma(z+2)/gamma(z+1)];
                                       gamma(z)
(%o9)                  [z gamma(z), --------, z + 1]
                                        z - 1
```

Derivada de `gamma`:

```
(%i10) diff(gamma(z),z);
(%o10)                          psi (z) gamma(z)
                                   0
```

Véase también `makegamma`.

La constante de Euler-Mascheroni es `%gamma`.

`log_gamma` (*z*) [Función]

Logaritmo natural de la función gamma.

gamma_incomplete_lower (a, z) [Función]

Función gamma incompleta inferior (A&S 6.5.2):

$$\gamma\left(a,z\right)=\int_{0}^{z}t^{a-1}\,e^{-t}\,dt$$

Véase también gamma_incomplete (función gamma incompleta superior).

gamma_incomplete (a,z) [Función]

Función gamma incompleta superior, A&S 6.5.3:

$$\Gamma\left(a,z\right)=\int_{z}^{\infty}t^{a-1}\,e^{-t}\,dt$$

Véanse también gamma_expand para controlar cómo se expresa gamma_incomplete en términos de funciones elementales y de erfc.

Véanse también las funciones relacionadas gamma_incomplete_regularized y gamma_incomplete_generalized.

gamma_incomplete_regularized (a,z) [Función]

Función gamma incompleta superior regularizada, A&S 6.5.1.

$$Q\left(a,z\right)=\frac{\Gamma\left(a,z\right)}{\Gamma\left(a\right)}$$

Véanse también gamma_expand para controlar cómo se expresa gamma_incomplete en términos de funciones elementales y de erfc.

Véase también gamma_incomplete.

gamma_incomplete_generalized (a,z1,z1) [Función]

Función gamma incompleta generalizada.

$$\Gamma\left(a,z_1,z_2\right)=\int_{z_1}^{z_2}t^{a-1}\,e^{-t}\,dt$$

Véanse también gamma_incomplete y gamma_incomplete_regularized.

gamma_expand [Variable opcional]

Valor por defecto: false

gamma_expand controla la expansión de gamma_incomplete. Si gamma_expand vale true, gamma_incomplete(v,z) se expande en términos de z, exp(z) y erfc(z), siempre que sea posible.

```
(%i1) gamma_incomplete(2,z);
(%o1)                       gamma_incomplete(2, z)
(%i2) gamma_expand:true;
(%o2)                                  true
(%i3) gamma_incomplete(2,z);
                                                - z
(%o3)                              (z + 1) %e
```

```
(%i4) gamma_incomplete(3/2,z);
                                    - z   sqrt(%pi) erfc(sqrt(z))
(%o4)               sqrt(z) %e    + ----------------------
                                              2
```

gammalim [Variable optativa]

Valor por defecto: 10000

La variable gammalim controla la simplificación de la función gamma con argumentos enteros o racionales. Si el valor absoluto del argumento no es mayor que gammalim, entonces se realizará la simplificación. Nótese que la variable factlim también controla la simplificación del resultado de gamma con argumento entero.

makegamma (*expr*) [Función]

Transforma las funciones binomial, factorial y beta que aparecen en *expr* en funciones gamma.

Véase también makefact.

beta (a, *b*) [Función]

La función beta se define como gamma(a) gamma(b)/gamma(a+b) (A&S 6.2.1).

Maxima simplifica la función beta para enteros positivos y números racionales cuya suma sea entera. Si beta_args_sum_to_integer vale true, Maxima también simplifica expresiones generales cuya suma sea también entera.

Cuando *a* o *b* sean nulos, la función beta no está definida.

En general, la función beta no está definida para enteros negativos. La excepción es para *a*=-*n*, siendo *n* un entero positivo y *b* otro entero positivo tal que *b*<=*n*, entonces es posible definir una continuación analítica. En este caso Maxima devuelve un resultado.

Si beta_expand vale true, expresiones como beta(a+n,b), beta(a-n,b), beta(a,b+n) o beta(a,b-n), siendo n entero, se simplifican.

Maxima puede evaluar la función beta para valores reales y complejos, tanto de tipo decimal flotante o big float. Para la evaluación numérica Maxima utiliza log_gamma:

```
  - log_gamma(b + a) + log_gamma(b) + log_gamma(a)
%e
```

Maxima reconoce la simetría de la función beta.

Maxima conoce las derivadas de la función beta, tanto respecto de *a* como de *b*.

Para expresar la función beta como un cociente de funciones gamma, véase makegamma.

Ejemplos:

Simplificación cuando uno de sus argumentos es entero:

```
(%i1) [beta(2,3),beta(2,1/3),beta(2,a)];
                                1   9      1
(%o1)                          [--, -, ---------]
                                12  4  a (a + 1)
```

Simplificación para argumentos racionales que suman un entero:

```
(%i2) [beta(1/2,5/2),beta(1/3,2/3),beta(1/4,3/4)];
```

```
                              3 %pi   2 %pi
(%o2)                        [-----, -------, sqrt(2) %pi]
                               8    sqrt(3)
```

Cuando se iguala `beta_args_sum_to_integer` a `true` se simplifican expresiones más generales si la suma de los argumentos se reduce a un entero:

```
(%i3) beta_args_sum_to_integer:true$
(%i4) beta(a+1,-a+2);
                                  %pi (a - 1) a
(%o4)                            ------------------
                                 2 sin(%pi (2 - a))
```

Posibles valores cuando uno de los argumentos es entero negativo:

```
(%i5) [beta(-3,1),beta(-3,2),beta(-3,3)];
                                      1  1    1
(%o5)                               [- -, -, - -]
                                      3  6    3
```

`beta(a+n,b)` o `beta(a-n)` con `n` entero se simplifica si `beta_expand` vale `true`:

```
(%i6) beta_expand:true$
(%i7) [beta(a+1,b),beta(a-1,b),beta(a+1,b)/beta(a,b+1)];
                     a beta(a, b)  beta(a, b) (b + a - 1)  a
(%o7)               [------------, ----------------------, -]
                        b + a              a - 1           b
```

La función beta no está definida si uno de sus argumentos es cero:

```
(%i7) beta(0,b);
beta: expected nonzero arguments; found 0, b
 -- an error.  To debug this try debugmode(true);
```

Evaluación numérica para argumentos reales y complejos de tipo decimal flotante o big float:

```
(%i8) beta(2.5,2.3);
(%o8) .08694748611299981

(%i9) beta(2.5,1.4+%i);
(%o9) 0.0640144950796695 - .1502078053286415 %i

(%i10) beta(2.5b0,2.3b0);
(%o10) 8.694748611299969b-2

(%i11) beta(2.5b0,1.4b0+%i);
(%o11) 6.401449507966944b-2 - 1.502078053286415b-1 %i
```

La función beta es simétrica con simetría especular:

```
(%i14) beta(a,b)-beta(b,a);
(%o14)                                0
(%i15) declare(a,complex,b,complex)$
(%i16) conjugate(beta(a,b));
```

```
(%o16)                beta(conjugate(a), conjugate(b))
```

Derivada de la función beta respecto de `a`:

```
(%i17) diff(beta(a,b),a);
(%o17)              - beta(a, b) (psi (b + a) - psi (a))
                                    0             0
```

`beta_incomplete` (*a*, *b*, *z*) [Función]

La definición básica de la función beta incompleta (A&S 6.6.1) es

```
 z
/
[         b - 1  a - 1
I  (1 - t)      t      dt
]
/
 0
```

Esta definición es posible para $realpart(a) > 0$, $realpart(b) > 0$ y $abs(z) < 1$. Para otras situaciones, la función beta incompleta puede definirse por medio de una función hipergeométrica generalizada:

```
gamma(a) hypergeometric_generalized([a, 1 - b], [a + 1], z) z
```

(Véase Funcións.wolfram.com para una completa definición de la función beta incompleta.)

Para enteros negativos $a = -n$ y enteros positivos $b = m$ con $m <= n$ la función beta incompleta se define como

```
                m - 1           k
                ====  (1 - m)  z
         n - 1  \            k
        z       >    -----------
                /     k! (n - k)
                ====
                k = 0
```

Maxima utiliza esta definición para simplificar `beta_incomplete` cuando *a* es entero negativo.

Cuando *a* es entero positivo, `beta_incomplete` se simplifica para cualesquiera argumentos *b* y *z*, y para *b* entero positivo para cualesquiera argumentos *a* y *z*, con la excepción de cuando *a* sea entero negativo.

Para $z = 0$ y $realpart(a) > 0$, `beta_incomplete` se anula. Para *z=1* y $realpart(b) > 0$, `beta_incomplete` se reduce a la función `beta(a,b)`.

Maxima evalúa `beta_incomplete` numéricamente para valores reales y complejos en forma decimal y big float. La evaluación numérica se realiza expandiendo la función beta incompleta en fracciones continuas.

Si `beta_expand` vale `true`, Maxima expande las expresiones `beta_incomplete(a+n,b,z)` y `beta_incomplete(a-n,b,z)`, siendo *n* entero positivo.

Maxima conoce las derivadas de `beta_incomplete` con respecto a las variables a, b y z, así como la integral respecto de la variable z.

Ejemplos:

Simplificación para a entero positivo:

```
(%i1) beta_incomplete(2,b,z);
                                        b
                                1 - (1 - z)  (b z + 1)
(%o1)                           ----------------------
                                      b (b + 1)
```

Simplificación para b entero positivo:

```
(%i2) beta_incomplete(a,2,z);
                                                    a
                                  (a (1 - z) + 1) z
(%o2)                             ------------------
                                      a (a + 1)
```

Simplificación para a y b enteros positivos:

```
(%i3) beta_incomplete(3,2,z);
                                                    3
                                  (3 (1 - z) + 1) z
(%o3)                             ------------------
                                          12
```

Para a entero negativo y $b <= (-a)$:

```
(%i4) beta_incomplete(-3,1,z);
                                           1
(%o4)                                  - ----
                                            3
                                         3 z
```

Simplificación para los valores $z = 0$ y $z = 1$:

```
(%i5) assume(a>0,b>0)$
(%i6) beta_incomplete(a,b,0);
(%o6)                                    0
(%i7) beta_incomplete(a,b,1);
(%o7)                               beta(a, b)
```

Evaluación numérica, tanto con float (precisión doble) como big float (precisión arbitraria):

```
(%i8) beta_incomplete(0.25,0.50,0.9);
(%o8)                          4.594959440269333
(%i9)  fpprec:25$
(%i10) beta_incomplete(0.25,0.50,0.9b0);
(%o10)                    4.5949594402693240869712O3b0
```

Para $abs(z) > 1$, `beta_incomplete` devuelve un resultado complejo:

```
(%i11) beta_incomplete(0.25,0.50,1.7);
(%o11)              5.244115108584249 - 1.45518047787844 %i
```

Resultados para argumentos complejos más generales:

```
(%i14) beta_incomplete(0.25+%i,1.0+%i,1.7+%i);
(%o14)             2.726960675662536 - .3831175704269199 %i
(%i15) beta_incomplete(1/2,5/4*%i,2.8+%i);
(%o15)             13.04649635168716 %i - 5.802067956270001
(%i16)
```

Expansión cuando `beta_expand` vale `true`:

```
(%i23) beta_incomplete(a+1,b,z),beta_expand:true;
                                                      b  a
                   a beta_incomplete(a, b, z)   (1 - z)  z
(%o23)             -------------------------- - -----------
                             b + a                 b + a

(%i24) beta_incomplete(a-1,b,z),beta_expand:true;
                                                          b  a - 1
           beta_incomplete(a, b, z) (- b - a + 1)   (1 - z)  z
(%o24)     ------------------------------------- - ---------------
                          1 - a                        1 - a
```

Derivada e integral de `beta_incomplete`:

```
(%i34) diff(beta_incomplete(a, b, z), z);
                                b - 1  a - 1
(%o34)                  (1 - z)      z
(%i35) integrate(beta_incomplete(a, b, z), z);
                b  a
         (1 - z)  z
(%o35) ----------- + beta_incomplete(a, b, z) z
           b + a
                                        a beta_incomplete(a, b, z)
                                      - --------------------------
                                                 b + a
(%i36) factor(diff(%, z));
(%o36)              beta_incomplete(a, b, z)
```

`beta_incomplete_regularized` (*a*, *b*, *z*) [Función]

Función beta incompleta regularizada A&S 6.6.2, definida como

```
beta_incomplete_regularized(a, b, z) =
                                      beta_incomplete(a, b, z)
                                      ------------------------
                                            beta(a, b)
```

Al igual que `beta_incomplete`, esta definición no es completa. Véase Funcións.wolfram.com para una definición completa de `beta_incomplete_regularized`.

`beta_incomplete_regularized` se simplifica para *a* o *b* entero positivo.

Para $z = 0$ y $realpart(a) > 0$, `beta_incomplete_regularized` se anula. Para *z=1* y $realpart(b) > 0$, `beta_incomplete_regularized` se reduce a 1.

Maxima evalúa `beta_incomplete_regularized` numéricamente para valores reales y complejos en forma decimal y big float.

Si `beta_expand` vale `true`, Maxima expande `beta_incomplete_regularized` para los argumentos $a + n$ o $a - n$, siendo n entero.

Maxima conoce las derivadas de `beta_incomplete_regularized` con respecto a las variables a, b y z, así como la integral respecto de la variable z.

Ejemplos:

Simplificación para a o b enteros positivos:

```
(%i1) beta_incomplete_regularized(2,b,z);
                                        b
(%o1)                      1 - (1 - z)  (b z + 1)

(%i2) beta_incomplete_regularized(a,2,z);
                                                a
(%o2)                        (a (1 - z) + 1) z

(%i3) beta_incomplete_regularized(3,2,z);
                                                3
(%o3)                        (3 (1 - z) + 1) z
```

Simplificación para los valores $z = 0$ y $z = 1$:

```
(%i4) assume(a>0,b>0)$
(%i5) beta_incomplete_regularized(a,b,0);
(%o5)                                0
(%i6) beta_incomplete_regularized(a,b,1);
(%o6)                                1
```

Evaluación numérica, tanto con float (precisión doble) como big float (precisión arbitraria):

```
(%i7) beta_incomplete_regularized(0.12,0.43,0.9);
(%o7)                        .9114011367359802
(%i8) fpprec:32$
(%i9) beta_incomplete_regularized(0.12,0.43,0.9b0);
(%o9)              9.1140113673598075519946998779975b-1
(%i10) beta_incomplete_regularized(1+%i,3/3,1.5*%i);
(%o10)            .2865367499935403 %i - 0.122995963334684
(%i11) fpprec:20$
(%i12) beta_incomplete_regularized(1+%i,3/3,1.5b0*%i);
(%o12)     2.8653674999354036142b-1 %i - 1.2299596333468400163b-1
```

Expansión cuando `beta_expand` vale `true`:

```
(%i13) beta_incomplete_regularized(a+1,b,z);
                                                      b  a
                                                (1 - z)  z
(%o13) beta_incomplete_regularized(a, b, z) - ------------
                                                a beta(a, b)
(%i14) beta_incomplete_regularized(a-1,b,z);
```

```
(%o14) beta_incomplete_regularized(a, b, z)
                                                          b  a - 1
                                                     (1 - z)  z
                                                - ----------------------
                                                  beta(a, b) (b + a - 1)
```

Derivada e integral respecto de z:

```
(%i15) diff(beta_incomplete_regularized(a,b,z),z);
                                b - 1  a - 1
                          (1 - z)      z
(%o15)                    -------------------
                              beta(a, b)
(%i16) integrate(beta_incomplete_regularized(a,b,z),z);
(%o16) beta_incomplete_regularized(a, b, z) z
                                                                 b  a
                                                           (1 - z)  z
             a (beta_incomplete_regularized(a, b, z) - ------------)
                                                       a beta(a, b)
           - ------------------------------------------------------
                                     b + a
```

`beta_incomplete_generalized` (*a*, *b*, *z1*, *z2*) [Función]

La definición básica de la función beta incompleta generalizada es

The basic definition of the generalized incomplete beta function is

```
 z2
/
[          b - 1  a - 1
I   (1 - t)      t      dt
]
/
 z1
```

Maxima simplifica `beta_incomplete_regularized` para a y b enteros positivos.

Para $realpart(a) > 0$ y $z1 = 0$ o $z2 = 0$, Maxima reduce `beta_incomplete_generalized` a `beta_incomplete`. Para $realpart(b) > 0$ y $z1 = 1$ o *z2=1*, Maxima reduce a una expresión con `beta` y `beta_incomplete`.

Maxima evalúa `beta_incomplete_generalized` numéricamente para valores reales y complejos en forma decimal y big float.

Si `beta_expand` vale `true`, Maxima expande `beta_incomplete_generalized` para los argumentos $a + n$ y $a - n$, siendo n entero positivo.

Maxima conoce las derivadas de `beta_incomplete_generalized` con respecto a las variables *a*, *b*, *z1* y *z2*, así como la integral respecto de las variables *z1* y *z2*.

Ejemplos:

Maxima simplifica `beta_incomplete_generalized` para a y b enteros positivos:

```
(%i1) beta_incomplete_generalized(2,b,z1,z2);
```

```
                          b                        b
                (1 - z1)  (b z1 + 1) - (1 - z2)  (b z2 + 1)
(%o1)           ------------------------------------------
                                b (b + 1)

(%i2) beta_incomplete_generalized(a,2,z1,z2);
                                  a                       a
                (a (1 - z2) + 1) z2  - (a (1 - z1) + 1) z1
(%o2)           ------------------------------------------
                                a (a + 1)

(%i3) beta_incomplete_generalized(3,2,z1,z2);
                2      2                        2      2
          (1 - z1)  (3 z1  + 2 z1 + 1) - (1 - z2)  (3 z2  + 2 z2 + 1)
(%o3)     ------------------------------------------------------------
                                    12
```

Simplificación para los valores $z1 = 0$, $z2 = 0$, $z1 = 1$ o $z2 = 1$:

```
(%i4) assume(a > 0, b > 0)$
(%i5) beta_incomplete_generalized(a,b,z1,0);
(%o5)                    - beta_incomplete(a, b, z1)

(%i6) beta_incomplete_generalized(a,b,0,z2);
(%o6)                    - beta_incomplete(a, b, z2)

(%i7) beta_incomplete_generalized(a,b,z1,1);
(%o7)               beta(a, b) - beta_incomplete(a, b, z1)

(%i8) beta_incomplete_generalized(a,b,1,z2);
(%o8)               beta_incomplete(a, b, z2) - beta(a, b)
```

Evaluación numérica para argumentos reales, tanto con float (precisión doble) como big float (precisión arbitraria):

```
(%i9) beta_incomplete_generalized(1/2,3/2,0.25,0.31);
(%o9)                        .09638178086368676

(%i10) fpprec:32$
(%i10) beta_incomplete_generalized(1/2,3/2,0.25,0.31b0);
(%o10)               9.6381780863686935309170054689964b-2
```

Evaluación numérica para argumentos complejos, tanto con float (precisión doble) como big float (precisión arbitraria):

```
(%i11) beta_incomplete_generalized(1/2+%i,3/2+%i,0.25,0.31);
(%o11)           - .09625463003205376 %i - .003323847735353769
(%i12) fpprec:20$
(%i13) beta_incomplete_generalized(1/2+%i,3/2+%i,0.25,0.31b0);
(%o13)     - 9.6254630032054178691b-2 %i - 3.3238477353543591914b-3
```

Expansión para $a + n$ o $a - n$, siendo n entero positivo con `beta_expand` igual `true`:

```
(%i14) beta_expand:true$
(%i15) beta_incomplete_generalized(a+1,b,z1,z2);
                b   a           b   a
       (1 - z1)  z1  - (1 - z2)  z2
(%o15) -----------------------------
                   b + a
                             a beta_incomplete_generalized(a, b, z1, z2)
                          + ----------------------------------------
                                              b + a

(%i16) beta_incomplete_generalized(a-1,b,z1,z2);
       beta_incomplete_generalized(a, b, z1, z2) (- b - a + 1)
(%o16) -------------------------------------------------------
                                1 - a
                                          b   a - 1           b   a - 1
                                  (1 - z2)  z2      - (1 - z1)  z1
                                - -----------------------------------
                                                1 - a
```

Derivada respecto de la variable *z1* e integrales respecto de *z1* y *z2*:

```
(%i17) diff(beta_incomplete_generalized(a,b,z1,z2),z1);
                                      b - 1   a - 1
(%o17)                      - (1 - z1)      z1

(%i18) integrate(beta_incomplete_generalized(a,b,z1,z2),z1);
(%o18) beta_incomplete_generalized(a, b, z1, z2) z1
                                       + beta_incomplete(a + 1, b, z1)

(%i19) integrate(beta_incomplete_generalized(a,b,z1,z2),z2);
(%o19) beta_incomplete_generalized(a, b, z1, z2) z2
                                       - beta_incomplete(a + 1, b, z2)
```

`beta_expand` [Variable opcional]

Valor por defecto: false

Si `beta_expand` vale `true`, `beta(a,b)` y sus funciones relacionadas se expanden para argumentos del tipo $a + n$ o $a - n$, siendo n un número entero.

`beta_args_sum_to_integer` [Variable opcional]

Valor por defecto: false

Si `beta_args_sum_to_integer` vale `true`, Maxima simplifica `beta(a,b)` cuando la suma de los argumentos a y b sea un entero.

`psi` [*n*](*x*) [Función]

Es la derivada de `log (gamma (x))` de orden *n*+1, de tal manera que `psi[0](x)` es la primera derivada, `psi[1](x)` la segunda derivada y así sucesivamente.

En general, Maxima no sabe cómo calcular valores numéricos de `psi`, pero sí conoce el valor exacto para algunos argumentos racionales. Existen algunas variables globales para controlar en qué rangos racionales debe devolver `psi` resultados exactos, si ello es

posible. Véanse las descripciones de `maxpsiposint`, `maxpsinegint`, `maxpsifracnum` y `maxpsifracdenom`. En resumen, *x* debe alcanzar un valor entre `maxpsinegint` y `maxpsiposint`. Si el valor absoluto de la parte fraccional de *x* es racional y tiene un numerador menor que `maxpsifracnum` y un denominador menor que `maxpsifracdenom`, la función `psi` devolverá un valor exacto.

La función `bfpsi` del paquete `bffac` puede calcular valores numéricos.

`maxpsiposint` [Variable opcional]

Valor por defecto: 20

La variable `maxpsiposint` guarda el mayor valor positivo para el que `psi[n](x)` intentará calcular un valor exacto.

`maxpsinegint` [Variable opcional]

Valor por defecto: -10

La variable `maxpsinegint` guarda el menor valor negativo para el que `psi[n](x)` intentará calcular un valor exacto. Si *x* es menor que `maxnegint`, `psi[n](x)` no devolverá una respuesta simplificada, aunque supiese cómo hacerlo.

`maxpsifracnum` [Variable opcional]

Valor por defecto: 6

Sea *x* un número racional menor que la unidad de la forma `p/q`. Si `p` es mayor que `maxpsifracnum`, entonces `psi[`*n*`](x)` no devolverá una respuesta simplificada.

`maxpsifracdenom` [Variable opcional]

Valor por defecto: 6

Sea *x* un número racional menor que la unidad de la forma `p/q`. Si `q` es mayor que `maxpsifracnum`, entonces `psi[`*n*`](x)` no devolverá una respuesta simplificada.

`makefact` (*expr*) [Función]

Transforma las funciones `binomial`, `gamma` y `beta` que aparecen en *expr* en su notación factorial.

Véase también `makegamma`.

`numfactor` (*expr*) [Función]

Devuelve el factor numérico que multiplica a la expresión *expr*, la cual debe tener un único término.

```
(%i1) gamma (7/2);
                              15 sqrt(%pi)
(%o1)                         ------------
                                   8
(%i2) numfactor (%);
                                   15
(%o2)                              --
                                   8
```

15.5 Integral exponencial

La integral exponencial y sus funciones relacionadas se definen en el capítulo 5 de Abramowitz y Stegun, *Handbook of Mathematical Functions*.

`expintegral_e1 (z)` [Función]
La integral exponencial E1(z) (A&S 5.1.1)

`expintegral_ei (z)` [Función]
La integral exponencial Ei(z) (A&S 5.1.2)

`expintegral_li (z)` [Función]
La integral exponencial Li(z) (A&S 5.1.3)

`expintegral_e (n,z)` [Función]
La integral exponencial En(z) (A&S 5.1.4)

`expintegral_si (z)` [Función]
La integral exponencial Si(z) (A&S 5.2.1)

`expintegral_ci (z)` [Función]
La integral exponencial Ci(z) (A&S 5.2.2)

`expintegral_shi (z)` [Función]
La integral exponencial Shi(z) (A&S 5.2.3)

`expintegral_chi (z)` [Función]
La integral exponencial Chi(z) (A&S 5.2.4)

`expintrep` [Option variable]
Valor por defecto: false

Transforma la representación de la integral exponencial en términos de las funciones `gamma_incomplete`, `expintegral_e1`, `expintegral_ei`, `expintegral_li`, `expintegral_trig` y `expintegral_hyp`.

`expintexpand` [Option variable]
Valor por defecto: false

Expande la integral exponencial E[n](z) para valores medios de la integral en términos de las funciones Erfc o Erf y para positivos enteros en términos de Ei .

15.6 Función de error

La función de error y sus asociadas se definen en el capítulo 7 de Abramowitz y Stegun, *Handbook of Mathematical Functions*.

`erf (z)` [Función]
Función de error erf(z) (A&S 7.1.1)

Véase también `erfflag`.

`erfc (z)` [Función]
Complemento de la función de error erfc(z) (A&S 7.1.2)

`erfc(z) = 1-erf(z)`

`erfi (z)` [Función]
Función de error imaginaria.

```
erfi(z) = -%i*erf(%i*z)
```

`erf_generalized (z1,z2)` [Función]
Función de error generalizada Erf(z1,z2)

`fresnel_c (z)` [Función]
Integral de Fresnel `C(z) = integrate(cos((%pi/2)*t^2),t,0,z)`. (A&S 7.3.1)

La simplificación `fresnel_c(-x) = -fresnel_c(x)` se aplica cuando la variable global `trigsign` vale `true`.

La simplificación `fresnel_s(%i*x) = -%i*fresnel_s(x)` se aplica cuando la variable global `%iargs` vale `true`.

Véanse también `erf_representation` y `hypergeometric_representation`.

`fresnel_s (z)` [Función]
Integral de Fresnel S(z) = integrate(sin((%pi/2)*t^2),t,0,z). (A&S 7.3.2)

La simplificación fresnel_s(-x) = -fresnel_s(x) se aplica cuando la variable global `trigsign` vale `true`.

La simplificación fresnel_s(%i*x) = %i*fresnel_s(x) se aplica cuando la variable global `%iargs` vale `true`.

Véanse también `erf_representation` y `hypergeometric_representation`.

`erf_representation` [Variable opcional]
Valor por defecto: false

Cuando valga `true` erfc, erfi, erf_generalized, fresnel_s y fresnel_c se transforman a erf.

`hypergeometric_representation` [Variable opcional]
Valor por defecto: false

Permite obtener la representación hipergeométrica de las funciones fresnel_s y fresnel_c.

15.7 Funciones de Struve

Las funciones de Struve se definen en el capítulo 12 de Abramowitz y Stegun, *Handbook of Mathematical Functions*.

`struve_h (v, z)` [Función]
Función H de Struve de orden v y argumento z, (A&S 12.1.1).

`struve_l (v, z)` [Función]
Función L modificada de Struve de orden v y argumento z, (A&S 12.2.1).

15.8 Funciones hipergeométricas

Las funciones hipergeométricas se definen en los capítulos 13 y 15 de Abramowitz y Stegun, *Handbook of Mathematical Functions.*

Maxima tiene un soporte limitado sobre estas funciones, que pueden aparecer en resultados devueltos por `hgfred`.

`%m` [*k*,*u*] (*z*) [Función]
Función M de Whittaker `M[k,u](z) = exp(-z/2)*z^(1/2+u)*M(1/2+u-k,1+2*u,z)`. (A&S 13.1.32)

`%w` [*k*,*u*] (*z*) [Función]
Función W de Whittaker. (A&S 13.1.33)

`%f` [*p*,*q*] ([a],[*b*],*z*) [Función]
Es la función hipergeométrica pFq(a1,a2,..ap;b1,b2,..bq;z), donde `a` es una lista de longitud `p` y `b` otra lista de longitud `q`.

`hypergeometric` ([*a1*, ..., *ap*],[*b1*, ... ,*bq*], *x*) [Función]
Es la función hipergeométrica. A diferencia de la función hipergeométrica `%f` de Maxima, la función `hypergeometric` es simplificadora; además, `hypergeometric` soporta la evaluación en doble (float) y gran (bigfloat) precisión. La evaluación numérica fuera del círculo unidad no está en general soportada, pero sí en el caso de la función hipergeométrica de Gauss, cuando $p = 2$ y $q = 1$.

Si la variable opcional `expand_hypergeometric` vale `true`, (el valor por defecto es `false`) y uno de los argumentos entr `a1` y `ap` es entero negativo (caso polinomial), entonces `hypergeometric` devuelve un polinomio expandido.

Ejemplos:

```
(%i1)  hypergeometric([],[],x);
(%o1) %e^x
```

Los polinomios se expanden automáticamente cuando `expand_hypergeometric` vale `true`.

```
(%i2) hypergeometric([-3],[7],x);
(%o2) hypergeometric([-3],[7],x)

(%i3) hypergeometric([-3],[7],x), expand_hypergeometric : true;
(%o3) -x^3/504+3*x^2/56-3*x/7+1
```

Se soporta la evaluación en doble (float) y gran (bigfloat) precisión:

```
(%i4) hypergeometric([5.1],[7.1 + %i],0.42);
(%o4)       1.346250786375334 - 0.0559061414208204 %i
(%i5) hypergeometric([5,6],[8], 5.7 - %i);
(%o5)     .007375824009774946 - .001049813688578674 %i
(%i6) hypergeometric([5,6],[8], 5.7b0 - %i), fpprec : 30;
(%o6) 7.37582400977494674506442010824b-3
                            - 1.04981368857867315858055393376b-3 %i
```

15.9 Funciones de cilindro parabólico

Las funciones de cilindro parabólico se definen en el capítulo 19 de Abramowitz y Stegun, *Handbook of Mathematical Functions.*

Maxima tiene un soporte limitado sobre estas funciones, que pueden aparecer en resultados devueltos por `hgfred`.

`parabolic_cylinder_d` (*v*, *z*) [Función]

Función de cilindro parabólico `parabolic_cylinder_d(v,z)`. (A&s 19.3.1)

15.10 Funciones y variables para las funciones especiales

`specint` (*exp*(- *s***t*) * *expr*, *t*) [Función]

Calcula la transformada de Laplace de *expr* respecto de la variable *t*. El integrando *expr* puede contener funciones especiales.

La función `specint` admite las funciones especiales siguientes: la gamma incompleta, las funciones de error (pero no `erfi`, siendo sencillo transformar `erfi` en la función de error `erf`), integrales exponenciales, funciones de Bessel (incluidos productos de funciones de Bessel), funciones de Hankel, de Hermite y los polinomios de Laguerre.

Además, `specint` también admite la función hipergeométrica `%f[p,q]([],[],z)`, la función de Whittaker de primera especie `%m[u,k](z)` y la de segunda especie `%w[u,k](z)`.

El resultado puede darse en términos de funciones especiales y es posible que incluya también funciones hipergeométricas sin simplificar.

Cuando `laplace` es incapaz de calcular la transformada de Laplace, entonces llama a la función `specint`. Puesto que `laplace` tiene programadas más reglas para calcular transformadas de Laplace, es preferible utilizar `laplace` en lugar de `specint`.

La ejecución de `demo(hypgeo)` muestra algunos ejemplos de transformadas de Laplace calculadas con `specint`.

Ejemplos:

```
(%i1) assume (p > 0, a > 0)$
(%i2) specint (t^(1/2) * exp(-a*t/4) * exp(-p*t), t);
                              sqrt(%pi)
(%o2)                         ------------
                                     a 3/2
                              2 (p + -)
                                     4
(%i3) specint (t^(1/2) * bessel_j(1, 2 * a^(1/2) * t^(1/2))
              * exp(-p*t), t);
                                       - a/p
                            sqrt(a) %e
(%o3)                       ---------------
                                  2
                                 p
```

Ejemplos para integrales exponenciales:

```
(%i4) assume(s>0,a>0,s-a>0)$
```

```
(%i5) ratsimp(specint(%e^(a*t)
                      *(log(a)+expintegral_e1(a*t))*%e^(-s*t),t));
                             log(s)
(%o5)                        ------
                             s - a
(%i6) logarc:true$

(%i7) gamma_expand:true$

radcan(specint((cos(t)*expintegral_si(t)
                     -sin(t)*expintegral_ci(t))*%e^(-s*t),t));
                             log(s)
(%o8)                        ------
                              2
                             s  + 1
ratsimp(specint((2*t*log(a)+2/a*sin(a*t)
                      -2*t*expintegral_ci(a*t))*%e^(-s*t),t));
                               2    2
                          log(s  + a )
(%o9)                     ------------
                                2
                               s
```

Resultados cuando se utiliza la expansión de `gamma_incomplete` y se cambia la representación de `expintegral_e1`:

```
(%i10) assume(s>0)$
(%i11) specint(1/sqrt(%pi*t)*unit_step(t-k)*%e^(-s*t),t);
                                          1
                              gamma_incomplete(-, k s)
                                          2
(%o11)                        ------------------------
                                 sqrt(%pi) sqrt(s)

(%i12) gamma_expand:true$
(%i13) specint(1/sqrt(%pi*t)*unit_step(t-k)*%e^(-s*t),t);
                               erfc(sqrt(k) sqrt(s))
(%o13)                         ---------------------
                                      sqrt(s)

(%i14) expintrep:expintegral_e1$
(%i15) ratsimp(specint(1/(t+a)^2*%e^(-s*t),t));
                               a s
                         a s %e    expintegral_e1(a s) - 1
(%o15)                 - ---------------------------------
                                        a
```

`hgfred` (`a`, *b*, *t*) [Función]

Simplifica la función hipergeométrica generalizada en términos de otras funciones más sencillas. *a* es una lista de parámetros del numerador y *b* lo es de parámetros del denominador.

En caso de que `hgfred` no pueda simplificar la función hipergeométrica devolverá una expresión de la forma `%f[p,q]([a], [b], x)`, siendo *p* el número de elementos de *a* y *q* el de *b*. Esta es la función hipergeométrica generalizada `pFq`.

```
(%i1) assume(not(equal(z,0)));
(%o1)                          [notequal(z, 0)]
(%i2) hgfred([v+1/2],[2*v+1],2*%i*z);

                    v/2                                 %i z
                   4    bessel_j(v, z) gamma(v + 1) %e
(%o2)              ---------------------------------------
                                      v
                                     z
(%i3) hgfred([1,1],[2],z);

                                   log(1 - z)
(%o3)                            - ----------
                                       z
(%i4) hgfred([a,a+1/2],[3/2],z^2);

                              1 - 2 a          1 - 2 a
                       (z + 1)        - (1 - z)
(%o4)                  -------------------------------
                                2 (1 - 2 a) z
```

Tal como muestra el siguiente ejemplo, puede ser de utilidad cargar también el paquete `orthopoly`. Nótese que *L* es el polinomio generalizado de Laguerre.

```
(%i5) load("orthopoly")$
(%i6) hgfred([-2],[a],z);

                                    (a - 1)
                                 2 L       (z)
                                    2
(%o6)                            -------------
                                   a (a + 1)
(%i7) ev(%);

                                  2
                                 z        2 z
(%o7)                          --------- - --- + 1
                               a (a + 1)    a
```

`lambert_w` (*z*) [Función]

Rama principal de la función W de Lambert, solución de la ecuación `z = W(z) * exp(W(z))`. (DLMF 4.13)

`generalized_lambert_w` (*k*, *z*) [Función]

k-ésima rama de la función W de Lambert's, W(z), solución de `z = W(z) * exp(W(z))`. (DLMF 4.13)

La rama principal, representada por Wp(z) en DLMF, es `lambert_w(z) = generalized_lambert_w(0,z)`.

La otra rama con valores reales, representada por Wm(z) en DLMF, es `generalized_lambert_w(-1,z)`.

`nzeta` (*z*) [Función]

Función de dispersión del plasma. `nzeta(z) = %i*sqrt(%pi)*exp(-z^2)*(1-erf(-%i*z))`

`nzetar` (*z*) [Función]

Devuelve `realpart(nzeta(z))`.

`nzetai` (*z*) [Función]

Devuelve `imagpart(nzeta(z))`.

16 Funciones elípticas

16.1 Introducción a las funciones e integrales elípticas

Maxima da soporte para las funciones elípticas jacobianas y para las integrales elípticas completas e incompletas. Esto incluye la manipulación simbólica de estas funciones y su evaluación numérica. Las definiciones de estas funciones y de muchas de sus propiedades se pueden encontrar en Abramowitz y Stegun, capítulos 16–17, que es la fuente principal utilizada para su programación en Maxima, aunque existen algunas diferencias.

En particular, todas las funciones e integrales elípticas utilizan el parámero m en lugar del módulo k o del ángulo $alfa$. Esta es una de las diferencias con Abramowitz y Stegun, que utilizan el ángulo para las funciones elípticas. Las siguientes relaciones son válidas:

$$m = k^2$$

y

$$k = \sin \alpha$$

Las funciones e integrales elípticas en Maxima tienen como objetivo primordial dar soporte al cálculo simbólico, de ahí que también estén incluidas la mayoría de las derivadas e integrales asociadas a estas funciones. No obstante lo anterior, si los argumentos dados a las funciones son decimales en coma flotante, los resultados también serán decimales.

Sin embargo, la mayoría de las propiedades no realacionadas con las derivadas de las funciones e integrales elípticas todavía no han sido programadas en Maxima.

Algunos ejemplos de funciones 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)
```

Algunos ejemplos de integrales 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))
```

El paquete para funciones e integrales elípticas fue programado por Raymond Toy. Se distribuye, igual que Maxima, bajo la General Public License (GPL).

16.2 Funciones y variables para funciones elípticas

`jacobi_sn` (*u*, *m*) [Función]

Función elíptica jacobiana $sn(u, m)$.

`jacobi_cn` (*u*, *m*) [Función]

Función elíptica jacobiana $cn(u, m)$.

`jacobi_dn` (*u*, *m*) [Función]

Función elíptica jacobiana $dn(u, m)$.

`jacobi_ns (u, m)` [Función]
Función elíptica jacobiana $ns(u, m) = 1/sn(u, m)$.

`jacobi_sc (u, m)` [Función]
Función elíptica jacobiana $sc(u, m) = sn(u, m)/cn(u, m)$.

`jacobi_sd (u, m)` [Función]
Función elíptica jacobiana $sd(u, m) = sn(u, m)/dn(u, m)$.

`jacobi_nc (u, m)` [Función]
Función elíptica jacobiana $nc(u, m) = 1/cn(u, m)$.

`jacobi_cs (u, m)` [Función]
Función elíptica jacobiana $cs(u, m) = cn(u, m)/sn(u, m)$.

`jacobi_cd (u, m)` [Función]
Función elíptica jacobiana $cd(u, m) = cn(u, m)/dn(u, m)$.

`jacobi_nd (u, m)` [Función]
Función elíptica jacobiana $nc(u, m) = 1/cn(u, m)$.

`jacobi_ds (u, m)` [Función]
Función elíptica jacobiana $ds(u, m) = dn(u, m)/sn(u, m)$.

`jacobi_dc (u, m)` [Función]
Función elíptica jacobiana $dc(u, m) = dn(u, m)/cn(u, m)$.

`inverse_jacobi_sn (u, m)` [Función]
Inversa de la función elíptica jacobiana $sn(u, m)$.

`inverse_jacobi_cn (u, m)` [Función]
Inversa de la función elíptica jacobiana $cn(u, m)$.

`inverse_jacobi_dn (u, m)` [Función]
Inversa de la función elíptica jacobiana $dn(u, m)$.

`inverse_jacobi_ns (u, m)` [Función]
Inversa de la función elíptica jacobiana $ns(u, m)$.

`inverse_jacobi_sc (u, m)` [Función]
Inversa de la función elíptica jacobiana $sc(u, m)$.

`inverse_jacobi_sd (u, m)` [Función]
Inversa de la función elíptica jacobiana $sd(u, m)$.

`inverse_jacobi_nc (u, m)` [Función]
Inversa de la función elíptica jacobiana $nc(u, m)$.

`inverse_jacobi_cs (u, m)` [Función]
Inversa de la función elíptica jacobiana $cs(u, m)$.

`inverse_jacobi_cd (u, m)` [Función]
Inversa de la función elíptica jacobiana $cd(u, m)$.

`inverse_jacobi_nd` (*u*, *m*) [Función]
Inversa de la función elíptica jacobiana $nc(u, m)$.

`inverse_jacobi_ds` (*u*, *m*) [Función]
Inversa de la función elíptica jacobiana $ds(u, m)$.

`inverse_jacobi_dc` (*u*, *m*) [Función]
Inversa de la función elíptica jacobiana $dc(u, m)$.

16.3 Funciones y variables para integrales elípticas

`elliptic_f` (*phi*, *m*) [Función]
Integral elíptica incompleta de primera especie, definida como

$$\int_0^{\phi} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}$$

Véanse también `elliptic_e` y `elliptic_kc`.

`elliptic_e` (*phi*, *m*) [Función]
Integral elíptica incompleta de segunda especie, definida como

$$\int_0^{\phi} \sqrt{1 - m\sin^2\theta}\, d\theta$$

Véanse también `elliptic_e` y `elliptic_ec`.

`elliptic_eu` (*u*, *m*) [Función]
Integral elíptica incompleta de segunda especie, definida como

$$\int_0^{u} \mathrm{dn}(v, m)^2 dv = \int_0^{\tau} \sqrt{\frac{1 - mt^2}{1 - t^2}}\, dt$$

donde $\tau = \mathrm{sn}(u, m)$.

Esto se relaciona con `elliptic_e` mediante

$$E(u, m) = E(\phi, m)$$

donde $\phi = \sin^{-1}\mathrm{sn}(u, m)$.

Véase también `elliptic_e`.

`elliptic_pi` (*n*, *phi*, *m*) [Función]
Integral elíptica incompleta de tercera especie, definida como

$$\int_0^{\phi} \frac{d\theta}{(1 - n\sin^2\theta)\sqrt{1 - m\sin^2\theta}}$$

Maxima sólo conoce la derivada respecto de *phi*.

`elliptic_kc` (`m`) [Función]

Integral elíptica completa de primera especie, definida como

$$\int_0^{\frac{\pi}{2}} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}$$

Para algunos valores de m, se conoce el valor de la integral en términos de la función *Gamma*. Hágase uso de `makegamma` para realizar su cálculo.

`elliptic_ec` (`m`) [Función]

Integral elíptica completa de segunda especie, definida como

$$\int_0^{\frac{\pi}{2}} \sqrt{1 - m\sin^2\theta} d\theta$$

Para algunos valores de m, se conoce el valor de la integral en términos de la función *Gamma*. Hágase uso de `makegamma` para realizar su cálculo.

17 Límites

17.1 Funciones y variables para límites

`lhospitallim` [Variable optativa]

Valor por defecto: 4

Es el número máximo de veces que la regla de L'Hopital es aplicada en la función `limit`, evitando bucles infinitos al iterar la regla en casos como `limit (cot(x)/csc(x), x, 0)`.

`limit` (*expr*, *x*, *val*, *dir*) [Función]
`limit` (*expr*, *x*, *val*) [Función]
`limit` (*expr*) [Función]

Calcula el límite de *expr* cuando la variable real *x* se aproxima al valor *val* desde la dirección *dir*. El argumento *dir* puede ser el valor `plus` para un límite por la derecha, `minus` para un límite por la izquierda o simplemente se omite para indicar un límite en ambos sentidos.

La función `limit` utiliza los símbolos especiales siguientes: `inf` (más infinito) y `minf` (menos infinito). En el resultado también puede hacer uso de `und` (indefinido), `ind` (indefinido pero acotado) y `infinity` (infinito complejo).

`infinity` (infinito complejo) es el resultado que se obtiene cuando el límite del módulo de la expresión es infinito positivo, pero el propio límite de la expresión no es infinito positivo ni negativo. Esto sucede, por ejemplo, cuando el límite del argumento complejo es una constante, como en `limit(log(x), x, minf)`, o cuando el argumento complejo oscila, como en `limit((-2)^x, x, inf)`, o en aquellos casos en los que el argumento complejo es diferente por cualquiera de los lados de un límite, como en `limit(1/x, x, 0)` o `limit(log(x), x, 0)`.

La variable `lhospitallim` guarda el número máximo de veces que la regla de L'Hopital es aplicada en la función `limit`, evitando bucles infinitos al iterar la regla en casos como `limit (cot(x)/csc(x), x, 0)`.

Si la variable `tlimswitch` vale `true`, hará que la función `limit` utilice desarrollos de Taylor siempre que le sea posible.

La variable `limsubst` evita que la función `limit` realice sustituciones sobre formas desconocidas, a fin de evitar fallos tales como que `limit (f(n)/f(n+1), n, inf)` devuelva 1. Dándole a `limsubst` el valor `true` se permitirán tales sustituciones.

La función `limit` con un solo argumento se utiliza frecuentemente para simplificar expresiones constantes, como por ejemplo `limit (inf-1)`.

La instrucción `example (limit)` muestra algunos ejemplos.

Para información sobre el método utilizado véase Wang, P., "Evaluation of Definite Integrals by Symbolic Manipulation", Ph.D. thesis, MAC TR-92, October 1971.

`limsubst` [Variable optativa]

Valor por defecto: `false`

La variable `limsubst` evita que la función `limit` realice sustituciones sobre formas desconocidas, a fin de evitar fallos tales como que `limit (f(n)/f(n+1), n, inf)` devuelva 1. Dándole a `limsubst` el valor `true` se permitirán tales sustituciones.

`tlimit` (*expr*, *x*, *val*, *dir*) [Función]
`tlimit` (*expr*, *x*, *val*) [Función]
`tlimit` (*expr*) [Función]

Calcula el límite del desarrollo de Taylor de la expresión `expr` de variable `x` en el punto `val` en la dirección `dir`.

`tlimswitch` [Variable optativa]

Valor por defecto: `true`

Si `tlimswitch` vale `true`, la función `limit` utilizará un desarrollo de Taylor si el límite de la expresión dada no se puede calcular directamente. Esto permite el cálculo de límites como `limit(x/(x-1)-1/log(x),x,1,plus)`. Si `tlimswitch` vale `false` y el límite de la expresión no se puede calcular directamente, la función `limit` devolverá una expresión sin evaluar.

18 Diferenciación

18.1 Funciones y variables para la diferenciación

`antid` (*expr*, *x*, *u*(*x*)) [Función]

Devuelve una lista con dos elementos, de manera que se pueda calcular la antiderivada de *expr* respecto de *x* a partir de la lista. La expresión *expr* puede contener una función no especificada *u* y sus derivadas.

Sea *L* la lista con dos elementos que devuelve la función `antid`. Entonces, `L[1] + 'integrate (L[2], x)` es una antiderivada de *expr* con respecto a *x*.

Si la ejecución de `antid` resulta exitosa, el segundo elemento de la lista retornada es cero. En caso contrario, el segundo elemento es distinto de cero y el primero puede ser nulo o no. Si `antid` no es capaz de hacer ningún progreso, el primer elemento es nulo y el segundo no nulo.

Es necesario ejecutar `load ("antid")` para cargar esta función. El paquete `antid` define también las funciones `nonzeroandfreeof` y `linear`.

La función `antid` está relacionada con `antidiff` como se indica a continuación. Sea *L* la lista devuelta por la función `antid`. Entonces, el resultado de `antidiff` es igual a `L[1] + 'integrate (L[2], x)`, donde *x* es la variable de integración.

Ejemplos:

```
(%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*)) [Función]

Devuelve la antiderivada de *expr* respecto de *x*. La expresión *expr* puede contener una función no especificada *u* y sus derivadas.

Cuando `antidiff` se ejecuta con éxito, la expresión resultante no tiene símbolos integrales (esto es, no tiene referencias a la función `integrate`). En otro caso, `antidiff` devuelve una expresión que se encuentra total o parcialmente bajo el signo de integración. Si `antidiff` no puede ralizar ningún progreso, el valor devuelto se encuentra completamente bajo la integral.

Es necesario ejecutar `load ("antid")` para cargar esta función. El paquete `antid` define también las funciones `nonzeroandfreeof` y `linear`.

La función `antidiff` está relacionada con `antid` como se indica a continuación. Sea *L* la lista de dos elementos que devuelve `antid`. Entonces, el valor retornado por `antidiff` es igual a `L[1] + 'integrate (L[2], x)`, donde *x* es la variable de integración.

Ejemplos:

```
(%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
                     /
```

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

Evalúa la expresión *expr* asignando a las variables los valores especificados para ellas en la lista de ecuaciones `[`*eqn_1*`, ..., `*eqn_n*`]` o en la ecuación simple *eqn*.

Si una subexpresión depende de cualquiera de las variables para la cual se especifica un valor, pero no puede ser evaluado, entonces `at` devuelve una forma nominal.

La función `at` realiza múltiples sustituciones en serie, no en paralelo.

Véase también `atvalue`. Para otras funciones que también llevan a cabo sustituciones, consúltense `subst` y `ev`.

Ejemplos:

```
(%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
```

`atomgrad` [Propiedad]

La propiedad `atomgrad` es asignada por `gradef`.

`atvalue` (*expr*, [*x_1* = *a_1*, ..., *x_m* = *a_m*], *c*) [Función]
`atvalue` (*expr*, *x_1* = *a_1*, *c*) [Función]

Asigna el valor *c* a *expr* en el punto `x = a`.

La expresión *expr* es una función del tipo `f(`*x_1*`, ..., `*x_m*`)`, o una derivada, `diff (f(`*x_1*`, ..., `*x_m*`), `*x_1*`, `*n_1*`, ..., `*x_n*`, `*n_m*`)` en la que aparecen los argumentos de la función de forma explícita. Los símbolos *n_i* se refieren al orden de diferenciación respecto de *x_i*.

El punto en el que `atvalue` establece el valor se especifica mediante la lista de ecuaciones `[x_1 = a_1, ..., x_m = a_m]`. Si hay una única variable *x_1*, la ecuación puede escribirse sin formar parte de una lista.

La llamada `printprops ([f_1, f_2, ...], atvalue)` muestra los valores asignados por `atvalue` a las funciones `f_1`, `f_2`, ... La llamada `printprops (f, atvalue)` muestra los valores asignados por `atvalue` a la función *f*. La llamada `printprops (all, atvalue)` muestra los valores asignados por `atvalue` a todas las funciones.

Los símbolos `@1`, `@2`, ... representan las variables *x_1*, *x_2*, ... cuando se muestran los valores asignados por `atvalue`.

La función `atvalue` evalúa sus argumentos y devuelve *c*, el valor asignado.

Ejemplos:

```
(%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` - [Función]

El cálculo exterior de formas diferenciales es una herramienta básica de la geometría diferencial desarrollada por Elie Cartan, teniendo importantes aplicaciones en la teoría de ecuaciones diferenciales en derivadas parciales. El paquete `cartan` dispone de las funciones `ext_diff` y `lie_diff`, así como de los operadores `~` (producto exterior) y `|` (contracción de una forma con un vector). La orden `demo (tensor)` permite ver una breve descripción de estas instrucciones, junto con ejemplos.

El paquete `cartan` fue escrito por F.B. Estabrook y H.D. Wahlquist.

`del (x)` [Función]

La expresión `del (x)` representa el diferencial de la variable x.

La función `diff` devuelve una expresión que contiene a `del` si no se ha especificado una variable independiente. En este caso, el valor retornado es el llamado "diferencial total".

Ejemplos:

```
(%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)` [Función]

Es la función delta de Dirac.

En el estado actual de desarrollo de Maxima, sólo `laplace` reconoce la función `delta`.

Ejemplo:

```
(%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` [Variable del sistema]

Valor por defecto: `[]`

La variable `dependencies` es la lista de átomos que tienen algún tipo de dependencia funcional, asignada por `depends` o `gradef`. La lista `dependencies` es acumulativa: cada llamada a `depends` o `gradef` añade elementos adicionales.

Véanse `depends` y `gradef`.

`depends (f_1, x_1, ..., f_n, x_n)` [Función]

Declara dependencias funcionales entre variables con el propósito de calcular derivadas. En ausencia de una dependencia declarada, `diff (f, x)` devuelve cero. Si se declara `depends (f, x)`, `diff (f, x)` devuelve una derivada simbólica (esto es, una expresión con `diff`).

Cada argumento *f_1*, *x_1*, etc., puede ser el nombre de una variable, de un arreglo o una lista de nombres. Cada elemento de *f_i* (quizás un único elemento) se declara como dependiente de cada elemento de *x_i* (quizás también un único elemento). Si alguno de los *f_i* es el nombre de un arreglo o contiene el nombre de un arreglo, todos los elemento del arregl dependen de *x_i*.

La función `diff` reconoce dependencias indirectas establecidas por `depends` y aplica la regla de la cadena en tales casos.

La instrucción `remove (f, dependency)` borra todas las dependencias declaradas para *f*.

La función `depends` devuelve una lista con las dependencias que han sido establecidas. Las dependencias se añaden a la variable global `dependencies`. La función `depends` evalúa sus argumentos.

La función `diff` es la única instrucción de Maxima que reconoce las dependencias establecidas por `depends`. Otras funciones (`integrate`, `laplace`, etc.) solamente reconocen dependencias explícitamente representadas por sus argumentos. Por ejemplo, `integrate` no reconoce la dependencia de `f` respecto de `x` a menos que se represente explícitamente 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` [Variable optativa]

Valor por defecto: `false`

Si `derivabbrev` vale `true`, las derivadas simbólicas (esto es, expresiones con `diff`) se muestran como subíndices. En otro caso, las derivadas se muestran en la notación de Leibniz, `dy/dx`.

`derivdegree (`*expr*, *y*, *x*`)` [Función]

Devuelve el mayor grado de la derivada de la variable dependiente *y* respecto de la variable independiente *x* que aparece en *expr*.

Ejemplo:

```
(%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*) [Función]

Hace que las derivadas calculadas por la instrucción `ev` se calculen respecto de las variables indicadas.

`derivsubst` [Variable optativa]

Valor por defecto: `false`

Si `derivsubst` vale `true`, una sustitución no sintáctica del estilo `subst (x, 'diff (y, t), 'diff (y, t, 2))` devuelve `'diff (x, t)`.

`diff` (*expr*, *x_1*, *n_1*, ..., *x_m*, *n_m*) [Función]
`diff` (*expr*, *x*, *n*) [Función]
`diff` (*expr*, *x*) [Función]
`diff` (*expr*) [Función]

Devuelve la derivada o diferencial de *expr* respecto de alguna o de todas las variables presentes en *expr*.

La llamada `diff (expr, x, n)` devuelve la *n*-esima derivada de *expr* respecto de *x*.

La llamada `diff (expr, x_1, n_1, ..., x_m, n_m)` devuelve la derivada parcial de *expr* con respecto de *x_1*, ..., *x_m*. Equivale a `diff (... (diff (expr, x_m, n_m) ...), x_1, n_1)`.

La llamada `diff (expr, x)` devuelve la primera derivada de *expr* respecto de la variable *x*.

La llamada `diff (expr)` devuelve el diferencial total de *expr*, esto es, la suma de las derivadas de *expr* respecto de cada una de sus variables, multiplicadas por el diferencial `del` de cada una de ellas.

La forma nominal de `diff` es necesaria en algunos contextos, como para definir ecuaciones diferenciales. En tales casos, `diff` puede ir precedida de un apóstrofo (como `'diff`) para evitar el cálculo de la derivada.

Si `derivabbrev` vale `true`, las derivadas se muestran como subíndices. En otro caso, se muestran en la notación de Leibniz, `dy/dx`.

Ejemplos:

```
(%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 el paquete sobre tensores se han introducido las siguientes modificaciones:

(1) Las derivadas de los objetos indexados en *expr* tendrán las variables *x_i* añadidas como argumentos adicionales. Entonces se ordenarán todos los índices de derivadas.

(2) Las *x_i* pueden ser enteros entre 1 hasta el valor de la variable `dimension` [valor por defecto: 4]. Esto hará que la diferenciación sea llevada a cabo con respecto al *x_i*-ésimo número de la lista `coordinates`, la cual debería contener una lista con los nombres de las coordenadas, por ejemplo, `[x, y, z, t]`. Si `coordinates` es una variableis atómica, entonces esa variable será utilizada como variable de diferenciación. Se permite la utilización de arreglos con los nombres de las coordenadas o nombres con subíndices, como `X[1]`, `X[2]`, ... to be used. Si a `coordinates` no se le ha asignado ningún valor, entonces las variables serán tratadas como se ha indicado en (1).

`diff` [Símbolo especial]

Si el nombre `diff` está presente en una llamada a la función `ev` en modo `evflag`, entonces se calculan todas las derivadas presentes en `expr`.

`express` (*expr*) [Función]

Transforma los nombres de los operadores diferenciales en expresiones que contienen derivadas parciales. Los operadores reconocidos por la función `express` son: `grad` (gradiente), `div` (divergencia), `curl` (rotacional), `laplacian` (laplaciano) y `~` (producto vectorial).

Las derivadas simbólicas (es decir, las que incluyen la forma nominal `diff`) que aparecen en la expresión devuelta por `express`, se pueden calcular pasándole a `ev` el argumento `diff`, o escribiéndolo directamente en la línea de comandos. En este contexto, `diff` actúa como `evfun`.

Es necesario ejecutar `load ("vect")` para cargar esta función.

Ejemplos:

```
(%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*) [Función]
`gradef` (`a`, *x*, *expr*) [Función]

Define las derivadas parciales, o componentes del gradiente, de la función *f* o variable *a*.

La llamada `gradef` (*f*(*x_1*, ..., *x_n*), *g_1*, ..., *g_m*) define `df/dx_i` como *g_i*, donde *g_i* es una expresión; *g_i* puede ser una llamada a función, pero no el nombre de una función. El número de derivadas parciales *m* puede ser menor que el número

de argumentos *n*, en cuyo caso las derivadas se definen solamente con respecto a *x_1*, ..., *x_m*.

La llamada `gradef (a, x, expr)` define la derivada de la variable *a* respecto de *x* en *expr*. Con esto se establece la dependencia de *a* respecto de *x* a través de `depends (a, x)`.

El primer argumento `f(x_1, ..., x_n)` o *a* no se evalúa, pero sí lo hacen el resto de argumentos *g_1*, ..., *g_m*. La llamada a `gradef` devuelve la función o variable para la que se define la derivada parcial.

La instrucción `gradef` puede redefinir las derivadas de las funciones propias de Maxima. Por ejemplo, `gradef (sin(x), sqrt (1 - sin(x)^2))` redefine la derivada de `sin`.

La instrucción `gradef` no puede definir derivadas parciales de funciones subindicadas.

La llamada `printprops ([f_1, ..., f_n], gradef)` muestra las derivadas parciales de las funciones *f_1*, ..., *f_n*, tal como las definió `gradef`.

La llamada `printprops ([a_n, ..., a_n], atomgrad)` muestra las derivadas parciales de las variables *a_n*, ..., *a_n*, tal como las definió `gradef`.

La variable `gradefs` contiene la lista de las funciones para las que se han definido derivadas parciales con la instrucción `gradef`, pero no incluye las variables para las que se han definido las derivadas parciales.

Los gradientes son necesarios cuando una función no se conoce explícitamente pero sí sus primeras derivadas y es necesario calcular las derivadas de orden mayor.

`gradefs` [Variable del sistema]

Valor por defecto: `[]`

La variable `gradefs` contiene la lista de las funciones para las que se han definido derivadas parciales con la instrucción `gradef`, pero no incluye las variables para las que se han definido las derivadas parciales.

`laplace (expr, t, s)` [Función]

Calcula la transformada de Laplace de *expr* con respecto de la variable *t* y parámetro de transformación *s*.

La función `laplace` reconoce en *expr* las funciones `delta`, `exp`, `log`, `sin`, `cos`, `sinh`, `cosh` y `erf`, así como `derivative`, `integrate`, `sum` y `ilt`. Si `laplace` no encuentra una transformada, entonces llama a la función `specint`, la cual puede encontrar la transformada de Laplace de expresiones con funciones especiales, tales como las de Bessel. `specint` también puede manipular la función `unit_step`. Véase `specint` para más información.

Cuando tampoco `specint` sea capaz de encontrar una solución, se devolverá una forma nominal.

La función `laplace` reconoce integrales de convolución de la forma `integrate (f(x) * g(t - x), x, 0, t)`, no pudiendo reconocer otros tipos de convoluciones.

Las relaciones funcionales se deben representar explícitamente en *expr*; las relaciones implícitas establecidas por `depends` no son reconocidas. Así, si *f* depende de *x* y *y*, `f (x, y)` debe aparecer en *expr*.

Véase también `ilt`, la transformada inversa de Laplace.

Ejemplos:

```
(%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
(%i5) assume(a>0)$
(%i6) laplace(gamma_incomplete(a,t),t,s),gamma_expand:true;
                                              - a - 1
                         gamma(a)   gamma(a) s
(%o6)                    -------- - -----------------
                            s              1     a
                                         (- + 1)
                                           s
(%i7) factor(laplace(gamma_incomplete(1/2,t),t,s));
                                             s + 1
                      sqrt(%pi) (sqrt(s) sqrt(-----) - 1)
                                               s
(%o7)                 -----------------------------------
                                 3/2      s + 1
                                s    sqrt(-----)
                                            s
(%i8) assume(exp(%pi*s)>1)$
(%i9) laplace(sum((-1)^n*unit_step(t-n*%pi)*sin(t),n,0,inf),t,s),simpsum;
                          %i                          %i
              ------------------------ - ------------------------
                                - %pi s                     - %pi s
              (s + %i) (1 - %e       )   (s - %i) (1 - %e       )
(%o9)         ------------------------------------------------------
                                    2
```

```
(%i9) factor(%);
                                        %pi s
                                       %e
(%o9)                        -------------------------------
                                                %pi s
                             (s - %i) (s + %i) (%e      - 1)
```

19 Integración

19.1 Introducción a la integración

Maxima tiene varias rutinas para calcular integrales. La función `integrate` hace uso de la mayor parte de ellas. También está el paquete `antid`, que opera con funciones no especificadas y sus derivadas. Para usos numéricos se dispone de la batería de integradores adaptativos de `QUADPACK`, como `quad_qag`, `quad_qags`, etc., que se describen en la sección `QUADPACK`. También se trabajan funciones hipergeométricas, véase `specint` para más detalles. En términos generales, Maxima sólo opera con funciones que son integrables en términos de funciones elementales, como las racionales, trigonométricas, logarítmicas, exponenciales, radicales, etc., y unas pocas extensiones de éstas, como la función de error o los dilogaritmos. No opera con integrales en términos de funciones desconocidas, como `g(x)` o `h(x)`.

19.2 Funciones y variables para integración

`changevar` (*expr*, *f*(*x*,*y*), *y*, *x*) [Función]

Hace el cambio de variable dado por *f(x,y)* = 0 en todas las integrales que aparecen en *expr* con la integración respecto de *x*. La nueva variable será *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
```

Si una expresión contiene formas nominales, como aquélla en la que aparece `'integrate` en el ejemplo, podrá ser evaluada por `ev` si se utiliza el término `nouns`. Por ejemplo, la expresión devuelta por `changevar` se puede evaluar haciendo `ev (%o3, nouns)`.

La función `changevar` también se puede utilizar para cambiar los índices de una suma o producto. Sin embargo, debe tenerse en cuenta que cuando se realiza un cambio en

una suma o producto, el mismo debe expresarse en términos de sumas, como `i = j+ ...`, no como una función de mayor grado.

Ejemplo:

```
(%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*) [Función]

Es una rutina para integrales dobles escrita en lenguaje Maxima y posteriormente traducida y compilada a código máquina. La instrucción `load ("dblint")` carga esta función. Utiliza el método de Simpson en las dos direcciones `x` e `y` para calcular

$$\int_a^b \int_{r(x)}^{s(x)} f\,(x,y)\; dy\, dx.$$

La función f debe ser una función traducida o compilada de dos variables, a la vez que r y s deben ser cada una de ellas una función traducida o compilada de una variable, mientras que a y b deben ser números en coma flotante. La rutina tiene dos variables globales que determinan el número de divisiones de los intervalos `x` e `y`: `dblint_x` y `dblint_y`, ambos con un valor por defecto de 10, pero que pueden cambiarse de forma independiente a otros valores enteros (hay `2*dblint_x+1` puntos a calcular en la dirección `x` y `2*dblint_y+1` en la dirección `y`). La rutina subdivide el eje X y luego para cada valor de X calcula primero `r(x)` y `s(x)`; entonces se subdivide el eje Y entre `r(x)` y `s(x)`, evaluándose la integral a lo largo del eje Y aplicando la regla de Simpson; a continuación, se evalúa la integral a lo largo del eje X utilizando también la regla de Simpson tomando como valores de función las integrales sobre Y. Este procedimiento puede ser numéricamente inestable por múltiples motivos, pero es razonablemente rápido: evítese su uso con funciones con grandes oscilaciones o que tengan singularidades. Las integrales del eje Y dependen de la proximidad de los límites `r(x)` y `s(x)`, de manera que si la distancia `s(x) - r(x)` varía rápidamente con X, puede dar lugar errores importantes debido a truncamientos de diferente amplitud en las integrales de Y. Se puede aumentar `dblint_x` y `dblint_y` al objeto de mejorar el recubrimiento de la región de integración, pero a costa del tiempo de cómputo. Es necesario que las funciones f, r y s estén traducidas o compiladas antes de utilizar

`dblint`, lo cual redundará en una mejora del tiempo de ejecución de varios órdenes de magnitud respecto de la ejecución de código interpretado.

`defint` (*expr*, *x*, `a`, *b*) [Función]

Intenta calcular una integral definida. La función `defint` es invocada por `integrate` cuando se especifican los límites de integración, por ejemplo `integrate (`*expr*`, `*x*`, `a`, `*b*`)`. Así, desde el punto de vista del usuario, es suficiente con utilizar `integrate`.

La función `defint` devuelve una expresión simbólica, bien sea el resultado calculado o la forma nominal. Véase `quad_qag` y sus funciones relacionadas para aproximaciones numéricas de integrales definidas.

`erfflag` [Variable optativa]

Valor por defecto: `true`

Si `erfflag` vale `false`, la función `risch` no introduce la función `erf` en el resultado si no había ninguna en el integrando.

`ilt` (*expr*, *s*, *t*) [Función]

Calcula la transformada inversa de Laplace de *expr* con respecto de *s* y parámetro *t*. El argumento *expr* debe ser una fracción de polinomios cuyo denominador tenga sólo factores lineales y cuadráticos. Utilizando las funciones `laplace` y `ilt`, junto con las funciones `solve` o `linsolve`, el usuario podrá resolver ciertas ecuaciones integrales.

```
(%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
```

`intanalysis` [Variable opcional]

Valor por defecto: `true`

Cuando vale `true`, la integración definida trata de encontrar polos en el integrando dentro del intervalo de integración. Si encuentra alguno, entonces la integral se calcula como valor principal. Si `intanalysis` vale `false`, entonces no se realiza esta comprobaciÃ³n y la integración se realiza sin tener en cuenta los polos.

Véase también `ldefint`.

Ejemplos:

Maxima puede calcular las siguientes integrales cuando a `intanalysis` se le asigna el valor `false`:

```
(%i1) integrate(1/(sqrt(x)+1),x,0,1);
                                 1
                                /
                                [       1
(%o1)                           I  ----------- dx
                                ]  sqrt(x) + 1
                                /
                                 0

(%i2) integrate(1/(sqrt(x)+1),x,0,1),intanalysis:false;
(%o2)                             2 - 2 log(2)

(%i3) integrate(cos(a)/sqrt((tan(a))^2 +1),a,-%pi/2,%pi/2);
The number 1 isn't in the domain of atanh
 -- an error. To debug this try: debugmode(true);

(%i4) intanalysis:false$
(%i5) integrate(cos(a)/sqrt((tan(a))^2+1),a,-%pi/2,%pi/2);
                                        %pi
(%o5)                                   ---
                                         2
```

`integrate` (*expr*, *x*) [Función]

`integrate` (*expr*, *x*, *a*, *b*) [Función]

Calcula simbólicamente la integral de *expr* respecto de *x*. La llamada `integrate (expr, x)` resuelve una integral indefinida, mientras que `integrate (expr, x, a, b)` resuelve una integral definida con límites de integración *a* y *b*. Los límites no

pueden contener a *x*. El argumento *a* no necesita ser menor que *b*. Si *b* es igual a *a*, `integrate` devuelve cero.

Véase `quad_qag` y funciones relacionadas para la aproximación numérica de integrales definidas. Véase `residue` para el cálculo de residuos (integración compleja). Véase `antid` para un método alternativo de resolución de integrales indefinidas.

Se obtendrá una integral (es decir, una expresión sin `integrate`) si `integrate` tiene éxito en el cálculo. En otro caso, la respuesta es la forma nominal de la integral (esto es, el operador `'integrate` precedido de apóstrofo) o una expresión que contiene una o más formas nominales. La forma nominal de `integrate` se muestra con un símbolo integral.

En ciertos casos es útil proporcionar una forma nominal 'a mano', haciendo preceder `integrate` con una comilla simple o apóstrofo, como en `'integrate (`*`expr, x`*`)`. Por ejemplo, la integral puede depender de algunos parámetros que todavía no han sido calculados. La forma nominal puede aplicarse después a sus argumentos haciendo `ev (`*`i`*`, nouns)` donde *i* es la forma nominal de interés.

La función `integrate` trata de manera diferente las integrales definidas de las indefinidas, empleando una batería de heurísticas especial para cada caso. Casos especiales de integrales definidas incluyen las que tienen límites de integración iguales a cero o a infinito (`inf` o `minf`), funciones trigonométricas con límites de integración igual a cero y `%pi` o `2 %pi`, funciones racionales, integrales relacionadas con las funciones `beta` y `psi` y algunas integrales logarítmicas y trigonométricas. El tratamiento de funciones racionales puede incluir el cálculo de residuos. Si no se reconoce ninguno de los casos especiales, se intenta resolver la integral idefinida y evaluarla en los límites de integración. Esto incluye tomar límites cuando alguno de los extremos del intervalo de integración se acerca a más infinito o a menos infinito; véase también `ldefint`.

Casos especiales de integrales indefinidas incluyen a las funciones trigonométricas, exponenciales, logarítmicas y racionales. La función `integrate` también hace uso de una pequeña tabla de integrales elementales.

La función `integrate` puede llevar a cabo cambios de variable si el integrando es de la forma `f(g(x)) * diff(g(x), x)`, entonces `integrate` trata de encontrar una subexpresión de `g(x)` tal que la derivada de `g(x)` divida el integrando. Esta búsqueda puede hacer uso de las derivadas establecidas con la función `gradef`. Véanse también `changevar` y `antid`.

Si ninguna de las heurísticas descritas resuelve la integral indefinida, se ejecuta el algoritmo de Risch. La variable `risch` puede utilizarse como una `evflag`, en una llamada a `ev` o en la línea de comandos por ejemplo, `ev (integrate (`*`expr, x`*`), risch)` o `integrate (`*`expr, x`*`), risch`. Si `risch` está presenta, `integrate` llama a la función `risch` sin intentar primero las heurísticas. Véase también `risch`.

La función `integrate` opera únicamente con relaciones funcionales que se representen explícitamente con la notación `f(x)`, sin considerar las dependencias implícitas establecidas mediante la función `depends`.

Es posible que `integrate` necesite conocer alguna propiedad de alguno de los parámetros presentes en el integrando, en cuyo caso `integrate` consultará en primer lugar la base de datos creada con `assume`, y si la variable de interés no se encuentra

ahí, `integrate` le preguntará al usuario. Dependiendo de la pregunta, posibles respuestas son: `yes;`, `no;`, `pos;`, `zero;` o `neg;`.

Por defecto, `integrate` no se considera lineal. Véanse `declare` y `linear`.

La función `integrate` intentará la integración por partes sólo en casos especiales.

Ejemplos:

- Integrales elementales indefinidas y definidas.

```
(%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
```

- Utilización de `assume` e interacción.

```
(%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
```

- Cambio de variable. En este ejemplo hay dos cambios de variable: uno utilizando una derivada establecida con `gradef` y otra utilizando la derivada `diff(r(x))` de una función no 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)))
```

- El valor devuelto contiene la forma nominal `'integrate`. En este ejemplo, Maxima puede extraer un factor del denominador de una función racional, pero no puede factorizar el resto. La función `grind` muestra la forma nominal `'integrate` del resultado. Véase también `integrate_use_rootsof` para más información sobre integrales de funciones racionales.

```
(%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$
```

- Definición de una función mediante una integral. El cuerpo de una función no se evalúa cuando ésta se define, de manera que el cuerpo de `f_1` en este ejemplo contiene la forma nominal de `integrate`. El operador comilla-comilla `''` hace que se evalúe la integral y su resultado será el que defina a la función `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` [Variable del sistema]

Valor por defecto: `%c`

Cuando una constante de integración se crea durante la integración definida de una ecuación, el nombre de la constante se construye concatenando `integration_constant` y `integration_constant_counter`.

A `integration_constant` se le puede asignar un símbolo cualquiera.

Ejemplos:

```
(%i1) integrate (x^2 = 1, x);
                              3
                             x
(%o1)                        -- = x + %c1
                             3
(%i2) integration_constant : 'k;
(%o2)                             k
(%i3) integrate (x^2 = 1, x);
                              3
                             x
(%o3)                        -- = x + k2
                             3
```

`integration_constant_counter` [Variable del sistema]

Valor por defecto: 0

Cuando una constante de integración se crea durante la integración definida de una ecuación, el nombre de la constante se construye concatenando `integration_constant` y `integration_constant_counter`.

La variable `integration_constant_counter` se incrementa antes de construir la constante de integración siguiente.

Ejemplos:

```
(%i1) integrate (x^2 = 1, x);
                              3
                             x
(%o1)                        -- = x + %c1
                             3
(%i2) integrate (x^2 = 1, x);
                              3
                             x
(%o2)                        -- = x + %c2
                             3
(%i3) integrate (x^2 = 1, x);
                              3
                             x
(%o3)                        -- = x + %c3
                             3
(%i4) reset (integration_constant_counter);
(%o4)             [integration_constant_counter]
(%i5) integrate (x^2 = 1, x);
                              3
                             x
(%o5)                        -- = x + %c1
                             3
```

`integrate_use_rootsof` [Variable optativa]

Valor por defecto: `false`

Si `integrate_use_rootsof` vale `true` y el denominador de una función racional no se puede factorizar, `integrate` devuelve la integral como una suma respecto de las raíces desconocidas del denominador.

Por ejemplo, dándole a `integrate_use_rootsof` el valor `false`, `integrate` devuelve la integral no resuelta de la función racional en forma nominal:

```
(%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)
```

Si ahora se le da a la variable el valor `true`, la parte no resuelta de la integral se expresa como una suma cuyos sumandos dependen de las raíces del denominador de la función racional:

```
(%i3) integrate_use_rootsof: true$
(%i4) integrate (1/(1+x+x^5), x);
      ====        2
      \       (%r4  - 4 %r4 + 5) log(x - %r4)
       >      ------------------------------
      /                    2
      ====            3 %r4  - 2 %r4
                        3      2
      %r4 in rootsof(%r4  - %r4  + 1, %r4)
(%o4) ------------------------------------------------------------
               7

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

Alternativamente, el usuario puede calcular las raíces del denominador separadamente y luego representar el integrando en función de dichas raíces, como por ejemplo `1/((x - a)*(x - b)*(x - c))` o `1/((x^2 - (a+b)*x + a*b)*(x - c))` si el denominador es un polinomio de tercer grado. En algunos casos, esto ayudará a Maxima mejorar sus resultados.

`ldefint` (*expr*, *x*, *a*, *b*) [Función]

Calcula la integral definida de *expr* utilizando `limit` tras el cálculo de la integral indefinida de *expr* respecto a *x* en los extremos de integración *b* y *a*. Si no consigue

calcular la integral definida, `ldefint` devuelve una expresión con los límites en forma nominal.

La función `integrate` no llama a `ldefint`, de modo que la ejecución de `ldefint (expr, x, a, b)` puede dar un resultado diferente que `integrate (expr, x, a, b)`. La función `ldefint` siempre utiliza el mismo método para calcular la integral definida, mientras que `integrate` puede hacer uso de varias heurísticas y reconocer así casos especiales.

`residue (expr, z, z_0)` [Función]

Calcula el residuo en el plano complejo de la expresión *expr* cuando la variable *z* toma el valor *z_0*. El residuo es el coeficiente de `(z - z_0)^(-1)` en el desarrollo de Laurent de *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)` [Función]

Integra *expr* respecto de *x* utilizando el caso trascendental del algoritmo de Risch. El caso algebraico del algoritmo de Risch no se ha implementado. Este método trata los casos de exponenciales y logaritmos anidados que no resuelve el procedimiento principal de `integrate`. La función `integrate` llamará automáticamente a `risch` si se presentan estos casos.

Si la variable `erfflag` vale `false`, evita que `risch` introduzca la función `erf` en la respuesta si ésta no estaba presente previamente en el integrando.

```
(%i1) risch (x^2*erf(x), x);
                                                          2
             3                        2                - x
        %pi x  erf(x) + (sqrt(%pi) x  + sqrt(%pi)) %e
(%o1)   ---------------------------------------------------
                              3 %pi
(%i2) diff(%, x), ratsimp;
                             2
(%o2)                       x  erf(x)
```

`tldefint (expr, x, a, b)` [Función]

Equivale a `ldefint` cuando `tlimswitch` vale `true`.

19.3 Introducción a QUADPACK

QUADPACK es un conjunto de funciones para el cálculo numérico de integrales definidas de una variable. Se creó a partir de un trabajo conjunto de R. Piessens[1], E. de Doncker[2], C. Ueberhuber[3], and D. Kahaner[4].

La librería QUADPACK incluida en Maxima es una traducción automática (mediante el programa `f2cl`) del código fuente Fortran de QUADPACK tal como se encuentra en la SLATEC Common Mathematical Library,Versión 4.1[5]. La librería SLATEC está fechada en julio de 1993, pero las funciones QUADPACK fueron escritas algunos años antes. Hay otra versión de QUADPACK en Netlib[6], pero no está claro hasta qué punto difiere de la que forma parte de la librería SLATEC.

Las funciones QUADPACK incluidas en Maxima son todas automáticas, en el sentido de que estas funciones intentan calcular sus resultados con una exactitud especificada, requiriendo un número indeterminado de evaluaciones de funciones. La traducción a Lisp que Maxima hace de QUADPACK incluye también algunas funciones que no son automáticas, pero que no son accesibles desde el nivel de Maxima.

Se puede encontrar más información sobre QUADPACK en el libro[7].

19.3.1 Perspectiva general

`quad_qag` Integración de una función general en un intervalo finito. La función `quad_qag` implementa un integrador global adaptativo simple utilizando una estrategia de Aind (Piessens, 1973). Se puede escoger entre seis pares de fórmulas de cuadratura de Gauss-Kronrod para la regla de evaluación. Las reglas de rango superior son útiles en los casos en los que los integrandos tienen un alto grado de oscilación.

`quad_qags`
Integración de una función general en un intervalo finito. La función `quad_qags` implementa la subdivisión de intervalos global adaptativa con extrapolación (de Doncker, 1978) mediante el algoritmo Epsilon (Wynn, 1956).

`quad_qagi`
Integración de una función general en un intervalo infinito o semi-infinito. El intervalo se proyecta sobre un intervalo finito y luego se aplica la misma estrategia que en `quad_qags`.

`quad_qawo`
Integración de $cos(omegax)f(x)$ o $sin(omegax)f(x)$ en un intervalo finito, siendo *omega* una constante. La regla de evaluación se basa en la técnica modificada

[1] Applied Mathematics and Programming Division, K.U. Leuven
[2] Applied Mathematics and Programming Division, K.U. Leuven
[3] Institut für Mathematik, T.U. Wien
[4] National Bureau of Standards, Washington, D.C., U.S.A
[5] http://www.netlib.org/slatec
[6] http://www.netlib.org/quadpack
[7] R. Piessens, E. de Doncker-Kapenga, C.W. Uberhuber, and D.K. Kahaner. *QUADPACK: A Subroutine Package for Automatic Integration.* Berlin: Springer-Verlag, 1983, ISBN 0387125531.

de Clenshaw-Curtis. La función `quad_qawo` aplica la subdivisión adaptativa con extrapolación, de forma similar a `quad_qags`.

`quad_qawf`

Calcula la transformada seno o coseno de Fourier en un intervalo semi-infinito. Se aplica el mismo método que en `quad_qawo` a sucesivos intervalos finitos, acelerando la convergencia mediante el algoritmo Epsilon (Wynn, 1956).

`quad_qaws`

Integración de $w(x)f(x)$ en un intervalo finito $[a, b]$, siendo w una función de la forma $(x-a)^{a}lpha(b-x)^{b}etav(x)$, con $v(x)$ igual a 1, a $log(x-a)$, a $log(b-x)$ o a $log(x-a)log(b-x)$ y con $alpha > -1$, y $beta > -1$. Se aplica una estrategia de subdivisión adaptativa global, con integración de Clenshaw-Curtis modificada en los subintervalos que contienen a a y a b.

`quad_qawc`

Calcula el valor principal de Cauchy de $f(x)/(x-c)$ en un intervalo finito (a, b) para una c dada. La estrategia es global adaptativa, utilizando la integración de Clenshaw-Curtis modificada en los subintervalos que contienen a $x = c$.

`quad_qagp`

Básicamente hace lo mismo que `quad_qags`, pero los puntos de singularidad o discontinuidad deben ser aportados por el usuario. Esto hace que el cálculo de una buena solución sea más fácil para el integrador.

19.4 Funciones y variables para QUADPACK

`quad_qag` (*f*(*x*), *x*, *a*, *b*, *key*, [*epsrel*, *epsabs*, *limit*]) [Función]
`quad_qag` (*f*, *x*, *a*, *b*, *key*, [*epsrel*, *epsabs*, *limit*]) [Función]

Integración de una función general en un intervalo finito. La función `quad_qag` implementa un integrador global adaptativo simple utilizando una estrategia de Aind (Piessens, 1973). Se puede escoger entre seis pares de fórmulas de cuadratura de Gauss-Kronrod para la regla de evaluación. Las reglas de rango superior son útiles en los casos en los que los integrandos tienen un alto grado de oscilación.

La función `quad_qag` calcula numéricamente la integral

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

utilizando un integrador adaptativo simple.

La función a integrar es *f(x)*, con variable independiente *x*, siendo el intervalo de integración el comprendido entre *a* y *b*. El argumento *key* indica el integrador a utilizar y debe ser un número entero entre 1 y 6, ambos inclusive. El valor de *key* selecciona el orden de la regla de integración de Gauss-Kronrod. Las reglas de rango superior son útiles en los casos en los que los integrandos tienen un alto grado de oscilación.

El integrando se puede especificar con el nombre de una función u operador de Maxima o de Lisp, como una expresión lambda o como una expresión general de Maxima.

La integración numérica se hace de forma adaptativa particionando la región de integración en subintervalos hasta conseguir la precisión requerida.

Los argumentos opcionales pueden especificarse en cualquier orden. Todos ellos toman la forma `key=val`. Tales argumentos son:

`epsrel` Error relativo deseado de la aproximación. El valor por defecto es 1d-8.

`epsabs` Error absoluto deseado de la aproximación. El valor por defecto es 0.

`limit` Tamaño del array interno utilizado para realizar la cuadratura. *limit* es el número máximo de subintervalos a utilizar. El valor por defecto es 200.

La función `quad_qag` devuelve una lista de cuatro elementos:

- la aproximación a la integral,
- el error absoluto estimado de la aproximación,
- el número de evaluaciones del integrando,
- un código de error.

El código de error (el cuarto elemento del resultado) puede tener los siguientes valores:

`0` si no ha habido problemas;

`1` si se utilizaron demasiados intervalos;

`2` si se encontró un número excesivo de errores de redondeo;

`3` si el integrando ha tenido un comportamiento extraño frente a la integración;

`6` si los argumentos de entrada no son válidos.

Ejemplos:

```
(%i1) quad_qag (x^(1/2)*log(1/x), x, 0, 1, 3, 'epsrel=5d-8);
(%o1)    [.4444444444492108, 3.1700968502883E-9, 961, 0]
(%i2) integrate (x^(1/2)*log(1/x), x, 0, 1);
                                4
(%o2)                           -
                                9
```

`quad_qags` (*f*(*x*), *x*, *a*, *b*, [*epsrel*, *epsabs*, *limit*]) [Función]
`quad_qags` (*f*, *x*, *a*, *b*, [*epsrel*, *epsabs*, *limit*]) [Función]

Integración de una función general en un intervalo finito. La función `quad_qags` implementa la subdivisión de intervalos global adaptativa con extrapolación (de Doncker, 1978) mediante el algoritmo Epsilon (Wynn, 1956).

La función `quad_qags` calcula la integral

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

La función a integrar es *f(x)*, de variable independiente *x*, siendo el intervalo de integración el comprendido entre *a* y *b*.

El integrando se puede especificar con el nombre de una función u operador de Maxima o de Lisp, como una expresión lambda o como una expresión general de Maxima.

Los argumentos opcionales pueden especificarse en cualquier orden. Todos ellos toman la forma `key=val`. Tales argumentos son:

`epsrel` Error relativo deseado de la aproximación. El valor por defecto es 1d-8.

`epsabs` Error absoluto deseado de la aproximación. El valor por defecto es 0.

`limit` Tamaño del array interno utilizado para realizar la cuadratura. *limit* es el número máximo de subintervalos a utilizar. El valor por defecto es 200.

La función `quad_qags` devuelve una lista de cuatro elementos:

- la aproximación a la integral,
- el error absoluto estimado de la aproximación,
- el número de evaluaciones del integrando,
- un código de error.

El código de error (el cuarto elemento del resultado) puede tener los siguientes valores:

`0` si no ha habido problemas;

`1` si se utilizaron demasiados intervalos;

`2` si se encontró un número excesivo de errores de redondeo;

`3` si el integrando ha tenido un comportamiento extraño frente a la integración;

`4` fallo de convergencia;

`5` la integral es probablemente divergente o de convergencia lenta;

`6` si los argumentos de entrada no son válidos.

Ejemplos:

```
(%i1) quad_qags (x^(1/2)*log(1/x), x, 0, 1, 'epsrel=1d-10);
(%o1)   [.4444444444444448, 1.11022302462516E-15, 315, 0]
```

Nótese que `quad_qags` es más precisa y eficiente que `quad_qag` para este integrando.

`quad_qagi` (*f*(*x*), *x*, `a`, *b*, [*epsrel*, *epsabs*, *limit*]) [Función]
`quad_qagi` (*f*, *x*, `a`, *b*, [*epsrel*, *epsabs*, *limit*]) [Función]

Integración de una función general en un intervalo infinito o semi-infinito. El intervalo se proyecta sobre un intervalo finito y luego se aplica la misma estrategia que en `quad_qags`.

La función `quad_qagi` calcula cualquiera las siguientes integrales:

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

$$\int_\infty^a f(x)dx$$

$$\int_{-\infty}^{\infty} f(x)dx$$

utilizando la rutina QAGI de Quadpack QAGI. La función a integrar es *f(x)*, con variable independiente *x*, siendo el intervalo de integración de rango infinito.

El integrando se puede especificar con el nombre de una función u operador de Maxima o de Lisp, como una expresión lambda o como una expresión general de Maxima.

Uno de los límites de integración debe ser infinito. De no ser así, `quad_qagi` devolverá una forma nominal.

Los argumentos opcionales pueden especificarse en cualquier orden. Todos ellos toman la forma `key=val`. Tales argumentos son:

`epsrel` Error relativo deseado de la aproximación. El valor por defecto es 1d-8.

`epsabs` Error absoluto deseado de la aproximación. El valor por defecto es 0.

`limit` Tamaño del array interno utilizado para realizar la cuadratura. *limit* es el número máximo de subintervalos a utilizar. El valor por defecto es 200.

La función `quad_qagi` devuelve una lista de cuatro elementos:

- la aproximación a la integral,
- el error absoluto estimado de la aproximación,
- el número de evaluaciones del integrando,
- un código de error.

El código de error (el cuarto elemento del resultado) puede tener los siguientes valores:

`0` si no ha habido problemas;

`1` si se utilizaron demasiados intervalos;

`2` si se encontró un número excesivo de errores de redondeo;

`3` si el integrando ha tenido un comportamiento extraño frente a la integración;

`4` fallo de convergencia;

`5` la integral es probablemente divergente o de convergencia lenta;

`6` si los argumentos de entrada no son válidos.

Ejemplos:

```
(%i1) quad_qagi (x^2*exp(-4*x), x, 0, inf, 'epsrel=1d-8);
(%o1)        [0.03125, 2.95916102995002E-11, 105, 0]
(%i2) integrate (x^2*exp(-4*x), x, 0, inf);
                                1
(%o2)                           --
                                32
```

quad_qawc (*f*(*x*), *x*, *c*, *a*, *b*, [*epsrel*, *epsabs*, *limit*]) [Función]
quad_qawc (*f*, *x*, *c*, *a*, *b*, [*epsrel*, *epsabs*, *limit*]) [Función]

Calcula el valor principal de Cauchy de $f(x)/(x-c)$ en un intervalo finito (a,b) para una c dada. La estrategia es global adaptativa, utilizando la integración de Clenshaw-Curtis modificada en los subintervalos que contienen a $x=c$.

La función `quad_qawc` calcula el valor principal de Cauchy de

$$\int_a^b \frac{f(x)}{x-c}\,dx$$

utilizando la rutina QAWC de Quadpack. La función a integrar es `f(x)/(x - c)`, con variable independiente x, siendo el intervalo de integración el comprendido entre a y b.

El integrando se puede especificar con el nombre de una función u operador de Maxima o de Lisp, como una expresión lambda o como una expresión general de Maxima.

Los argumentos opcionales pueden especificarse en cualquier orden. Todos ellos toman la forma `key=val`. Tales argumentos son:

`epsrel` Error relativo deseado de la aproximación. El valor por defecto es 1d-8.

`epsabs` Error absoluto deseado de la aproximación. El valor por defecto es 0.

`limit` Tamaño del array interno utilizado para realizar la cuadratura. *limit* es el número máximo de subintervalos a utilizar. El valor por defecto es 200.

`quad_qawc` returns a list of four elements:

- la aproximación a la integral,
- el error absoluto estimado de la aproximación,
- el número de evaluaciones del integrando,
- un código de error.

El código de error (el cuarto elemento del resultado) puede tener los siguientes valores:

`0` si no ha habido problemas;

`1` si se utilizaron demasiados intervalos;

`2` si se encontró un número excesivo de errores de redondeo;

`3` si el integrando ha tenido un comportamiento extraño frente a la integración;

`6` si los argumentos de entrada no son válidos.

Ejemplos:

```
(%i1) quad_qawc (2^(-5)*((x-1)^2+4^(-5))^(-1), x, 2, 0, 5,
                 'epsrel=1d-7);
(%o1)    [- 3.130120337415925, 1.306830140249558E-8, 495, 0]
(%i2) integrate (2^(-alpha)*(((x-1)^2 + 4^(-alpha))*(x-2))^(-1),
                 x, 0, 5);
Principal Value
        alpha        9 4                 9
```

```
                                                  4                  log(-------------- + --------------)
                                                                              alpha + 3                  alpha + 3
                                                                             4          + 4             4          + 4
(%o2)                                  (-----------------------------------------------------------------------
                                                                                    alpha
                                                                               2 4      + 2
                                                  3 alpha                                   3 alpha
                                                  -------                                   -------
                                                     2                    alpha/2             2                alpha/2
                                                4                  atan(4       )          4                atan(- 4 4       )
                                             - ---------------------- + ---------------------------------------------)
                                                       alpha                                       alpha
                                                      4      + 1                                  4      + 1
                                               alpha
                                              /2
(%i3) ev (%, alpha=5, numer);
(%o3)                                                  - 3.130120337415917
```

quad_qawf (*f*(*x*), *x*, *a*, *omega*, *trig*, [*epsabs*, *limit*, *maxp1*, *limlst*]) [Función]
quad_qawf (*f*, *x*, *a*, *omega*, *trig*, [*epsabs*, *limit*, *maxp1*, *limlst*]) [Función]

Calcula la transformada seno o coseno de Fourier en un intervalo semi-infinito. Se aplica el mismo método que en `quad_qawo` a sucesivos intervalos finitos, acelerando la convergencia mediante el algoritmo Epsilon (Wynn, 1956).

La función `quad_qawf` calcula la integral

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

La función peso w se selecciona mediante *trig*:

`cos` $w(x) = cos(omegax)$

`sin` $w(x) = sin(omegax)$

El integrando se puede especificar con el nombre de una función u operador de Maxima o de Lisp, como una expresión lambda o como una expresión general de Maxima

Los argumentos opcionales pueden especificarse en cualquier orden. Todos ellos toman la forma `key=val`. Tales argumentos son:

`epsabs` El error absoluto deseado para la aproximación. El valor por defecto es 1d-10.

`limit` Tamaño del arreglo interno de trabajo. (*limit* - *limlst*)/2 es el número máximo de subintervalos para la partición. El valor por defecto es 200.

`maxp1` Número máximo de momentos de Chebyshev. Debe ser mayor que 0. El valor por defecto es 100.

`limlst` Cota superior del número de ciclos. Debe ser mayor o igual que 3. El valor por defecto es 10.

`quad_qawf` returns a list of four elements:

- la aproximación a la integral,
- el error absoluto estimado de la aproximación,
- el número de evaluaciones del integrando,
- un código de error.

El código de error (el cuarto elemento del resultado) puede tener los siguientes valores:

`0` si no ha habido problemas;

`1` si se utilizaron demasiados intervalos;

`2` si se encontró un número excesivo de errores de redondeo;

`3` si el integrando ha tenido un comportamiento extraño frente a la integración;

`6` si los argumentos de entrada no son válidos.

Ejemplos:

```
(%i1) quad_qawf (exp(-x^2), x, 0, 1, 'cos, 'epsabs=1d-9);
(%o1)   [.6901942235215714, 2.84846300257552E-11, 215, 0]
(%i2) integrate (exp(-x^2)*cos(x), x, 0, inf);
                           - 1/4
                         %e      sqrt(%pi)
(%o2)                    -----------------
                                 2
(%i3) ev (%, numer);
(%o3)                    .6901942235215714
```

`quad_qawo` (*f*(*x*), *x*, *a*, *b*, *omega*, *trig*, [*epsrel*, *epsabs*, *limit*, *maxp1*, *limlst*]) [Función]

`quad_qawo` (*f*, *x*, *a*, *b*, *omega*, *trig*, [*epsrel*, *epsabs*, *limit*, *maxp1*, *limlst*]) [Función]

Integración de $cos(omegax)f(x)$ o $sin(omegax)f(x)$ en un intervalo finito, siendo *omega* una constante. La regla de evaluación se basa en la técnica modificada de Clenshaw-Curtis. La función `quad_qawo` aplica la subdivisión adaptativa con extrapolación, de forma similar a `quad_qags`.

La función `quad_qawo` realiza la integración utilizando la rutina QAWO de Quadpack:

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

La función peso w se selecciona mediante *trig*:

`cos` $w(x) = cos(omegax)$

`sin` $w(x) = sin(omegax)$

El integrando se puede especificar con el nombre de una función u operador de Maxima o de Lisp, como una expresión lambda o como una expresión general de Maxima

Los argumentos opcionales pueden especificarse en cualquier orden. Todos ellos toman la forma `key=val`. Tales argumentos son:

`epsrel` El error absoluto deseado para la aproximación. El valor por defecto es 1d-8.

`epsabs` Error absoluto deseado de la aproximación. El valor por defecto es 0.

`limit` Tamaño del arreglo interno de trabajo. *limit*/2 es el número máximo de subintervalos para la partición. El valor por defecto es 200.

`maxp1` Número máximo de momentos de Chebyshev. Debe ser mayor que 0. El valor por defecto es 100.

`limlst` Cota superior del número de ciclos. Debe ser mayor o igual que 3. El valor por defecto es 10.

`quad_qawo` returns a list of four elements:

- la aproximación a la integral,
- el error absoluto estimado de la aproximación,
- el número de evaluaciones del integrando,
- un código de error.

El código de error (el cuarto elemento del resultado) puede tener los siguientes valores:

`0` si no ha habido problemas;

`1` si se utilizaron demasiados intervalos;

`2` si se encontró un número excesivo de errores de redondeo;

`3` si el integrando ha tenido un comportamiento extraño frente a la integración;

`6` si los argumentos de entrada no son válidos.

Ejemplos:

```
(%i1) quad_qawo (x^(-1/2)*exp(-2^(-2)*x), x, 1d-8, 20*2^2, 1, cos);
(%o1)     [1.376043389877692, 4.72710759424899E-11, 765, 0]
(%i2) rectform (integrate (x^(-1/2)*exp(-2^(-alpha)*x) * cos(x),
                           x, 0, inf));
                      alpha/2 - 1/2            2 alpha
        sqrt(%pi) 2              sqrt(sqrt(2        + 1) + 1)
(%o2)   ----------------------------------------------------
                                2 alpha
                           sqrt(2        + 1)
(%i3) ev (%, alpha=2, numer);
(%o3)                     1.376043390090716
```

`quad_qaws` (*f*(*x*), *x*, *a*, *b*, *alpha*, *beta*, *wfun*, [*epsrel*, *epsabs*, *limit*]) [Función]
`quad_qaws` (*f*, *x*, *a*, *b*, *alpha*, *beta*, *wfun*, [*epsrel*, *epsabs*, *limit*]) [Función]

Integración de $w(x)f(x)$ en un intervalo finito $[a, b]$, siendo w una función de la forma $(x-a)^a lpha(b-x)^b etav(x)$, con $v(x)$ igual a 1, a $log(x-a)$, a $log(b-x)$ o a $log(x-$

$a)log(b-x)$ y con $alpha > -1$, y $beta > -1$. Se aplica una estrategia de subdivisión adaptativa global, con integración de Clenshaw-Curtis modificada en los subintervalos que contienen a a y a b.

La función `quad_qaws` realiza la integración utizando la rutina QAWS de Quadpack:

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

La función peso w se selecciona mediante *wfun*:

`1` $w(x) = (x-a)^a lfa(b-x)^b eta$

`2` $w(x) = (x-a)^a lfa(b-x)^b etalog(x-a)$

`3` $w(x) = (x-a)^a lfa(b-x)^b etalog(b-x)$

`4` $w(x) = (x-a)^a lfa(b-x)^b etalog(x-a)log(b-x)$

El integrando se puede especificar con el nombre de una función u operador de Maxima o de Lisp, como una expresión lambda o como una expresión general de Maxima

Los argumentos opcionales pueden especificarse en cualquier orden. Todos ellos toman la forma `key=val`. Tales argumentos son:

`epsrel` El error absoluto deseado para la aproximación. El valor por defecto es 1d-8.

`epsabs` Error absoluto deseado de la aproximación. El valor por defecto es 0.

`limit` Tamaño del array interno utilizado para realizar la cuadratura. (*limit* - *limlst*)/2 es el número máximo de subintervalos a utilizar. El valor por defecto es 200.

`quad_qaws` returns a list of four elements:

- la aproximación a la integral,
- el error absoluto estimado de la aproximación,
- el número de evaluaciones del integrando,
- un código de error.

El código de error (el cuarto elemento del resultado) puede tener los siguientes valores:

`0` si no ha habido problemas;

`1` si se utilizaron demasiados intervalos;

`2` si se encontró un número excesivo de errores de redondeo;

`3` si el integrando ha tenido un comportamiento extraño frente a la integración;

`6` si los argumentos de entrada no son válidos.

Ejemplos:

```
(%i1) quad_qaws (1/(x+1+2^(-4)), x, -1, 1, -0.5, -0.5, 1,
                 'epsabs=1d-9);
(%o1)     [8.750097361672832, 1.24321522715422E-10, 170, 0]
```

```
(%i2) integrate ((1-x*x)^(-1/2)/(x+1+2^(-alpha)), x, -1, 1);
       alpha
Is  4 2      - 1  positive, negative, or zero?

pos;
                            alpha          alpha
                     2 %pi 2      sqrt(2 2      + 1)
(%o2)                ------------------------------
                                  alpha
                               4 2      + 2
(%i3) ev (%, alpha=4, numer);
(%o3)                      8.750097361672829
```

quad_qagp (*f*(*x*), *x*, a, *b*, *points*, [*epsrel*, *epsabs*, *limit*]) [Fución]
quad_qagp (*f*, *x*, a, *b*, *points*, [*epsrel*, *epsabs*, *limit*]) [Fución]

Integra una función general sobre un intervalo acotado. La función `quad_qagp` implementa un método adaptativo global de subdivisión del intervalo con extrapolación (de Doncker, 1978) basado en el algoritmo Epsilon (Wynn, 1956).

`quad_qagp` calcula la integral

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

La función a integrar es *f(x)*, con variable independiente *x*, en el intervalo limitado por *a* y *b*.

El integrando puede especificarse mediante el nombre de una función de Maxima o de Lisp o un operador, como una expresión lambda de Maxima, o como una expresión general de Maxima.

Para ayudar al integrador, el usuario debe aportar una lista de puntos donde el integrando es singular o discontinuo.

Las opciones se suministran como argumentos y se pueden escribir en cualquier orden. Deben tomar la forma `opción=valor`. Las opciones son:

`epsrel` Error relativo de aproximación deseado. Valor por defecto es 1d-8.

`epsabs` Error absoluto de aproximación deseado. Valor por defecto es 0.

`limit` Tamaño del array interno de trabajo. *limit* es el máximo número de subintervalos a utilizar. Valor por defecto es 200.

`quad_qagp` devuelve una lista con cuatro elementos:

- una aproximación a la integral,
- el error absoluto estimado de la aproximación,
- el número de evaluaciones del integrando,
- un código de error.

El código de error (cuarto elemento de la lista devuelta) puede tener los siguientes valores:

`0` no se encontraron errores;

`1` se han hecho demasiados subintervalos;

`2` se detectó un error de redondeo muy grande;

`3` se ha observado un comportamiento del integrando extremadamente malo;

`4` fallo de convergencia;

`5` la integral es probablemente divergengente o converge muy lentamente;

`6` entrada inválida.

Ejemplos:

```
(%i1) quad_qagp(x^3*log(abs((x^2-1)*(x^2-2))),x,0,3,[1,sqrt(2)]);
(%o1)   [52.74074838347143, 2.6247632689546663e-7, 1029, 0]
(%i2) quad_qags(x^3*log(abs((x^2-1)*(x^2-2))), x, 0, 3);
(%o2)   [52.74074847951494, 4.088443219529836e-7, 1869, 0]
```

El integrando tiene singularidades en `1` y `sqrt(2)`, de manera que suministramos estos puntos a `quad_qagp`. También se observa que `quad_qagp` es más exacto y eficiente que `quad_qags`.

`quad_control` (*parameter*, [*value*]) [Fución]

Controla la gestión de los errores de `QUADPACK`. El parámetro debe ser uno de los siguientes símbolos:

`current_error`
El número de error actual.

`control` Controla si los mensajes se imprimen o no. Si el valor es cero o menor, los mensajes se suprimen.

`max_message`
El máximo número de veces que se imprime cualquier mensaje.

Si no se da *value*, entonces se devuelve el valor actual asociado a *parameter*. En cambio, si se da *value*, se hace la asignación correspondiente a *parameter*, adquiriendo este nuevo valor.

20 Ecuaciones

20.1 Funciones y variable para las ecuaciones

`%rnum_list` [Variable del sistema]

Valor por defecto: `[]`

La variable `%rnum_list` es la lista de variables introducidas en las soluciones por la funciones `solve` y `algsys`. Las variables `%r` se añaden a `%rnum_list` en su orden de creación. Esto es útil para hacer sustituciones en la solución a posteriori.

```
(%i1) solve ([x + y = 3], [x,y]);
(%o1)                [[x = 3 - %r1, y = %r1]]
(%i2) %rnum_list;
(%o2)                          [%r1]
(%i3) sol : solve ([x + 2*y + 3*z = 4], [x,y,z]);
(%o3)   [[x = - 2 %r3 - 3 %r2 + 4, y = %r3, z = %r2]]
(%i4) %rnum_list;
(%o4)                        [%r2, %r3]
(%i5) for i : 1 thru length (%rnum_list) do
        sol : subst (t[i], %rnum_list[i], sol)$
(%i6) sol;
(%o6)     [[x = - 2 t  - 3 t  + 4, y = t , z = t ]]
                     2      1           2       1
```

`algepsilon` [Variable opcional]

Valor por defecto: 10^8

La variable `algepsilon` es utilizada por `algsys`.

`algexact` [Variable opcional]

Valor por defecto: `false`

El contenido de la variable `algexact` afecta al comportamiento de `algsys` de la siguiente forma:

Si `algexact` vale `true`, `algsys` llamará siempre a `solve` y luego utilizará `realroots`.

Si `algexact` vale `false`, `solve` será llamada sólo si la ecuación no es univariante, o si es cuadrática o bicuadrática.

Sin embargo, `algexact: true` no garantiza que únicamente se obtengan soluciones exactas, ya que aunque `algsys` intente siempre dar soluciones exactas, dará resultados aproximados si no encuentra una solución mejor.

`algsys` ([*expr_1*, ..., *expr_m*], [*x_1*, ..., *x_n*]) [Función]
`algsys` ([*eqn_1*, ..., *eqn_m*], [*x_1*, ..., *x_n*]) [Función]

Resuelve el sistema de ecuaciones polinómicas *expr_1*, ..., *expr_m* o las ecuaciones *eqn_1*, ..., *eqn_m* para las variables *x_1*, ..., *x_n*. La expresión *expr* equivale a la ecuación *expr* `= 0`. Puede haber más ecuaciones que variables o viceversa.

La función `algsys` devuelve una lista de soluciones, cada una de las cuales consistente a su vez en una lista de ecuaciones asociando valores a las variables *x_1*, ..., *x_n* que

satisfacen el sistema de ecuaciones. Si `algsys` no puede encontrar soluciones devuelve la lista vacía `[]`.

Si es necesario se introducen en la solución los símbolos `%r1`, `%r2`, ..., para representar parámetros arbitrarios; estas variables también se añaden a la lista `%rnum_list`.

El proceso que se sigue es el siguiente:

(1) Primero se factorizan las ecuaciones y se reparten en subsistemas.

(2) Para cada subsistema *S_i*, se seleccionan una ecuación *E* y una variable *x*. Se elige la variable que tenga grado menor. Entonces se calcula el resultado de *E* y *E_j* respecto de *x*, siendo las *E_j* el resto de ecuaciones del subsistema *S_i*. De aquí se obtiene otro subsistema *S_i'* con una incógnita menos, ya que *x* ha sido eliminada. El proceso ahora vuelve al paso (1).

(3) En ocasiones se obtiene un subsistema consistente en una única ecuación. Si la ecuación es multivariante y no se han introducido aproximaciones en formato decimal de coma flotante, entonces se llama a `solve` para tratar de encontrar una solución exacta.

En algunos casos, `solve` no puede encontrar la solución, o si lo consigue puede que el resultado tenga una expresión muy grande.

Si la ecuación tiene una sóla incógnita y es lineal, o cuadrática o bicuadrática, entonces se llama a la función `solve` si no se han introducido aproximaciones en formato decimal. Si se han introducido aproximaciones, o si hay más de una incógnita, o si no es lineal, ni cuadrática ni bicuadrática, y si la variables `realonly` vale `true`, entonces se llama a la función `realroots` para calcular las soluciones reales. Si `realonly` vale `false`, entonces se llama a `allroots` para obtener las soluciones reales y complejas.

Si `algsys` devuelve una solución que tiene menos dígitos significativos de los requeridos, el usuario puede cambiar a voluntad el valor de `algepsilon` para obtener mayor precisión.

Si `algexact` vale `true`, se llamará siempre a `solve`.

Cuando `algsys` encuentra una ecuación con múltiples incógnitas y que contiene aproximaciones en coma flotante (normalmente debido a la imposibilidad de encontrar soluciones exactas en pasos anteriores), entonces no intenta aplicar los métodos exactos a estas ecuaciones y presenta el mensaje: "`algsys` cannot solve - system too complicated."

Las interacciones con `radcan` pueden dar lugar a expresiones grandes o complicadas. En tal caso, puede ser posible aislar partes del resultado con `pickapart` o `reveal`.

Ocasionalmente, `radcan` puede introducir la unidad imaginaria `%i` en una solución que de hecho es real.

Ejemplos:

```
(%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*) [Función]
`allroots` (*eqn*) [Función]

Calcula aproximaciones numéricas de las raíces reales y complejas del polinomio *expr* o ecuación polinómica *eqn* de una variable.

Si la variable `polyfactor` vale `true` hace que la función `allroots` factorice el polinomio para números reales si el polinomio es real, o para números complejos si el polinomio es complejo.

La función `allroots` puede dar resultados inexactos en caso de que haya raíces múltiples. Si el polinomio es real, `allroots (%i*`*p*`)` puede alcanzar mejores aproximaciones que `allroots (`*p*`)`, ya que `allroots` ejecuta entonces un algoritmo diferente.

La función `allroots` no opera sobre expresiones no polinómicas, pues requiere que el numerador sea reducible a un polinomio y el denominador sea, como mucho, un número complejo.

Para polinomios complejos se utiliza el algoritmo de Jenkins y Traub descrito en (Algorithm 419, *Comm. ACM*, vol. 15, (1972), p. 97). Para polinomios reales se utiliza el algoritmo de Jenkins descrito en (Algorithm 493, *ACM TOMS*, vol. 1, (1975), p.178).

Ejemplos:

```
(%i1) eqn: (1 + 2*x)^3 = 13.5*(1 + x^5);
                              3          5
(%o1)                (2 x + 1)  = 13.5 (x  + 1)
(%i2) soln: allroots (eqn);
(%o2) [x = .8296749902129361, x = - 1.015755543828121,
```

```
x = .9659625152196369 %i - .4069597231924075,

x = - .9659625152196369 %i - .4069597231924075, x = 1.0]
(%i3) for e in soln
        do (e2: subst (e, eqn), disp (expand (lhs(e2) - rhs(e2))));
                      - 3.5527136788005E-15

                     - 5.32907051820075E-15

          4.44089209850063E-15 %i - 4.88498130835069E-15

         - 4.44089209850063E-15 %i - 4.88498130835069E-15

                       3.5527136788005E-15

(%o3)                          done
(%i4) polyfactor: true$
(%i5) allroots (eqn);
(%o5) - 13.5 (x - 1.0) (x - .8296749902129361)

                                2
 (x + 1.015755543828121) (x  + .8139194463848151 x

 + 1.098699797110288)
```

`bfallroots` (*expr*) [Función]
`bfallroots` (*eqn*) [Función]

Calcula aproximaciones numéricas de las raíces reales y complejas del polinomio *expr* o de la ecuación polinómica *eqn* de una variable.

En todos los aspectos, `bfallroots` es idéntica a `allroots`, excepto que `bfallroots` calcula las raíces en formato bigfloat (números decimales de precisión arbitraria).

Véase `allroots` para más información.

`backsubst` [Variable opcional]

Valor por defecto: `true`

Si `backsubst` vale `false`, evita la retrosustitución en `linsolve` tras la triangularización de las ecuaciones. Esto puede ser de utilidad en problemas muy grandes, en los que la retrosustitución puede provocar la generación de expresiones extremadamente largas.

```
(%i1) eq1 : x + y + z = 6$
(%i2) eq2 : x - y + z = 2$
(%i3) eq3 : x + y - z = 0$
(%i4) backsubst : false$
(%i5) linsolve ([eq1, eq2, eq3], [x,y,z]);
(%o5)              [x = z - y, y = 2, z = 3]
(%i6) backsubst : true$
```

```
(%i7) linsolve ([eq1, eq2, eq3], [x,y,z]);
(%o7)               [x = 1, y = 2, z = 3]
```

`breakup` [Variable opcional]

Valor por defecto: `true`

Si `breakup` vale `true`, `solve` expresa sus soluciones a las ecuaciones cúbicas y cuárticas en términos de subexpresiones comunes, las cuales son asignadas a etiquetas del tipo `%t1`, `%t2`, etc. En otro caso, no se identifican subexpresiones comunes.

La asignación `breakup: true` sólo tiene efecto cuando `programmode` vale `false`.

Ejemplos:

```
(%i1) programmode: false$
(%i2) breakup: true$
(%i3) solve (x^3 + x^2 - 1);

                         sqrt(23)    25 1/3
(%t3)                   (--------- + --)
                         6 sqrt(3)   54
Solution:

                                       sqrt(3) %i   1
                                       ---------- - -
                 sqrt(3) %i   1            2        2   1
(%t4)    x = (- ---------- - -) %t3 + -------------- - -
                    2        2            9 %t3        3

                                       sqrt(3) %i   1
                                     - ---------- - -
                sqrt(3) %i   1             2        2   1
(%t5)    x = (---------- - -) %t3 + ----------------- - -
                   2        2             9 %t3         3

                                    1     1
(%t6)                  x = %t3 + ----- - -
                                  9 %t3   3
(%o6)                    [%t4, %t5, %t6]
(%i6) breakup: false$
(%i7) solve (x^3 + x^2 - 1);
Solution:

             sqrt(3) %i   1
             ---------- - -
                 2        2         sqrt(23)    25 1/3
(%t7) x = --------------------- + (--------- + --)
             sqrt(23)    25 1/3     6 sqrt(3)   54
          9 (--------- + --)
             6 sqrt(3)   54
```

```
                                                      sqrt(3) %i   1    1
                                                    (- ---------- - -) - -
                                                           2       2    3

              sqrt(23)    25 1/3  sqrt(3) %i   1
(%t8) x = (--------- + --)    (---------- - -)
              6 sqrt(3)   54          2       2

                                                    sqrt(3) %i   1
                                                  - ---------- - -
                                                       2       2      1
                                              + ---------------------- - -
                                                    sqrt(23)    25 1/3   3
                                                9 (--------- + --)
                                                    6 sqrt(3)   54

               sqrt(23)    25 1/3             1              1
(%t9)  x = (--------- + --)    + ---------------------- - -
               6 sqrt(3)   54          sqrt(23)    25 1/3   3
                                   9 (--------- + --)
                                     6 sqrt(3)   54
(%o9)                    [%t7, %t8, %t9]
```

`dimension` (*eqn*) [Función]
`dimension` (*eqn_1*, ..., *eqn_n*) [Función]

El paquete `dimen` es para análisis dimensional. La instrucción `load ("dimen")` carga el paquete y `demo ("dimen")` presenta una pequeña demostración.

`dispflag` [Variable opcional]

Valor por defecto: `true`

Si `dispflag` vale `false`, entonces se inhibirá que Maxima muestre resultados de las funciones que resuelven ecuaciones cuando éstas son llamadas desde dentro de un bloque (`block`). Cuando un bloque termina con el signo del dólar, $, a la variable `dispflag` se le asigna `false`.

`funcsolve` (*eqn*, *g*(*t*)) [Función]

Devuelve [*g*(*t*) = ...] o [], dependiendo de que exista o no una función racional *g*(*t*) que satisfaga *eqn*, la cual debe ser un polinomio de primer orden, lineal para *g*(*t*) y *g*(*t*+1)

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

```
Dependent equations eliminated:  (4 3)
                                      n
(%o2)                 f(n) = ---------------
                                (n + 1) (n + 2)
```

Aviso: esta es una implemetación rudimentaria, por lo que debe ser utilizada con cautela.

`globalsolve` [Variable opcional]

Valor por defecto: `false`

Si `globalsolve` vale `true`, a las incógnitas de las ecuaciones se les asignan las soluciones encontradas por `linsolve` y por `solve` cuando se resuelven sistemas de dos o más ecuaciones lineales.

Si `globalsolve` vale `false`, las soluciones encontradas por `linsolve` y por `solve` cuando se resuelven sistemas de dos o más ecuaciones lineales se expresan como ecuaciones y a las incógnitas no se le asignan valores.

Cuando se resuelven ecuaciones que no son sistemas de dos o más ecuaciones lineales, `solve` ignora el valor de `globalsolve`. Otras funciones que resuelven ecuaciones (como `algsys`) ignoran siempre el valor de `globalsolve`.

Ejemplos:

```
(%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*) [Función]

El paquete `inteqn` se dedica a la resolución de ecuaciones integrales. Para hacer uso de él, ejecutar la instrucción `load ("inteqn")`.

El argumento *ie* es la ecuación integral; *unk* es la función incógnita; *tech* es el método a aplicar para efectuar la resolución del problema (*tech* = `first` significa: aplica el primer método que encuentre una solución; *tech* = `all` significa: aplica todos los métodos posibles); *n* es el número máximo de términos que debe tomar `taylor`, `neumann`, `firstkindseries` o `fredseries` (también es el máximo nivel de recursión para el método de diferenciación); *guess* es la solución candidata inicial para `neumann` o `firstkindseries`.

Valores por defecto para los argumentos segundo a quinto son:

unk: `p(x)`, donde *p* es la primera función desconocida que Maxima encuentra en el integrando y *x* es la variable que actúa como argumento en la primera aparición de *p* encontrada fuera de una integral en el caso de ecuaciones de segunda especie (`secondkind`), o es la única variable aparte de la de integración en el caso de ecuaciones de primera especie (`firstkind`). Si el intento de encontrar *x* falla, el usuario será consultado para suministrar una variable independiente.

ieqnprint [Variable opcional]

Valor por defecto: `true`

La variable `ieqnprint` controla el comportamiento del resultado retornado por la instrucción `ieqn`. Si `ieqnprint` vale `false`, la lista devuelta por la función `ieqn` tiene el formato

[*solución*, *método utilizado*, *nterms*, *variable*]

donde *variable* estará ausente si la solución es exacta; en otro caso, será la palabra `approximate` o `incomplete` según que la solución sea inexacta o que no tenga forma explícita, respectivamente. Si se ha utilizado un método basado en series, *nterms* es el número de términos utilizado, que puede ser menor que el `n` dado a `ieqn`.

lhs (*expr*) [Función]

Devuelve el miembro izquierdo (es decir, el primer argumento) de la expresión *expr*, cuando el operador de *expr* es uno de los operadores de relación `< <= = # equal notequal >= >`, o un operadores de asignación `:= ::= : ::`, o un operador infijo binario definido por el usuario mediante `infix`.

Si *expr* es un átomo o si su operador es diferente de los citados más arriba, `lhs` devuelve *expr*.

Véase también `rhs`.

Ejemplo:

```
(%i1) e: aa + bb = cc;
(%o1)                      bb + aa = cc
(%i2) lhs (e);
(%o2)                         bb + aa
(%i3) rhs (e);
(%o3)                           cc
(%i4) [lhs (aa < bb), lhs (aa <= bb),
       lhs (aa >= bb), lhs (aa > bb)];
(%o4)                     [aa, aa, aa, aa]
(%i5) [lhs (aa = bb), lhs (aa # bb), lhs (equal (aa, bb)),
       lhs (notequal (aa, bb))];
(%o5)                     [aa, aa, aa, aa]
(%i6) e1: '(foo(x) := 2*x);
(%o6)                       foo(x) := 2 x
(%i7) e2: '(bar(y) ::= 3*y);
(%o7)                      bar(y) ::= 3 y
(%i8) e3: '(x : y);
(%o8)                           x : y
(%i9) e4: '(x :: y);
(%o9)                          x :: y
(%i10) [lhs (e1), lhs (e2), lhs (e3), lhs (e4)];
(%o10)                [foo(x), bar(y), x, x]
(%i11) infix ("][");
(%o11)                          ][
(%i12) lhs (aa ][ bb);
(%o12)                          aa
```

`linsolve` ([*expr_1*, ..., *expr_m*], [*x_1*, ..., *x_n*]) [Función]

Resuelve la lista de ecuaciones lineales simultáneas para la lista de variables. Las expresiones deben ser polinomios lineales respecto de las variables o ecuaciones.

Si `globalsolve` vale `true`, a cada incógnita se le asigna el valor de la solución encontrada.

Si `backsubst` vale `false`, `linsolve` no hace la sustitución tras la triangulariación de las ecuaciones. Esto puede ser necesario en problemas muy grandes en los que la sustitución puede dar lugar a la generación de expresiones enormes.

Si `linsolve_params` vale `true`, `linsolve` también genera símbolos `%r` para representar parámetros arbitrarios como los descritos para la función `algsys`. Si vale `false`, el resultado devuelto por `linsolve` expresará, si es el sistema es indeterminado, unas variables en función de otras.

Si `programmode` vale `false`, `linsolve` muestra la solución con etiquetas de expresiones intermedias (`%t`) y devuelve las lista de etiquetas.

```
(%i1) e1: x + z = y;
(%o1)                         z + x = y
(%i2) e2: 2*a*x - y = 2*a^2;
```

```
                                          2
(%o2)                   2 a x - y = 2 a
(%i3) e3: y - 2*z = 2;
(%o3)                       y - 2 z = 2
(%i4) [globalsolve: false, programmode: true];
(%o4)                      [false, true]
(%i5) linsolve ([e1, e2, e3], [x, y, z]);
(%o5)            [x = a + 1, y = 2 a, z = a - 1]
(%i6) [globalsolve: false, programmode: false];
(%o6)                     [false, false]
(%i7) linsolve ([e1, e2, e3], [x, y, z]);
Solution

(%t7)                       z = a - 1

(%t8)                        y = 2 a

(%t9)                       x = a + 1
(%o9)                     [%t7, %t8, %t9]
(%i9) ''%;
(%o9)            [z = a - 1, y = 2 a, x = a + 1]
(%i10) [globalsolve: true, programmode: false];
(%o10)                     [true, false]
(%i11) linsolve ([e1, e2, e3], [x, y, z]);
Solution

(%t11)                      z : a - 1

(%t12)                       y : 2 a

(%t13)                      x : a + 1
(%o13)                  [%t11, %t12, %t13]
(%i13) ''%;
(%o13)           [z : a - 1, y : 2 a, x : a + 1]
(%i14) [x, y, z];
(%o14)                  [a + 1, 2 a, a - 1]
(%i15) [globalsolve: true, programmode: true];
(%o15)                     [true, true]
(%i16) linsolve ([e1, e2, e3], '[x, y, z]);
(%o16)           [x : a + 1, y : 2 a, z : a - 1]
(%i17) [x, y, z];
(%o17)                  [a + 1, 2 a, a - 1]
```

`linsolvewarn` [Variable opcional]

Valor por defecto: `true`

Si `linsolvewarn` vale `true`, `linsolve` mostrará el mensaje: "Dependent equations eliminated".

`linsolve_params` [Variable opcional]

Valor por defecto: `true`

Si `linsolve_params` vale `true`, `linsolve` también genera símbolos `%r` para representar parámetros arbitrarios como los descritos para la función `algsys`. Si vale `false`, el resultado devuelto por `linsolve` expresará, si es el sistema es indeterminado, unas variables en función de otras.

`multiplicities` [System variable]

Valor por defecto: `not_set_yet`

La variable `multiplicities` es una con las multiplicidades de las soluciones encontradas por `solve` o `realroots`.

`nroots` (*p*, *low*, *high*) [Función]

Devuelve el número de raíces reales del polinomio real univariante *p* en el intervalo semiabierto (*low*, *high*]. Los extremos del intervalo pueden ser `minf` o `inf`, menos y más infinito.

La función `nroots` utiliza el método de las secuencias de Sturm.

```
(%i1) p: x^10 - 2*x^4 + 1/2$
(%i2) nroots (p, -6, 9.1);
(%o2)                          4
```

`nthroot` (*p*, *n*) [Función]

Siendo `p` un polinomio de coeficientes enteros y `n` un entero positivo, `nthroot` devuelve un polinomio `q`, también de coeficientes enteros, tal que `q^n=p`, o un mensaje de error indicando que `p` no es una `n`-potencia exacta. Esta función es bastante más rápida que `factor` y que `sqfr`.

`polyfactor` [Variable opcional]

Valor por defecto: `false`

Cuando `polyfactor` vale `true`, las funciones `allroots` y `bfallroots` factorizan el polinomio sobre los números reales si el polinomio es real, o factoriza sobre los complejos si el polinomio es complejo.

Véase un ejemplo en `allroots`.

`programmode` [Variable opcional]

Valor por defecto: `true`

Si `programmode` vale `true`, `solve`, `realroots`, `allroots` y `linsolve` devuelve sus soluciones como elementos de una lista.

Si `programmode` vale `false`, `solve` y las demás crean expresiones intermedias etiquetadas `%t1`, `t2`, etc., y les asinan las soluciones.

```
(%i1) solve(x^2+x+1);
                     sqrt(3) %i + 1      sqrt(3) %i - 1
(%o1)        [x = - --------------, x = --------------]
                           2                   2
(%i2) programmode:false$
(%i3) solve(x^2+x+1);
```

```
Solution:

                                      sqrt(3) %i + 1
(%t3)                        x = - --------------
                                         2

                                     sqrt(3) %i - 1
(%t4)                         x = --------------
                                         2
(%o4)                            [%t4, %t5]
```

`realonly` [Variable opcional]

Valor por defecto: `false`

Si `realonly` vale `true`, `algsys` sólo devuelve aquellas soluciones exentas de la constante `%i`.

`realroots` (*expr*, *bound*) [Función]
`realroots` (*eqn*, *bound*) [Función]
`realroots` (*expr*) [Función]
`realroots` (*eqn*) [Función]

Calcula aproximaciones racionales de las raíces reales del polinomio *expr* o de la ecuación polinómica *eqn* de una variable, dentro de la tolerancia especificada por *bound*. Los coeficientes de *expr* o de *eqn* deben ser números literales, por lo que las constantes simbólicas como `%pi` no son aceptadas.

La función `realroots` guarda las multiplicidades de las raíces encontradas en la variable global `multiplicities`.

La función `realroots` genera una secuencia de Sturm para acotar cada raíz, aplicando después el método de bisección para afinar las aproximaciones. Todos los coeficientes se convierten a formas racionales equivalentes antes de comenzar la búsqueda de las raíces, de modo que los cálculos se realizan con aritmética exacta racional. Incluso en el caso de que algunos coeficientes sean números decimales en coma flotante, los resultados son racionales, a menos que se les fuerce a ser decimales con las variables `float` o `numer`.

Si *bound* es menor que la unidad, todas las raíces enteras se expresan en forma exacta. Si no se especifica *bound*, se le supone igual al valor de la variable global `rootsepsilon`.

Si la variable global `programmode` vale `true`, la función `realroots` devuelve una lista de la forma `[x = x_1, x = x_2, ...]`. Si `programmode` vale `false`, `realroots` crea etiquetas `%t1`, `%t2`, ... para las expresiones intermedias, les asigna valores y, finalmente, devuelve la lista de etiquetas.

Ejemplos:

```
(%i1) realroots (-1 - x + x^5, 5e-6);
                                  612003
(%o1)                      [x = ------]
                                  524288
(%i2) ev (%[1], float);
```

```
(%o2)                 x = 1.167303085327148
(%i3) ev (-1 - x + x^5, %);
(%o3)                - 7.396496210176905E-6
(%i1) realroots (expand ((1 - x)^5 * (2 - x)^3 * (3 - x)), 1e-20);
(%o1)                 [x = 1, x = 2, x = 3]
(%i2) multiplicities;
(%o2)                       [5, 3, 1]
```

`rhs` (*expr*) [Función]

Devuelve el miembro derecho (es decir, el segundo argumento) de la expresión *expr*, cuando el operador de *expr* es uno de los operadores de relación `< <= = # equal notequal >= >`, o un operadores de asignación `:= ::= : ::`, o un operador infijo binario definido por el usuario mediante `infix`.

Si *expr* es un átomo o si su operador es diferente de los citados más arriba, `rhs` devuelve *expr*.

Véase también `lhs`.

Ejemplo:

```
(%i1) e: aa + bb = cc;
(%o1)                     bb + aa = cc
(%i2) lhs (e);
(%o2)                        bb + aa
(%i3) rhs (e);
(%o3)                          cc
(%i4) [rhs (aa < bb), rhs (aa <= bb),
       rhs (aa >= bb), rhs (aa > bb)];
(%o4)                   [bb, bb, bb, bb]
(%i5) [rhs (aa = bb), rhs (aa # bb), rhs (equal (aa, bb)),
       rhs (notequal (aa, bb))];
(%o5)                   [bb, bb, bb, bb]
(%i6) e1: '(foo(x) := 2*x);
(%o6)                    foo(x) := 2 x
(%i7) e2: '(bar(y) ::= 3*y);
(%o7)                   bar(y) ::= 3 y
(%i8) e3: '(x : y);
(%o8)                         x : y
(%i9) e4: '(x :: y);
(%o9)                        x :: y
(%i10) [rhs (e1), rhs (e2), rhs (e3), rhs (e4)];
(%o10)                  [2 x, 3 y, y, y]
(%i11) infix ("][");
(%o11)                         ][
(%i12) rhs (aa ][ bb);
(%o12)                         bb
```

`rootsconmode` [Variable opcional]

Valor por defecto: `true`

La variable `rootsconmode` controla el comportamiento de la instrucción `rootscontract`. Véase `rootscontract` para más detalles.

`rootscontract` (*expr*) [Función]

Convierte productos de raíces en raíces de productos. Por ejemplo, `rootscontract (sqrt(x)*y^(3/2))` devuelve `sqrt(x*y^3)`.

Si `radexpand` vale `true` y `domain` vale `real`, `rootscontract` convierte `abs` en `sqrt`, por ejemplo, `rootscontract (abs(x)*sqrt(y))` devuelve `sqrt(x^2*y)`.

La opción `rootsconmode` afecta el resultado de `rootscontract` como sigue:

Problema	Valor de rootsconmode	Resultadod de 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)`

Si `rootsconmode` vale `false`, `rootscontract` contrae sólamente respecto de exponentes racionales cuyos denominadores sean iguales. La clave para los ejemplos `rootsconmode: true` es simplemente que 2 divide a 4 pero no a 3. La asignación `rootsconmode: all` hace que se calcule el mínimo común múltiplo de los denominadores de los exponentes.

La función `rootscontract` utiliza `ratsimp` de forma similar a como lo hace `logcontract`.

Ejemplos:

```
(%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` [Variable opcional]

Valor por defecto: 1.0e-7

La variable `rootsepsilon` es la tolerancia que establece el intervalo de confianza para las raíces calculadas por la función `realroots`.

`solve (`*expr*`, `*x*`)` [Función]
`solve (`*expr*`)` [Función]
`solve ([`*eqn_1*`, ..., `*eqn_n*`], [`*x_1*`, ..., `*x_n*`])` [Función]

Resuelve la ecuación algebraica *expr* de incógnita *x* y devuelve una lista de igualdades con la *x* despejada. Si *expr* no es una igualdad, se supone que se quiere resolver la ecuación *expr* `= 0`. El argumento *x* puede ser una función (por ejemplo, `f(x)`), u otra expresión no atómica, excepto una suma o producto. Puede omitirse *x* si *expr* contiene solamente una variable. El argumento *expr* puede ser una expresión racional y puede contener funciones trigonométricas, exponenciales, etc.

Se utiliza el siguiente método de resolución:

Sea E la expresión y X la incógnita. Si E es lineal respecto de X entonces X se resuelve de forma trivial. En caso contrario, si E es de la forma `A*X^N + B` entonces el resultado es `(-B/A)^1/N)` multiplicado por las `N`-ésimas raíces de la unidad.

Si E no es lineal respecto de X entonces el máximo común divisor de los exponentes de X en E (supóngase que es N) se divide entre los exponentes y la multiplicidad de las raíces se multiplica por N. Entonces es llamado recursivamente `solve` para este resultado. Si E es factorizable entonces `solve` es invocado para cada uno de los factores. Finalmente, `solve` usará, según sea necesario, las fórmulas cuadrática, cúbica o cuártica.

En caso de que E sea un polinomio respecto de una función de la incógnita, por ejemplo `F(X)`, entonces se calcula primero para `F(X)` (sea C el resultado obtenido), entonces la ecuación `F(X)=C` se resuelve para X en el supuesto que se conozca la inversa de la función F.

Si la variable `breakup` vale `false` hará que `solve` muestre las soluciones de las ecuaciones cúbicas o cuárticas como expresiones únicas, en lugar de utilizar varias subexpresiones comunes, que es el formato por defecto.

A la variable `multiplicities` se le asignará una lista con las multiplicidades de las soluciones individuales devueltas por `solve`, `realroots` o `allroots`. La instrucción `apropos (solve)` hará que se muestren las variables optativas que de algún modo afectan al comportamiento de `solve`. Se podrá luego utilizar la función `describe` para aquellas variables cuyo objeto no esté claro.

La llamada `solve ([`*eqn_1*`, ..., `*eqn_n*`], [`*x_1*`, ..., `*x_n*`])` resuelve un sistema de ecuaciones polinómicas simultáneas (lineales o no) llamando a `linsolve` o `algsys` y

devuelve una lista de listas con soluciones para las incógnitas. En caso de haberse llamado a `linsolve` esta lista contendrá una única lista de soluciones. La llamada a `solve` tiene dos listas como argumentos. La primera lista tiene las ecuaciones a resolver y la segunda son las incógnitas cuyos valores se quieren calcular. Si el número de variables en las ecuaciones es igual al número de incógnitas, el segundo argumento puede omitirse.

Si `programmode` vale `false`, `solve` muestra la solución con etiquetas de expresiones intermedias (`%t`) y devuelve las lista de etiquetas.

Si `globalsolve` vale `true` y el problema consiste en resolver un sistema de dos o más ecuaciones lineales, a cada incógnita se le asigna el valor encontrado en la resolución del sistema.

Ejemplos:

```
(%i1) solve (asin (cos (3*x))*(f(x) - 1), x);

SOLVE is using arc-trig functions to get a solution.
Some solutions will be lost.
                            %pi
(%o1)                   [x = ---, f(x) = 1]
                             6
(%i2) ev (solve (5^f(x) = 125, f(x)), solveradcan);
                                log(125)
(%o2)                    [f(x) = --------]
                                 log(5)
(%i3) [4*x^2 - y^2 = 12, x*y - x = 2];
                       2    2
(%o3)              [4 x  - y  = 12, x y - x = 2]
(%i4) solve (%, [x, y]);
(%o4) [[x = 2, y = 2], [x = .5202594388652008 %i

 - .1331240357358706, y = .0767837852378778

 - 3.608003221870287 %i], [x = - .5202594388652008 %i

 - .1331240357358706, y = 3.608003221870287 %i

 + .0767837852378778], [x = - 1.733751846381093,

y = - .1535675710019696]]
(%i5) solve (1 + a*x + x^3, x);
                                          3
              sqrt(3) %i   1   sqrt(4 a  + 27)   1 1/3
(%o5) [x = (- ---------- - -) (--------------- - -)
                  2        2      6 sqrt(3)      2

        sqrt(3) %i   1
       (---------- - -) a
```

```
           2       2
- -------------------------, x =
             3
     sqrt(4 a  + 27)   1 1/3
  3 (--------------- - -)
         6 sqrt(3)      2

                          3
 sqrt(3) %i   1   sqrt(4 a  + 27)   1 1/3
(---------- - -) (--------------- - -)
     2        2      6 sqrt(3)      2

         sqrt(3) %i   1
      (- ---------- - -) a
             2        2
- -------------------------, x =
             3
     sqrt(4 a  + 27)   1 1/3
  3 (--------------- - -)
         6 sqrt(3)      2

         3
 sqrt(4 a  + 27)   1 1/3               a
(--------------- - -)    - -------------------------]
    6 sqrt(3)      2                  3
                              sqrt(4 a  + 27)   1 1/3
                            3 (--------------- - -)
                                  6 sqrt(3)      2
(%i6) solve (x^3 - 1);
             sqrt(3) %i - 1        sqrt(3) %i + 1
(%o6)   [x = --------------, x = - --------------, x = 1]
                   2                     2
(%i7) solve (x^6 - 1);
           sqrt(3) %i + 1      sqrt(3) %i - 1
(%o7) [x = --------------, x = --------------, x = - 1,
                 2                   2

                    sqrt(3) %i + 1        sqrt(3) %i - 1
              x = - --------------, x = - --------------, x = 1]
                          2                     2
(%i8) ev (x^6 - 1, %[1]);
                                      6
                     (sqrt(3) %i + 1)
(%o8)                ---------------- - 1
                            64
(%i9) expand (%);
(%o9)                         0
```

```
(%i10) x^2 - 1;
                                2
(%o10)                         x  - 1
(%i11) solve (%, x);
(%o11)                  [x = - 1, x = 1]
(%i12) ev (%th(2), %[1]);
(%o12)                           0
```

Los í `%r` se utilizan para indicar parámetros en las soluciones.

```
(%i1) solve([x+y=1,2*x+2*y=2],[x,y]);

solve: dependent equations eliminated: (2)
(%o1)                      [[x = 1 - %r1, y = %r1]]
```

Véanse `algsys` y `%rnum_list` para más información.

`solvedecomposes` [Variable opcional]

Valor por defecto: `true`

Si `solvedecomposes` vale `true`, `solve` llama a `polydecomp` en caso de que se le pida resolver ecuaciones polinómicas.

`solveexplicit` [Variable opcional]

Valor por defecto: `false`

Si `solveexplicit` vale `true`, le inhibe a `solve` devolver soluciones implícitas, esto es, soluciones de la forma `F(x) = 0`, donde `F` es cierta función.

`solvefactors` [Variable opcional]

Valor por defecto: `true`

Si `solvefactors` vale `false`, `solve` no intenta factorizar la expresión. Este valor `false` puede ser útil en algunos casos en los que la factorización no es necesaria.

`solvenullwarn` [Variable opcional]

Valor por defecto: `true`

Si `solvenullwarn` vale `true`, `solve` muestra un mensaje de aviso si es llamado con una lista de ecuaciones vacía o con una lista de incógnitas vacía. Por ejemplo, `solve ([], [])` imprimirá dos mensajes de aviso y devolverá `[]`.

`solveradcan` [Variable opcional]

Valor por defecto: `false`

Si `solveradcan` vale `true`, `solve` llama a `radcan`, lo que hará que `solve` se ejecute de forma más lenta, pero permitirá que se resuelvan ciertas ecuaciones que contengan exponenciales y logaritmos.

`solvetrigwarn` [Variable opcional]

Valor por defecto: `true`

Si `solvetrigwarn` vale `true`, `solve` puede presentar un mensaje diciendo que está utilizando funciones trigonométricas inversas para resolver la ecuación, y que por lo tanto puede estar ignorando algunas soluciones.

21 Ecuaciones Diferenciales

21.1 Introducción a las ecuaciones diferenciales

Esta sección describe las funciones disponibles en Maxima para el cálculo de las soluciones analíticas de ciertos tipos de ecuaciones de primer y segundo orden. Para las soluciones numéricas de sistemas de ecuaciones diferenciales, véase el paquete adicional `dynamics`. Para las representaciones gráficas en el espacio de fases, véase el paquete `plotdf`.

21.2 Funciones y variables para ecuaciones diferenciales.

`bc2` (*soluc*, *xval1*, *yval1*, *xval2*, *yval2*) [Función]

Resuelve el problema del valor en la frontera para ecuaciones diferenciales de segundo orden. Aquí, *soluc* es una solución general de la ecuación, como las que calcula `ode2`, *xval1* especifica el valor de la variable independiente en el primer punto mediante una expresión de la forma *x = x0*, mientras que *yval1* hace lo propio para la variable dependiente. Las expresiones *xval2* y *yval2* dan los valores de estas mismas variables en un segundo punto, utilizando el mismo formato.

`desolve` (*ecu*, *x*) [Función]

`desolve` ([*eqn_1*, ..., *eqn_n*], [*x_1*, ..., *x_n*]) [Función]

La función `desolve` resuelve sistemas de ecuaciones diferenciales ordinarias lineales utilizando la transformada de Laplace. Aquí las *eqi* (`i`=1,..,n) son ecuaciones diferenciales con variables dependientes *x_1*, ..., *x_n*. La dependencia funcional de *x_1*, ..., *x_n* respecto de una variable independiente, por ejemplo *x*, debe indicarse explícitamente, tanto en las variables como en las derivadas. Por ejemplo,

```
eqn_1: 'diff(f,x,2) = sin(x) + 'diff(g,x);
eqn_2: 'diff(f,x) + x^2 - f = 2*'diff(g,x,2);
```

no es el formato apropiado. El método correcto es

```
eqn_1: 'diff(f(x),x,2) = sin(x) + 'diff(g(x),x);
eqn_2: 'diff(f(x),x) + x^2 - f(x) = 2*'diff(g(x),x,2);
```

La llamada a la función `desolve` sería entonces

```
desolve([eqn_1, eqn_2], [f(x),g(x)]);
```

Si las condiciones iniciales en `x=0` son conocidas, deben ser suministradas antes de llamar a `desolve` haciendo uso previo de la función `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)]
```

Si `desolve` no encuentra una solución, entonces devuelve `false`.

`ic1` (*soluc*, *xval*, *yval*) [Función]

Resuelve el problema del valor inicial en ecuaciones diferenciales de primer orden. Aquí, *soluc* es una solución general de la ecuación, como las que calcula `ode2`, *xval* es una ecuación de la forma *x = x0* para la variable independiente y *yval* es una ecuación de la forma *y = y0* para la variable dependiente. Véase `ode2` para un ejemplo sobre su utilización.

`ic2` (*soluc*, *xval*, *yval*, *dval*) [Función]

Resuelve el problema del valor inicial en ecuaciones diferenciales de segundo orden. Aquí, *soluc* es una solución general de la ecuación, como las que calcula `ode2`, *xval* es una ecuación de la forma *x = x0* para la variable independiente y *yval* es una ecuación de la forma *y = y0* para la variable dependiente, siendo *dval* una expresión de la forma `diff(y,x) = dy0` que especifica la primera derivada de la variable dependiente respecto de la independiente en el punto *xval*.

Véase `ode2` para un ejemplo de su uso.

`ode2` (*ecu*, *dvar*, *ivar*) [Función]

La función `ode2` resuelve ecuaciones diferenciales ordinarias de primer y segundo orden. Admite tres argumentos: una ecuación diferencial ordinaria *ecu*, la variable dependiente *dvar* y la variable independiente *ivar*. Si ha tenido éxito en la resolución de la ecuación, devuelve una solución, explícita o implícita, para la variable dependiente. El símbolo `%c` se utiliza para representar la constante en el caso de ecuaciones de primer orden y los símbolos `%k1` y `%k2` son las constantes de las ecuaciones de segundo orden. Si por cualquier razón `ode2` no puede calcular la solución, devolverá `false`, acompañado quizás de un mensaje de error. Los métodos utilizados para las ecuaciones de primer orden, en el orden en que se hace la tentativa de resolución son: lineal, separable, exacto (pudiendo solicitar en este caso un factor de integración), homogéneo, ecuación de Bernoulli y un método homogéneo generalizado. Para las ecuaciones de segundo orden: coeficiente constante, exacto, homogéneo lineal con coeficientes no constantes que pueden ser transformados en coeficientes constantes, ecuación equidimensional o de Euler, método de variación de parámetros y ecuaciones exentas de

las variables dependientes o independientes de manera que se puedan reducir a dos ecuaciones lineales de primer a ser resueltas secuencialmente. Durante el proceso de resolución de ecuaciones diferenciales ordinarias, ciertas variables se utilizan con el único propósito de suministrar información al usuario: `method` almacena el método utilizado para encontrar la solución (como por ejemplo `linear`), `intfactor` para el factor de integración que se haya podido utilizar, `odeindex` para el índice del método de Bernoulli o el homogéneo generalizado y `yp` para la solución particular del método de variación de parámetros.

A fin de resolver problemas con valores iniciales y problemas con valores en la frontera, la función `ic1` está disponible para ecuaciones de primer orden y las funciones `ic2` y `bc2` para problemas de valores iniciales y de frontera, respectivamente, en el caso de las ecuaciones de segundo orden.

Ejemplo:

```
(%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
```

22 Métodos numéricos

22.1 Introducción a la transformada rápida de Fourier

El paquete `fft` contiene funciones para el cálculo numérico (no simbólico) de la transformada rápida de Fourier.

22.2 Funciones y variables para la transformada rápida de Fourier

`polartorect` (*magnitude_array*, *phase_array*) [Función]

Transforma valores complejos de la forma `r %e^(%i t)` a la forma `a + b %i`, siendo r el módulo y t la fase. Ambos valores r y t son arrays unidimensionales cuyos tamños son iguales a la misma potencia de dos.

Los valores originales de los arrays de entrada son reemplazados por las partes real e imaginaria, `a` y `b`, de los correspondientes números complejos. El resultado se calcula como

```
a = r cos(t)
b = r sin(t)
```

`polartorect` es la función inversa de `recttopolar`.

Para utilizar esta función ejecútese antes `load("fft")`. Véase también `fft`.

`recttopolar` (*real_array*, *imaginary_array*) [Función]

Transforma valores complejos de la forma `a + b %i` a la forma `r %e^(%i t)`, siendo a la parte real y a la imaginaria. Ambos valores a y b son arrays unidimensionales cuyos tamños son iguales a la misma potencia de dos.

Los valores originales de los arrays de entrada son reemplazados por los módulos y las fases, `r` y `t`, de los correspondientes números complejos. El resultado se calcula como

```
r = sqrt(a^2 + b^2)
t = atan2(b, a)
```

El ángulo calculado pertence al rango de `-%pi` a `%pi`.

`recttopolar` es la función inversa de `polartorect`.

Para utilizar esta función ejecútese antes `load("fft")`. Véase también `fft`.

`inverse_fft` (*y*) [Función]

Calcula la transformada inversa rápida de Fourier.

y es una lista o array (declarado o no) que contiene los datos a transformar. El número de elementos debe ser una potencia de dos. Los elementos deben ser números literales (enteros, racionales, de punto flotante o decimales grandes), constantes simbólicas, expresiones del tipo `a + b*%i`, siendo `a` y `b` números literales, o constantes simbólicas.

La función `inverse_fft` devuelve un nuevo objeto del mismo tipo que *y*, el cual no se ve modificado. Los resultados se calculan siempre como decimales o expresiones `a + b*%i`, siendo `a` y `b` decimales.

La transformada inversa discreta de Fourier se define como se indica a continuación. Si `x` es el resultado de la transformada inversa, entonces para `j` entre 0 y `n - 1` se tiene

```
x[j] = sum(y[k] exp(-2 %i %pi j k / n), k, 0, n - 1)
```

Para utilizar esta función ejecútese antes `load("fft")`.

Véanse también `fft` (transformada directa), `recttopolar` y `polartorect`.

Ejemplos:

Datos reales.

```
(%i1) load ("fft") $
(%i2) fpprintprec : 4 $
(%i3) L : [1, 2, 3, 4, -1, -2, -3, -4] $
(%i4) L1 : inverse_fft (L);
(%o4) [0.0, 14.49 %i - .8284, 0.0, 2.485 %i + 4.828, 0.0,
                        4.828 - 2.485 %i, 0.0, - 14.49 %i - .8284]
(%i5) L2 : fft (L1);
(%o5) [1.0, 2.0 - 2.168L-19 %i, 3.0 - 7.525L-20 %i,
4.0 - 4.256L-19 %i, - 1.0, 2.168L-19 %i - 2.0,
7.525L-20 %i - 3.0, 4.256L-19 %i - 4.0]
(%i6) lmax (abs (L2 - L));
(%o6)                        3.545L-16
```

Datos complejos.

```
(%i1) load ("fft") $
(%i2) fpprintprec : 4 $
(%i3) L : [1, 1 + %i, 1 - %i, -1, -1, 1 - %i, 1 + %i, 1] $
(%i4) L1 : inverse_fft (L);
(%o4) [4.0, 2.711L-19 %i + 4.0, 2.0 %i - 2.0,
- 2.828 %i - 2.828, 0.0, 5.421L-20 %i + 4.0, - 2.0 %i - 2.0,
2.828 %i + 2.828]
(%i5) L2 : fft (L1);
(%o5) [4.066E-20 %i + 1.0, 1.0 %i + 1.0, 1.0 - 1.0 %i,
1.55L-19 %i - 1.0, - 4.066E-20 %i - 1.0, 1.0 - 1.0 %i,
1.0 %i + 1.0, 1.0 - 7.368L-20 %i]
(%i6) lmax (abs (L2 - L));
(%o6)                        6.841L-17
```

`fft` (*x*) [Función]

Calcula la transformada rápida compleja de Fourier.

x es una lista o array (declarado o no) que contiene los datos a transformar. El número de elementos debe ser una potencia de dos. Los elementos deben ser números literales (enteros, racionales, de punto flotante o decimales grandes), constantes simbólicas, expresiones del tipo `a + b*%i`, siendo `a` y `b` números literales, o constantes simbólicas.

La función `fft` devuelve un nuevo objeto del mismo tipo que *x*, el cual no se ve modificado. Los resultados se calculan siempre como decimales o expresiones `a + b*%i`, siendo `a` y `b` decimales.

La transformada discreta de Fourier se define como se indica a continuación. Si `y` es el resultado de la transformada inversa, entonces para `k` entre 0 y `n - 1` se tiene

```
y[k] = (1/n) sum(x[j] exp(+2 %i %pi j k / n), j, 0, n - 1)
```

Si los datos x son reales, los coeficientes reales `a` y `b` se pueden calcular de manera que

```
x[j] = sum(a[k]*cos(2*%pi*j*k/n)+b[k]*sin(2*%pi*j*k/n), k, 0, n/2)
```

con

```
a[0] = realpart (y[0])
b[0] = 0
```

y, para k entre 1 y n/2 - 1,

```
a[k] = realpart (y[k] + y[n - k])
b[k] = imagpart (y[n - k] - y[k])
```

y

```
a[n/2] = realpart (y[n/2])
b[n/2] = 0
```

Para utilizar esta función ejecútese antes `load("fft")`.

Véanse también `inverse_fft` (transformada inversa), `recttopolar` y `polartorect`.

Ejemplos:

Datos reales.

```
(%i1) load ("fft") $
(%i2) fpprintprec : 4 $
(%i3) L : [1, 2, 3, 4, -1, -2, -3, -4] $
(%i4) L1 : fft (L);
(%o4) [0.0, - 1.811 %i - .1036, 0.0, .6036 - .3107 %i, 0.0,
                           .3107 %i + .6036, 0.0, 1.811 %i - .1036]
(%i5) L2 : inverse_fft (L1);
(%o5) [1.0, 2.168L-19 %i + 2.0, 7.525L-20 %i + 3.0,
4.256L-19 %i + 4.0, - 1.0, - 2.168L-19 %i - 2.0,
- 7.525L-20 %i - 3.0, - 4.256L-19 %i - 4.0]
(%i6) lmax (abs (L2 - L));
(%o6)                        3.545L-16
```

Datos complejos.

```
(%i1) load ("fft") $
(%i2) fpprintprec : 4 $
(%i3) L : [1, 1 + %i, 1 - %i, -1, -1, 1 - %i, 1 + %i, 1] $
(%i4) L1 : fft (L);
(%o4) [0.5, .3536 %i + .3536, - 0.25 %i - 0.25,
0.5 - 6.776L-21 %i, 0.0, - .3536 %i - .3536, 0.25 %i - 0.25,
0.5 - 3.388L-20 %i]
(%i5) L2 : inverse_fft (L1);
(%o5) [1.0 - 4.066E-20 %i, 1.0 %i + 1.0, 1.0 - 1.0 %i,
- 1.008L-19 %i - 1.0, 4.066E-20 %i - 1.0, 1.0 - 1.0 %i,
1.0 %i + 1.0, 1.947L-20 %i + 1.0]
```

```
(%i6) lmax (abs (L2 - L));
(%o6)                        6.83L-17
```

Cálculo de los coeficientes del seno y coseno.

```
(%i1) load ("fft") $
(%i2) fpprintprec : 4 $
(%i3) L : [1, 2, 3, 4, 5, 6, 7, 8] $
(%i4) n : length (L) $
(%i5) x : make_array (any, n) $
(%i6) fillarray (x, L) $
(%i7) y : fft (x) $
(%i8) a : make_array (any, n/2 + 1) $
(%i9) b : make_array (any, n/2 + 1) $
(%i10) a[0] : realpart (y[0]) $
(%i11) b[0] : 0 $
(%i12) for k : 1 thru n/2 - 1 do
   (a[k] : realpart (y[k] + y[n - k]),
    b[k] : imagpart (y[n - k] - y[k]));
(%o12)                          done
(%i13) a[n/2] : y[n/2] $
(%i14) b[n/2] : 0 $
(%i15) listarray (a);
(%o15)          [4.5, - 1.0, - 1.0, - 1.0, - 0.5]
(%i16) listarray (b);
(%o16)           [0, - 2.414, - 1.0, - .4142, 0]
(%i17) f(j) := sum (a[k] * cos (2*%pi*j*k / n) + b[k] * sin (2*%pi*j*k / n), k, 0
(%i18) makelist (float (f (j)), j, 0, n - 1);
(%o18)      [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]
```

22.3 Funciones para la resolución numérica de ecuaciones

`horner` (*expr*, *x*) [Función]
`horner` (*expr*) [Función]

Cambia el formato de *expr* según la regla de Horner utilizando *x* como variable principal, si ésta se especifica. El argumento `x` se puede omitir, en cuyo caso se considerará como variable principal la de *expr* en su formato racional canónico (CRE).

La función `horner` puede mejorar las estabilidad si `expr` va a ser numéricamente evaluada. También es útil si Maxima se utiliza para generar programas que serán ejecutados en Fortran. Véase también `stringout`.

```
(%i1) expr: 1e-155*x^2 - 5.5*x + 5.2e155;
                              2
(%o1)            1.0E-155 x  - 5.5 x + 5.2E+155
(%i2) expr2: horner (%, x), keepfloat: true;
(%o2)            (1.0E-155 x - 5.5) x + 5.2E+155
(%i3) ev (expr, x=1e155);
Maxima encountered a Lisp error:
```

```
 floating point overflow

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
(%i4) ev (expr2, x=1e155);
(%o4)                        7.0E+154
```

`find_root` (*expr*, *x*, *a*, *b*, [*abserr*, *relerr*]) [Función]
`find_root` (*f*, *a*, *b*, [*abserr*, *relerr*]) [Función]
`bf_find_root` (*expr*, *x*, *a*, *b*, [*abserr*, *relerr*]) [Función]
`bf_find_root` (*f*, *a*, *b*, [*abserr*, *relerr*]) [Función]
`find_root_error` [Variable opcional]
`find_root_abs` [Variable opcional]
`find_root_rel` [Variable opcional]

Calcula una raíz de la expresión *expr* o de la función *f* en el intervalo cerrado $[a, b]$. La expresión *expr* puede ser una ecuación, en cuyo caso `find_root` busca una raíz de `lhs(`*expr*`) - rhs(`*expr*`)`.

Dado que Maxima puede evaluar *expr* o *f* en $[a, b]$, entonces, si *expr* o *f* es continua, `find_root` encuentrará la raíz buscada, o raíces, en caso de existir varias.

La función `find_root` aplica al principio la búsqueda por bipartición. Si la expresión es lo suficientemente suave, entonces `find_root` aplicará el método de interpolación lineal.

`bf_find_root` es una versión de `find_root` para números reales de precisión arbitraria (bigfloat). La función se evalúa utilizando la aritmética de estos números, devolviendo un resultado numérico de este tipo. En cualquier otro aspecto, `bf_find_root` es idéntica a `find_root`, siendo la explicación que sigue igualmente válida para `bf_find_root`.

La precisión de `find_root` está controlada por `abserr` y `relerr`, que son claves opcionales para `find_root`. Estas claves toman la forma `key=val`. Las claves disponibles son:

`abserr` Error absoluto deseado de la función en la raíz. El valor por defecto es `find_root_abs`.

`relerr` Error relativo deseado de la raíz. El valor por defecto es `find_root_rel`.

`find_root` se detiene cuando la función alcanza un valor menor o igual que `abserr`, o si las sucesivas aproximaciones *x_0*, *x_1* difieren en no más que `relerr * max(abs(x_0), abs(x_1))`. Los valores por defecto de `find_root_abs` y `find_root_rel` son ambos cero.

`find_root` espera que la función en cuestión tenga signos diferentes en los extremos del intervalo. Si la función toma valores numéricos en ambos extremos y estos números son del mismo signo, entonces el comportamiento de `find_root` se controla con `find_root_error`. Cuando `find_root_error` vale `true`, `find_root` devuelve un mensaje de error; en caso contrario, `find_root` devuelve el valor de `find_root_error`. El valor por defecto de `find_root_error` es `true`.

Si en algún momento del proceso de búsqueda *f* alcanza un valor no numérico, `find_root` devuelve una expresión parcialmente evaluada.

Se ignora el orden de a y b; la región de búsqueda es $[min(a, b), max(a, b)]$.

Ejemplos:

```
(%i1) f(x) := sin(x) - x/2;
                                          x
(%o1)                  f(x) := sin(x) - -
                                          2
(%i2) find_root (sin(x) - x/2, x, 0.1, %pi);
(%o2)                   1.895494267033981
(%i3) find_root (sin(x) = x/2, x, 0.1, %pi);
(%o3)                   1.895494267033981
(%i4) find_root (f(x), x, 0.1, %pi);
(%o4)                   1.895494267033981
(%i5) find_root (f, 0.1, %pi);
(%o5)                   1.895494267033981
(%i6) find_root (exp(x) = y, x, 0, 100);
                               x
(%o6)           find_root(%e  = y, x, 0.0, 100.0)
(%i7) find_root (exp(x) = y, x, 0, 100), y = 10;
(%o7)                   2.302585092994046
(%i8) log (10.0);
(%o8)                   2.302585092994046
(%i9) fpprec:32;
(%o9)                             32
(%i10) bf_find_root (exp(x) = y, x, 0, 100), y = 10;
(%o10)                  2.3025850929940456840179914546844b0
(%i11) log(10b0);
(%o11)                  2.3025850929940456840179914546844b0
```

`newton` (*expr*, *x*, *x_0*, *eps*) [Función]

Devuelve una solución aproximada de *expr* = 0 obtenida por el método de Newton, considerando *expr* como una función de una variable, *x*. La búsqueda comienza con *x* = *x_0* y continúa hasta que se verifique `abs(`*expr*`)` < *eps*, donde *expr* se evalúa con el valor actual de *x*.

La función `newton` permite que en *expr* haya variables no definidas, siempre y cuando la condición de terminación `abs(`*expr*`)` < *eps* pueda reducirse a un valor lógico `true` o `false`; de este modo, no es necesario que *expr* tome un valor numérico.

Ejecútese `load("newton1")` para cargar esta función.

Véanse también `realroots`, `allroots`, `find_root` y `mnewton`.

Ejemplos:

```
(%i1) load ("newton1");
(%o1) /usr/share/maxima/5.10.0cvs/share/numeric/newton1.mac
(%i2) newton (cos (u), u, 1, 1/100);
(%o2)                   1.570675277161251
(%i3) ev (cos (u), u = %);
(%o3)                 1.2104963335033528E-4
(%i4) assume (a > 0);
```

```
(%o4)                          [a > 0]
(%i5) newton (x^2 - a^2, x, a/2, a^2/100);
(%o5)                  1.00030487804878 a
(%i6) ev (x^2 - a^2, x = %);
                                                2
(%o6)                6.098490481853958E-4 a
```

22.4 Introducción a la resolución numérica de ecuaciones diferenciales

Las ecuaciones diferenciales ordinarias (EDO) que se resuelven con las funciones de esta sección deben tener la forma

$$\frac{dy}{dx} = F(x, y)$$

la cual es una EDO de primer orden. Las ecuaciones diferenciales de orden n deben escribirse como un sistema de n ecuaciones de primer orden del tipo anterior. Por ejemplo, una EDO de segundo orden debe escribirse como un sistema de dos ecuaciones,

$$\frac{dx}{dt} = G(x, y, t) \qquad \frac{dy}{dt} = F(x, y, t)$$

El primer argumento de las funciones debe ser una lista con las expresiones de los miembros derechos de las EDOs. Las variables cuyas derivadas se representan por las expresiones anteriores deben darse en una segunda lista. En el caso antes citado, las variables son x y y. La variable independiente, t en los mismos ejemplos anteriores, pueden darse mediante una opción adicional. Si las expresiones dadas no dependen de esa variable independiente, el sistema recibe el nombre de autónomo.

22.5 Funciones para la resolución numérica de ecuaciones diferenciales

`plotdf` (*dydx*, *...options...*) [Función]
`plotdf` (*dvdu*, [*u*,*v*], *...options...*) [Función]
`plotdf` ([*dxdt*,*dydt*], *...options...*) [Función]
`plotdf` ([*dudt*,*dvdt*], [*u*,*v*], *...options...*) [Función]

Dibuja un campo de direcciones en dos dimensiones x y y.

dydx, *dxdt* y *dydt* son expresiones que dependen de x y y. Además de esas dos variables, las dos expresiones pueden depender de un conjunto de parámetros, con valores numéricos que son dados por medio de la opción `parameters` (la sintaxis de esa opción se explica mas al frente), o con un rango de posibles valores definidos con la opción *sliders*.

Varias otras opciones se pueden incluir dentro del comando, o seleccionadas en el menú. Haciendo click en un punto del gráfico se puede hacer que sea dibujada la curva integral que pasa por ese punto; lo mismo puede ser hecho dando las coordenadas del punto con la opción `trajectory_at` dentro del comando plotdf. La dirección de integración se puede controlar con la opción `direction`, que acepta valores de *forward*, *backward* ou *both*. El número de pasos realizado en la integración numérica se controla

con la opción `nsteps` y el incremento del tiempo en cada paso con la opción `tstep`. Se usa el método de Adams Moulton para hacer la integración numérica; también es posible cambiar para el método de Runge-Kutta de cuarto orden con ajuste de pasos.

Menú de la ventana del gráfico:

El menú de la ventana gráfica dispone de las siguientes opciones: *Zoom*, que permite cambiar el comportamiento del ratón, de manera que hará posible el hacer zoom en la región del gráfico haciendo clic con el botón izquierdo. Cada clic agranda la imagen manteniendo como centro de la misma el punto sobre el cual se ha hecho clic. Manteniendo pulsada la tecla `Shift` mientras se hace clic, retrocede al tamaño anterior. Para reanudar el cálculo de las trayectorias cuando se hace clic, seleccine la opción *Integrate* del menú.

La opción *Config* del menú se puede utilizar para cambiar la(s) EDO(S) y algunos otros ajustes. Después de hacer los cambios, se debe utilizar la opción *Replot* para activar los nuevos ajustes. Si en el campo *Trajectory at* del menú de diálogo de *Config* se introducen un par de coordenadas y luego se pulsa la tecla `retorno`, se mostrará una nueva curva integral, además de las ya dibujadas. Si se selecciona la opción *Replot*, sólo se mostrará la última curva integral seleccionada.

Manteniendo pulsado el botón derecho del ratón mientras se mueve el cursor, se puede arrastrar el gráfico horizontal y verticalmente. Otros parámetros, como pueden ser el número de pasos, el valor inicial de t, las coordenadas del centro y el radio, pueden cambiarse en el submenú de la opción *Config*.

Con la opción *Save*, se puede obtener una copia del gráfico en una impresora Postscript o guardarlo en un fichero Postscript. Para optar entre la impresión o guardar en fichero, se debe seleccionar *Print Options* en la ventana de diálogo de *Config*. Una vez cubiertos los campos de la ventana de diálogo de *Save*, será necesario seleccionar la opción *Save* del primer menú para crear el fichero o imprimir el gráfico.

Opciones gráficas:

La función `plotdf` admite varias opciones, cada una de las cuales es una lista de dos o más elementos. El primer elemento es el nombre de la opción, y el resto está formado por el valor o valores asignados a dicha opción.

La función `plotdf` reconoce las siguientes opciones:

- *tstep* establece la amplitud de los incrementos en la variable independiente t, utilizados para calcular la curva integral. Si se aporta sólo una expresión *dydx*, la variable x será directamente proporcional a t. El valor por defecto es 0.1.
- *nsteps* establece el número de pasos de longitud `tstep` que se utilizarán en la variable independiente para calcular la curva integral. El valor por defecto es 100.
- *direction* establece la dirección de la variable independiente que será seguida para calcular una curva integral. Valores posibles son: `forward`, para hacer que la variable independiente aumente `nsteps` veces, con incrementos `tstep`; `backward`, para hacer que la variable independiente disminuya; `both`, para extender la curva integral `nsteps` pasos hacia adelante y `nsteps` pasos hacia atrás. Las palabras `right` y `left` se pueden utilizar como sinónimos de `forward` y `backward`. El valor por defecto es `both`.

- *tinitial* establece el valor inicial de la variable t utilizado para calcular curvas integrales. Puesto que las ecuaciones diferenciales son autónomas, esta opción sólo aparecerá en los gráficos de las curvas como funciones de t. El valor por defecto es 0.
- *versus_t* se utiliza para crear una segunda ventana gráfica, con el gráfico de una curva integral, como dos funciones x, y, de variable independiente t. Si se le da a `versus_t` cualquier valor diferente de 0, se mostrará la segunda ventana gráfica, la cual incluye otro menú, similar al de la ventana principal. El valor por defecto es 0.
- *trajectory_at* establece las coordenadas *xinitial* y *yinitial* para el extremo inicial de la curva integral. No tiene asignado valor por defecto.
- *parameters* establece una lista de parámetros, junto con sus valores numéricos, que son utilizados en la definición de la ecuación diferencial. Los nombres de los parámetros y sus valores deben escribirse en formato de cadena de caracteres como una secuencia de pares `nombre=valor` separados por comas.
- *sliders* establece una lista de parámetros que se cambiarán interactivamente utilizando barras de deslizamiento, así como los rangos de variación de dichos parámetros. Los nombres de los parámetros y sus rangos deben escribirse en formato de cadena de caracteres como una secuencia de pares `nombre=min:max` separados por comas.
- *xfun* establece una cadena de caracteres con funciones de x separadas por puntos y comas para ser representadas por encima del campo de direcciones. Estas funciones serán interpretadas por Tcl, no por Maxima.
- *xradius* es la mitad de la longitud del rango de valores a representar en la dirección x. El valor por defecto es 10.
- *yradius* es la mitad de la longitud del rango de valores a representar en la dirección y. El valor por defecto es 10.
- *xcenter* es la coordenada x del punto situado en el centro del gráfico. El valor por defecto es 0.
- *ycenter* es la coordenada y del punto situado en el centro del gráfico. El valor por defecto es 0.
- *width* establece el ancho de la ventana gráfica en píxeles. El valor por defecto es 500.
- *height* establece la altura de la ventana gráfica en píxeles. El valor por defecto es 500.

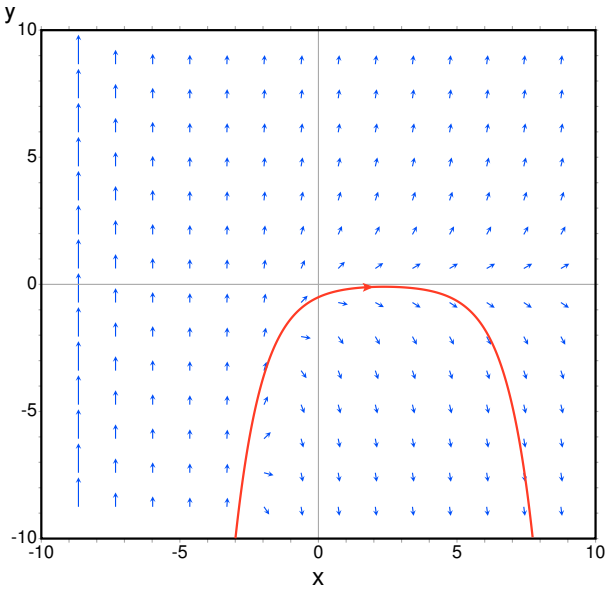
Ejemplos:

NOTA: Dependiendo de la interface que se use para Maxima, las funciones que usan `openmath`, incluida `plotdf`, pueden desencadenar un fallo si terminan en punto y coma, en vez del símbolo de dólar. Para evitar problemas, se usará el símbolo de dólar en todos ejemplos.

- Para mostrar el campo de direcciones de la ecuación diferencial $y' = exp(-x) + y$ y la solución que pasa por $(2, -0.1)$:

```
(%i1) load("plotdf")$
```

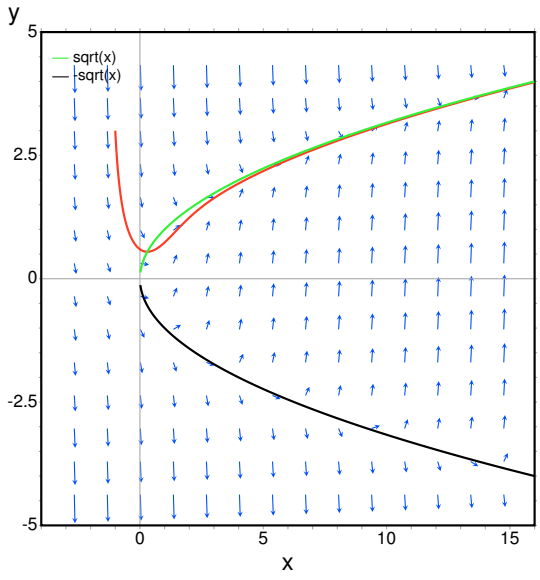
```
(%i2) plotdf(exp(-x)+y,[trajectory_at,2,-0.1]);
```



- Para mostrar el campo de direcciones de la ecuación $diff(y,x) = x - y^2$ y la solución de condición inicial $y(-1) = 3$, se puede utilizar la sentencia:

```
(%i3) plotdf(x-y^2,[xfun,"sqrt(x);-sqrt(x)"],
          [trajectory_at,-1,3], [direction,forward],
          [yradius,5],[xcenter,6]);
```

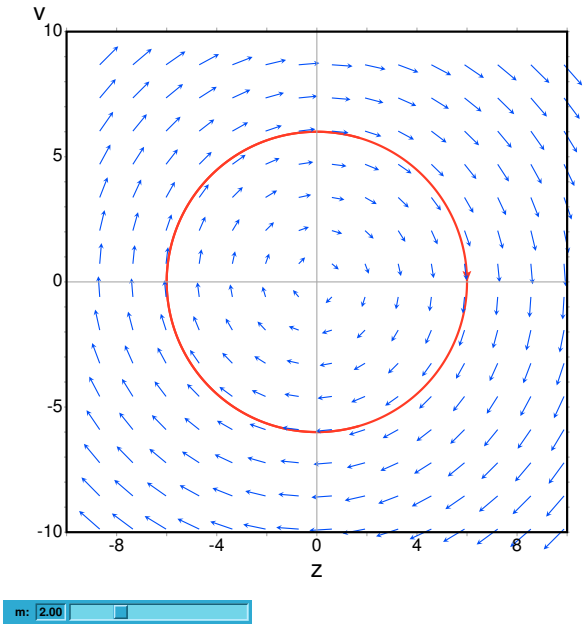
  El gráfico también muestra la función $y = sqrt(x)$.



- El siguiente ejemplo muestra el campo de direcciones de un oscilador armónico, definido por las ecuaciones $dx/dt = y$ y $dy/dt = -k*x/m$, y la curva integral que

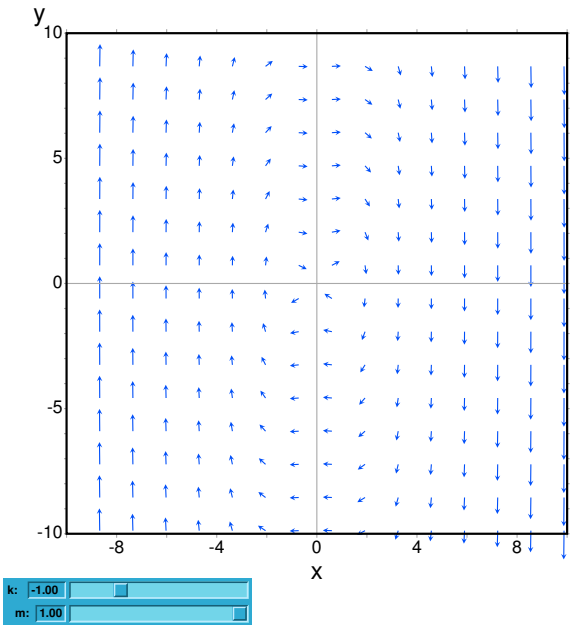
pasa por $(x, y) = (6, 0)$, con una barra de deslizamiento que permitirá cambiar el valor de m interactivamente (k permanece fijo a 2):

```
(%i4) plotdf([y,-k*x/m],[parameters,"m=2,k=2"],
             [sliders,"m=1:5"], [trajectory_at,6,0]);
```



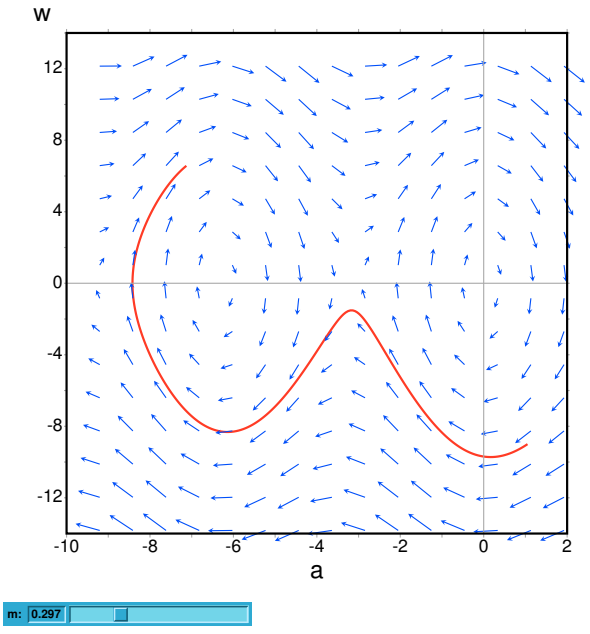
- Para representar el campo de direcciones de la ecuación de Duffing, $m * x'' + c * x' + k * x + b * x^3 = 0$, se introduce la variable $y = x'$ y se hace:

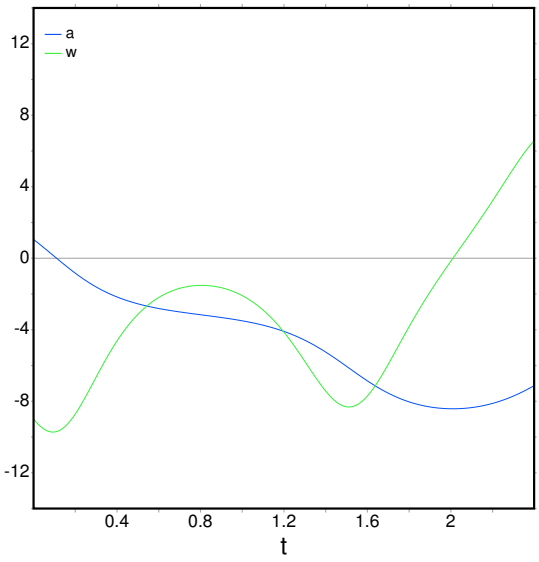
```
(%i5) plotdf([y,-(k*x + c*y + b*x^3)/m],
             [parameters,"k=-1,m=1.0,c=0,b=1"],
             [sliders,"k=-2:2,m=-1:1"],[tstep,0.1]);
```



- El campo de direcciones de un péndulo amortiguado, incluyendo la solución para condiciones iniciales dadas, con una barra de deslizamiento que se puede utilizar para cambiar el valor de la masa, m, y con el gráfico de las dos variables de estado como funciones del tiempo:

```
(%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]);
```





`ploteq` (*exp*, *...options...*) [Función]

Dibuja curvas equipotenciales para *exp*, que debe ser una expresión dependiente de dos variables. Las curvas se obtienen integrando la ecuación diferencial que define las trayectorias ortogonales a las soluciones del sistema autónomo que se obtiene del gradiente de la expresión dada. El dibujo también puede mostrar las curvas integrales de ese sistema de gradientes (opción `fieldlines`).

Este programa también necesita Xmaxima, incluso si se ejecuta Maxima desde una consola, pues el dibujo se creará por el código Tk de Xmaxima. Por defecto, la región dibujada estará vacía hasta que el usuario haga clic en un punto, dé sus coordenadas a través del menú o mediante la opción `trajectory_at`.

La mayor parte de opciones aceptadas por `plotdf` se pueden utilizar también con `ploteq` y el aspecto del interfaz es el mismo que el descrito para `plotdf`.

Ejemplo:

```
(%i1) V: 900/((x+1)^2+y^2)^(1/2)-900/((x-1)^2+y^2)^(1/2)$
(%i2) ploteq(V,[x,-2,2],[y,-2,2],[fieldlines,"blue"])$
```

Haciendo clic sobre un punto se dibujará la curva equipotencial que pasa por ese punto (en rojo) y la trayectoria ortogonal (en azul).

`rk` (*ODE*, *var*, *initial*, *dominio*) [Función]
`rk` ([*ODE1*,...,*ODEm*], [*v1*,...,*vm*], [*init1*,...,*initm*], *dominio*) [Función]

La primera forma se usa para resolver numéricamente una ecuación diferencial ordinaria de primer orden (EDO), y la segunda forma resuelve numéricamente un sistema de *m* de esas ecuaciones, usando el método de Runge-Kutta de cuarto orden. *var* representa la variable dependiente. EDO debe ser una expresión que dependa únicamente de las variables independiente y dependente, y define la derivada de la variable dependiente en función de la variable independiente.

La variable independiente se representa con *dominio*, que debe ser una lista con cuatro elementos, como por ejemplo:

```
[t, 0, 10, 0.1]
```

el primer elemento de la lista identifica la variable independiente, el segundo y tercer elementos son los valores inicial y final para esa variable, y el último elemento da el valor de los incrementos que deberán ser usados dentro de ese intervalo.

Si se van a resolver *m* ecuaciones, deberá haber *m* variables dependientes *v1*, *v2*, ..., *vm*. Los valores iniciales para esas variables serán *inic1*, *inic2*, ..., *inicm*. Continuará existiendo apenas una variable independiente definida por la lista *domain*, como en el caso anterior. *EDO1*, ..., *EDOm* son las expresiones que definen las derivadas de cada una de las variables dependientes en función de la variable independiente. Las únicas variables que pueden aparecer en cada una de esas expresiones son la variable independiente y cualquiera de las variables dependientes. Es importante que las derivadas *EDO1*, ..., *EDOm* sean colocadas en la lista en el mismo orden en que fueron agrupadas las variables dependientes; por ejemplo, el tercer elemento de la lista será interpretado como la derivada de la tercera variable dependiente.

El programa intenta integrar las ecuaciones desde el valor inicial de la variable independiente, hasta el valor final, usando incrementos fijos. Si en algún paso una de las variables dependientes toma un valor absoluto muy grande, la integración será suspendida en ese punto. El resultado será una lista con un número de elementos igual al número de iteraciones realizadas. Cada elemento en la lista de resultados es también una lista con *m*+1 elementos: el valor de la variable independiente, seguido de los valores de las variables dependientes correspondientes a ese punto.

23 Matrices y Álgebra Lineal

23.1 Introducción a las matrices y el álgebra lineal

23.1.1 Operador punto

El operador `.` realiza la multiplicación matricial y el producto escalar. Cuando los operandos son dos matrices columna o matrices fila `a` y `b`, la expresión `a.b` es equivalente a `sum (a[i]*b[i], i, 1, length(a))`. Si `a` y `b` no son complejos, estamos en el caso del producto escalar. En caso de ser `a` y `b` vectores en el campo complejo, el producto escalar se define como `conjugate(a).b`; la función `innerproduct` del paquete `eigen` realiza el producto escalar complejo.

Cuando los operandos son matrices de índole más general, el resultado que se obtiene es el producto matricial de `a` por `b`. El número de filas de `b` debe ser igual al número de columnas de `a`, y el resultado tiene un número de filas igual al de `a` y un número de columnas igual al de `b`.

Al objeto de distinguir `.` como operador aritmético del punto decimal de la notación en coma flotante, puede ser necesario dejar espacios a ambos lados. Por ejemplo, `5.e3` es `5000.0` pero `5 . e3` es `5` por `e3`.

Hay algunas variables globales que controlan la simplificación de expresiones que contengan al operador `.`, a saber, `dot`, `dot0nscsimp`, `dot0simp`, `dot1simp`, `dotassoc`, `dotconstrules`, `dotdistrib`, `dotexptsimp`, `dotident`, y `dotscrules`.

23.1.2 Vectores

El paquete `vect` define funciones para análisis vectorial. Para cargar el paquete en memoria se debe hacer `load ("vect")` y con `demo ("vect")` se presenta una demostración sobre las funciones del paquete.

El paquete de análisis vectorial puede combinar y simplificar expresiones simbólicas que incluyan productos escalares y vectoriales, junto con los operadores de gradiente, divergencia, rotacional y laplaciano. La distribución de estos operadores sobre sumas o productos se gobierna por ciertas variables, al igual que otras transformaciones, incluida la expansión en componentes en cualquier sistema de coordenadas especificado. También hay funciones para obtener el potencial escalar o vectorial de un campo.

El paquete `vect` contiene las siguientes funciones: `vectorsimp`, `scalefactors`, `express`, `potential` y `vectorpotential`.

Por defecto, el paquete `vect` no declara el operador `.` como conmutativo. Para transformarlo en conmutativo, se debe ejecutar previamente la instrucción `declare(".", commutative)`.

23.1.3 Paquete eigen

El paquete `eigen` contiene funciones para el cálculo simbólico de valores y vectores propios. Maxima carga el paquete automáticamente si se hace una llamada a cualquiera de las dos funciones `eigenvalues` o `eigenvectors`. El paquete se puede cargar de forma explícita mediante `load ("eigen")`.

La instrucción `demo ("eigen")` hace una demostración de las funciones de este paquete; `batch ("eigen")` realiza la misma demostración pero sin pausas entre los sucesivos cálculos.

Las funciones del paquete `eigen` son `innerproduct`, `unitvector`, `columnvector`, `gramschmidt`, `eigenvalues`, `eigenvectors`, `uniteigenvectors` y `similaritytransform`.

23.2 Funciones y variables para las matrices y el álgebra lineal

`addcol` (*M*, *lista_1*, ..., *lista_n*) [Función]
Añade la/s columna/s dada/s por la/s lista/s (o matrices) a la matriz *M*.

`addrow` (*M*, *lista_1*, ..., *lista_n*) [Función]
Añade la/s fila/s dada/s por la/s lista/s (o matrices) a la matriz *M*.

`adjoint` (*M*) [Función]
Devuelve el adjunto de la matriz *M*. La matriz adjunta es la transpuesta de la matriz de cofactores de *M*.

`augcoefmatrix` ([*eqn_1*, ..., *eqn_m*], [*x_1*, ..., *x_n*]) [Función]
Devuelve la matriz aumentada de coeficientes del sistema de ecuaciones lineales *eqn_1*, ..., *eqn_m* de variables *x_1*, ..., *x_n*. Se trata de la matriz de coeficientes con una columna adicional para los términos constantes de cada ecuación, es decir, aquellos términos que no dependen de las variables *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   ]
```

`cauchy_matrix` ([*x_1*,*x_2*, ..., *x_m*], [*y_1*,*y_2*, ..., *y_n*]) [Función]
`cauchy_matrix` ([*x_1*,*x_2*, ..., *x_n*]) [Función]
Devuelve una matriz de Cauchy *n* by *m* de elementos *a[i,j]* = 1/(x_*i*+*y_i*). El segundo elemento de `cauchy_matrix` es opcional, y en caso de no estar presente, los elementos serán de la forma *a[i,j]* = 1/(x_*i*+x_*j*).

Observación: en la literatura, la matriz de Cauchy se define a veces con sus elementos de la forma *a[i,j]* = 1/(*x_i*-*y_i*).

Ejemplos:

```
(%i1) cauchy_matrix([x1,x2],[y1,y2]);
                          [    1         1    ]
                          [ -------   ------- ]
                          [ y1 + x1   y2 + x1 ]
(%o1)                     [                   ]
                          [    1         1    ]
                          [ -------   ------- ]
                          [ y1 + x2   y2 + x2 ]

(%i2) cauchy_matrix([x1,x2]);
                          [   1         1    ]
                          [  ----     ------- ]
```

```
                          [  2 x1    x2 + x1 ]
(%o2)                     [                  ]
                          [    1       1     ]
                          [ -------   ----   ]
                          [ x2 + x1   2 x2   ]
```

`charpoly` (*M*, *x*) [Función]

Calcula el polinomio característico de la matriz *M* respecto de la variable *x*. Esto es, `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*]) [Función]

Devuelve la matriz de coeficientes para las variables *x_1*, ..., *x_n* del sistema de ecuaciones lineales *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*) [Función]
Devuelve la *i*-ésima columna de la matriz *M*. El resultado es una matriz de una sola columna.

`columnvector` (*L*) [Función]
`covect` (*L*) [Función]
Devuelve una matriz con una columna y `length` (*L*) filas, conteniendo los elementos de la lista *L*.

La llamada `covect` es un sinónimo de `columnvector`.

Es necesario cargar la función haciendo `load ("eigen")`.

Ejemplo:

```
(%i1) load ("eigen")$
Warning - you are redefining the Macsyma function eigenvalues
Warning - you are redefining the Macsyma function eigenvectors
(%i2) columnvector ([aa, bb, cc, dd]);
                                [ aa ]
                                [    ]
                                [ bb ]
(%o2)                           [    ]
                                [ cc ]
                                [    ]
                                [ dd ]
```

`copymatrix` (*M*) [Función]
Devuelve una copia de la matriz *M*. Esta es la única manera de obtener una réplica de *M* además de la de copiar elemento a elemento.

Nótese que una asignación de una matriz a otra, como en `m2: m1`, no hace una copia de `m1`. Asignaciones del tipo `m2 [i,j]: x` o `setelmx (x, i, j, m2` también modifica `m1 [i,j]`. Si se crea una copia con `copymatrix` y luego se hacen asignaciones se tendrá una copia separada y modificada.

`determinant` (*M*) [Función]
Calcula el determinante de *M* por un método similar al de eliminación de Gauss

La forma del resultado depende del valor asignado a `ratmx`.

Existe una rutina especial para calcular determinantes de matrices con elementos dispersas, la cual será invocada cuando las variables `ratmx` y `sparse` valgan ambas `true`.

`detout` [Variable opcional]
Valor por defecto: `false`

Cuando `detout` vale `true`, el determinante de la matriz cuya inversa se calcula aparece como un factor fuera de la matriz.

Para que esta variable surta efecto, `doallmxops` y `doscmxops` deberían tener el valor `false` (véanse sus descripciones). Alternativamente, esta variable puede ser suministrada a `ev`.

Ejemplo:

```
(%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*) [Función]

Devuelve una matriz diagonal de orden *n* con los elementos de la diagonal todos ellos iguales a *x*. La llamada `diagmatrix (n, 1)` devuelve una matriz identidad (igual que `ident (n)`).

La variable *n* debe ser un número entero, en caso contrario `diagmatrix` envía un mensaje de error.

x puede ser cualquier tipo de expresión, incluso otra matriz. Si *x* es una matriz, no se copia; todos los elementos de la diagonal son iguales a *x*.

`doallmxops` [Variable opcional]

Valor por defecto: `true`

Cuando `doallmxops` vale `true`, todas las operaciones relacionadas con matrices son llevadas a cabo. Cuando es `false`, entonces las selecciones para `dot` controlan las operaciones a ejecutar.

`domxexpt` [Variable opcional]

Valor por defecto: `true`

Cuando `domxexpt` vale `true`, un exponente matricial, como `exp (M)` donde M es una matriz, se interpreta como una matriz cuyo elemento `[i,j]` es igual a `exp (m[i,j])`. En otro caso, `exp (M)` se evalúa como `exp (ev(M))`.

La variable `domxexpt` afecta a todas las expresiones de la forma *`base^exponente`* donde *base* es una expresión escalar o constante y *exponente* es una lista o matriz.

Ejemplo:

```
(%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` [Variable opcional]

Valor por defecto: `true`

Cuando `domxmxops` vale `true`, se realizan todas las operaciones entre matrices o entre matrices y listas (pero no las operaciones entre matrices y escalares); si esta variable es `false` tales operaciones no se realizan.

`domxnctimes` [Variable opcional]

Valor por defecto: `false`

Cuando `domxnctimes` vale `true`, se calculan los productos no conmutativos entre matrices.

`dontfactor` [Variable opcional]

Valor por defecto: `[]`

En `dontfactor` puede guardarse una lista de variables respecto de las cuales no se realizarán factorizaciones. Inicialmente, la lista está vacía.

`doscmxops` [Variable opcional]

Valor por defecto: `false`

Cuando `doscmxops` vale `true`, se realizan las operaciones entre escalares y matrices.

`doscmxplus` [Variable opcional]

Valor por defecto: `false`

Cuando `doscmxplus` vale `true`, las operaciones entre escalares y matrices dan como resultado una matriz.

`dot0nscsimp` [Variable opcional]

Valor por defecto: `true`

(Esta descripción no está clara en la versión inglesa original.)

`dotassoc` [Variable opcional]

Valor por defecto: `true`

Cuando `dotassoc` vale `true`, una expresión como `(A.B).C` se transforma en `A.(B.C)`.

`dotconstrules` [Variable opcional]

Valor por defecto: `true`

Cuando `dotconstrules` vale `true`, un producto no conmutativo de una constante con otro término se transforma en un producto conmutativo.

`dotdistrib` [Variable opcional]

Valor por defecto: `false`

Cuando `dotdistrib` vale `true`, una expresión como `A.(B + C)` se transforma en `A.B + A.C`.

`dotexptsimp` [Variable opcional]

Valor por defecto: `true`

Cuando `dotexptsimp` vale `true`, una expresión como `A.A` se transforma en `A^^2`.

`dotident` [Variable opcional]

Valor por defecto: 1

El valor de la variable `dotident` es el resultado devuelto por `X^^0`.

`dotscrules` [Variable opcional]

Valor por defecto: `false`

Cuando `dotscrules` vale `true`, una expresión como `A.SC` o `SC.A` se transforma en `SC*A` y `A.(SC*B)` en `SC*(A.B)`.

`echelon` (*M*) [Función]

Devuelve la forma escalonada de la matriz *M*, obtenida por eliminación gaussiana. La forma escalonada se calcula a partir de *M* mediante operaciones elementales con sus filas, de tal manera que el primer elemento no nulo de cada fila en la matriz resultado es la unidad y que cada elemento de la columna por debajo del primer uno de cada fila sean todos ceros.

La función `triangularize` también lleva a cabo la eliminación gaussiana, pero no normaliza el primer elemento no nulo de cada fila.

Otras funciones, como `lu_factor` y `cholesky`, también dan como resultados matrices 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*) [Función]
`eivals` (*M*) [Función]

Devuelve una lista con dos sublistas. La primera sublista la forman los valores propios de la matriz *M* y la segunda sus multiplicidades correspondientes.

El nombre `eivals` es un sinónimo de `eigenvalues`.

La función `eigenvalues` llama a la función `solve` para calcular las raíces del polinomio característico de la matriz. En ocasiones, `solve` no podrá encontrar dichas raíces, en cuyo caso otras funciones de este paquete no trabajarán correctamente, a excepción de `innerproduct`, `unitvector`, `columnvector` y `gramschmidt`.

En algunos casos los valores propios encontrados por `solve` serán expresiones complicadas, las cuales se podrán simplificar haciendo uso de otras funciones.

El paquete `eigen.mac` se carga en memoria de forma automática cuando se invocan `eigenvalues` o `eigenvectors`. Si `eigen.mac` no está ya cargado, `load ("eigen")` lo carga. Tras la carga, todas las funciones y variables del paquete estarán activas.

`eigenvectors` (*M*) [Función]
`eivects` (*M*) [Función]

Calcula los vectores propios de la matriz *M*. El resultado devuelto es una lista con dos elementos; el primero está formado por dos listas, la primera con los valores propios de *M* y la segunda con sus respectivas multiplicidades, el segundo elemento es una lista de listas de vectores propios, una por cada valor propio, pudiendo haber uno o más vectores propios en cada lista.

Tomando la matriz *M* como argumento, devuelve una lista de listas, la primera de las cuales es la salida de `eigenvalues` y las siguientes son los vectorios propios de la matriz asociados a los valores propios correspondientes. Los vectores propios calculados son los vectores propios por la derecha.

El nombre `eivects` es un sinónimo de `eigenvectors`.

El paquete `eigen.mac` se carga en memoria de forma automática cuando se invocan `eigenvalues` o `eigenvectors`. Si `eigen.mac` no está ya cargado, `load ("eigen")` lo carga. Tras la carga, todas las funciones y variables del paquete estarán activas.

Las variables que afectan a esta función son:

`nondiagonalizable` toma el valor `true` o `false` dependiendo de si la matriz no es diagonalizable o diagonalizable tras la ejecución de `eigenvectors`.

`hermitianmatrix`, si vale `true`, entonces los vectores propios degenerados de la matriz hermítica son ortogonalizados mediante el algoritmo de Gram-Schmidt.

`knowneigvals`, si vale `true`, entonces el paquete `eigen` da por sentado que los valores propios de la matriz son conocidos por el usuario y almacenados en la variable global `listeigvals`. `listeigvals` debería ser similar a la salida de `eigenvalues`.

La función `algsys` se utiliza aquí para calcular los vectores propios. A veces, `algsys` no podrá calcular una solución. En algunos casos, será posible simplificar los valores propios calculándolos en primer lugar con `eigenvalues` y luego utilizando otras funciones para simplificarlos. Tras la simplificación, `eigenvectors` podrá ser llamada otra vez con la variable `knowneigvals` ajustada al valor `true`.

Véase también `eigenvalues`.

Ejemplos:

Una matriz con un único vector propio por cada valor propio.

```
(%i1) M1 : matrix ([11, -1], [1, 7]);
                             [ 11  - 1 ]
(%o1)                        [         ]
                             [ 1    7  ]
(%i2) [vals, vecs] : eigenvectors (M1);
(%o2) [[[9 - sqrt(3), sqrt(3) + 9], [1, 1]],
                          [[[1, sqrt(3) + 2]], [[1, 2 - sqrt(3)]]]]
(%i3) for i thru length (vals[1]) do disp (val[i] = vals[1][i],
  mult[i] = vals[2][i], vec[i] = vecs[i]);
                        val  = 9 - sqrt(3)
                           1

                             mult  = 1
                                 1

                      vec  = [[1, sqrt(3) + 2]]
                         1

                        val  = sqrt(3) + 9
                           2

                             mult  = 1
                                 2

                      vec  = [[1, 2 - sqrt(3)]]
                         2

(%o3)                           done
```

Una matriz con dos vectores propios para uno de los valores propios.

```
(%i1) M1 : matrix ([0, 1, 0, 0], [0, 0, 0, 0], [0, 0, 2, 0], [0, 0, 0, 2]);
                           [ 0  1  0  0 ]
                           [            ]
                           [ 0  0  0  0 ]
(%o1)                      [            ]
                           [ 0  0  2  0 ]
                           [            ]
                           [ 0  0  0  2 ]
(%i2) [vals, vecs] : eigenvectors (M1);
(%o2) [[[0, 2], [2, 2]], [[[1, 0, 0, 0]],
                                    [[0, 0, 1, 0], [0, 0, 0, 1]]]]
(%i3) for i thru length (vals[1]) do disp (val[i] = vals[1][i],
  mult[i] = vals[2][i], vec[i] = vecs[i]);
                             val  = 0
                                1
```

```
                                  mult  = 2
                                      1

                               vec  = [[1, 0, 0, 0]]
                                  1

                                  val  = 2
                                     2

                                  mult  = 2
                                      2

                            vec  = [[0, 0, 1, 0], [0, 0, 0, 1]]
                               2

(%o3)                                done
```

ematrix (*m*, *n*, *x*, *i*, *j*) [Función]

Devuelve una matriz de orden *m* por *n*, con todos sus elementos nulos, excepto el que ocupa la posición `[i, j]`, que es igual a *x*.

entermatrix (*m*, *n*) [Función]

Devuelve una matriz de orden *m* por *n*, cuyos elementos son leidos de forma interactiva.

Si *n* es igual a *m*, Maxima pregunta por el tipo de matriz (diagonal, simétrica, antisimétrica o general) y luego por cada elemento. Cada respuesta introducida por el usuario debe terminar con un punto y coma `;` o con un signo de dólar `$`.

Si *n* y *m* no son iguales, Maxima pregunta por el valor de cada elemento.

Los elementos de la matriz pueden ser cualquier tipo de expresión, que en todo caso será evaluada. `entermatrix` evalúa sus argumentos.

```
(%i1) n: 3$
(%i2) m: entermatrix (n, n)$

Is the matrix  1. Diagonal  2. Symmetric  3. Antisymmetric
   4. General
Answer 1, 2, 3 or 4 :
1$
Row 1 Column 1:
(a+b)^n$
Row 2 Column 2:
(a+b)^(n+1)$
Row 3 Column 3:
(a+b)^(n+2)$

Matrix entered.
(%i3) m;
```

```
                    [        3                    ]
                    [ (b + a)      0         0     ]
                    [                              ]
(%o3)               [                  4           ]
                    [    0      (b + a)      0     ]
                    [                              ]
                    [                           5  ]
                    [    0         0      (b + a)  ]
```

genmatrix (a, *i_2*, *j_2*, *i_1*, *j_1*) [Función]
genmatrix (a, *i_2*, *j_2*, *i_1*) [Función]
genmatrix (a, *i_2*, *j_2*) [Función]

Devuelve una matriz generada a partir de *a*, siendo `a[i_1,j_1]` el elemento superior izquierdo y `a[i_2,j_2]` el inferior derecho de la matriz. Aquí *a* se declara como una arreglo (creado por `array`, pero no por `make_array`), o un array no declarado, o una función array, o una expresión lambda de dos argumentos. (An array function is created like other functions with `:=` or `define`, but arguments are enclosed in square brackets instead of parentheses.)

Si se omite *j_1*, entonces se le asigna el valor *i_1*. Si tanto *j_1* como *i_1* se omiten, a las dos variables se le asigna el valor 1.

Si un elemento `i,j` del arreglo no está definido, se le asignará el elemento simbólico `a[i,j]`.

```
(%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*) [Función]
`gramschmidt` (*x*, *F*) [Función]

Ejecuta el algoritmo de ortogonalización de Gram-Schmidt sobre *x*, que puede ser una matriz o una lista de listas. La función `gramschmidt` no altera el valor de *x*. El producto interno por defecto empleado en `gramschmidt` es `innerproduct`, o *F*, si se ha hecho uso de esta opción.

Si *x* es una matriz, el algoritmo se aplica a las filas de *x*. Si *x* es una lista de listas, el algoritmo se aplica a las sublistas, las cuales deben tener el mismo número de miembros. En cualquier caso, el valor devuelto es una lista de listas, cuyas sublistas son ortogonales.

La función `factor` es invocada en cada paso del algoritmo para simplificar resultados intermedios. Como consecuencia, el valor retornado puede contener enteros factorizados.

El nombre `gschmit` es sinónimo de `gramschmidt`.

Es necesario cargar la función haciendo `load ("eigen")`.

Ejemplo:

Algoritmo de Gram-Schmidt utilizando el producto interno por defecto.

```
(%i1) load ("eigen")$
(%i2) x: matrix ([1, 2, 3], [9, 18, 30], [12, 48, 60]);
                             [ 1   2   3  ]
                             [            ]
(%o2)                        [ 9   18  30 ]
                             [            ]
                             [ 12  48  60 ]
(%i3) y: gramschmidt (x);
                          2      2            4     3
                         3      3   3 5      2  3  2  3
(%o3)  [[1, 2, 3], [- ---, - --, ---], [- ----, ----, 0]]
                         2 7    7   2 7       5     5
(%i4) map (innerproduct, [y[1], y[2], y[3]], [y[2], y[3], y[1]]);
(%o4)                        [0, 0, 0]
```

Algoritmo de Gram-Schmidt utilizando un producto interno especificado por el usuario.

```
(%i1) load ("eigen")$
(%i2) ip (f, g) := integrate (f * g, u, a, b);
(%o2)          ip(f, g) := integrate(f g, u, a, b)
(%i3) y : gramschmidt ([1, sin(u), cos(u)], ip), a= -%pi/2, b=%pi/2;
                                  %pi cos(u) - 2
(%o3)             [1, sin(u), --------------]
                                       %pi
(%i4) map (ip, [y[1], y[2], y[3]], [y[2], y[3], y[1]]), a= -%pi/2, b=%pi/2;
(%o4)                        [0, 0, 0]
```

`ident` (*n*) [Función]

Devuelve la matriz identidad de orden *n*.

`innerproduct` (*x*, *y*) [Función]

`inprod` (*x*, *y*) [Función]

Devuelve el producto interior o escalar de *x* por *y*, que deben ser listas de igual longitud, o ambas matrices columa o fila de igual longitud. El valor devuelto es `conjugate (x) . y`, donde `.` es el operador de multiplicación no conmutativa.

Es necesario cargar la función haciendo `load ("eigen")`.

El nombre `inprod` es sinónimo de `innerproduct`.

`invert` (*M*) [Función]

Devuelve la inversa de la matriz *M*, calculada por el método del adjunto.

La implementación actual no es eficiente para matrices de orden grande.

Cuando `detout` vale `true`, el determinante se deja fuera de la inversa a modo de factor escalar.

Los elementos de la matriz inversa no se expanden. Si *M* tiene elementos polinómicos, se puede mejorar el aspecto del resultado haciendo `expand (invert (m)), detout`.

Véase la descripción de `^^` (exponente no conmutativo) para información sobre otro método para invertir matrices.

`list_matrix_entries` (*M*) [Función]

Devuelve una lista con todos los elementos de la matriz *M*.

Ejemplo:

```
(%i1) list_matrix_entries(matrix([a,b],[c,d]));
(%o1)                     [a, b, c, d]
```

`lmxchar` [Variable opcional]

Valor por defecto: `[`

La variable `lmxchar` guarda el carácter a mostrar como delimitador izquierdo de la matriz. Véase también `rmxchar`.

Ejemplo:

```
(%i1) lmxchar: "|"$
(%i2) matrix ([a, b, c], [d, e, f], [g, h, i]);
```

```
                                  | a  b  c |
                                  |         |
          (%o2)                   | d  e  f |
                                  |         |
                                  | g  h  i |
```

matrix (*fila_1*, ..., *fila_n*) [Función]

Devuelve una matriz rectangular con las filas *fila_1*, ..., *fila_n*. Cada fila es una lista de expresiones. Todas las filas deben tener el mismo número de miembros.

Las operaciones `+` (suma), `-` (resta), `*` (multiplicación) y `/` (división), se llevan a cabo elemento a elemento cuando los operandos son dos matrices, un escalar y una matriz o una matriz con un escalar. La operación `^` (exponenciación, equivalente a `**`) se lleva cabo también elemento a elemento si los operandos son un escalr y una matriz o uma matriz y un escalar, pero no si los operandos son dos matrices.

El producto matricial se representa con el operador de multiplicación no conmutativa `.`. El correspondiente operador de exponenciación no conmutativa es `^^`. Dada la matriz `A`, `A.A` = `A^^2` y `A^^-1` es la inversa de A, si existe.

Algunas variables controlan la simplificación de expresiones que incluyan estas operaciones: `doallmxops`, `domxexpt`, `domxmxops`, `doscmxops` y `doscmxplus`.

Hay otras opciones adicionales relacionadas con matrices: `lmxchar`, `rmxchar`, `ratmx`, `listarith`, `detout`, `scalarmatrix` y `sparse`.

Hay también algunas funciones que admiten matrices como argumentos o que devuelven resultados matriciales: `eigenvalues`, `eigenvectors`, `determinant`, `charpoly`, `genmatrix`, `addcol`, `addrow`, `copymatrix`, `transpose`, `echelon` y `rank`.

Ejemplos:

- Construcción de matrices a partir 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  ]
```

- Suma elemento a elemento.

```
(%i3) x + y;
                         [ %pi + 17  %e + 3 ]
(%o3)                    [                  ]
                         [  a - 8    b + 11 ]
```

- Resta elemento a elemento.

```
(%i4) x - y;
                         [ 17 - %pi  3 - %e ]
(%o4)                    [                  ]
                         [ - a - 8   11 - b ]
```

- Multiplicación elemento a elemento.

```
(%i5) x * y;
                          [ 17 %pi  3 %e ]
(%o5)                     [              ]
                          [ - 8 a   11 b ]
```

- División elemento a elemento.

```
(%i6) x / y;
                          [ 17       - 1 ]
                          [ ---  3 %e    ]
                          [ %pi          ]
(%o6)                     [              ]
                          [   8    11    ]
                          [ - -    --    ]
                          [   a    b     ]
```

- Matriz elevada a un exponente escalar, operación elemento a elemento.

```
(%i7) x ^ 3;
                           [ 4913    27  ]
(%o7)                      [             ]
                           [ - 512  1331 ]
```

- Base escalar y exponente matricial, operación elemento a elemento.

```
(%i8) exp(y);
                           [   %pi    %e ]
                           [ %e     %e   ]
(%o8)                      [             ]
                           [    a     b  ]
                           [  %e    %e   ]
```

- Base y exponente matriciales. Esta operación no se realiza elemento a elemento.

```
(%i9) x ^ y;
                                    [ %pi  %e ]
                                    [         ]
                                    [  a   b  ]
                         [ 17   3  ]
(%o9)                    [         ]
                         [ - 8  11 ]
```

- Multiplicación matricial no conmutativa.

```
(%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   ]
```

- Exponenciación matricial no conmutativa. Una base escalar b elevada a un exponente matricial M se lleva a cabo elemento a elemento y por lo tanto `b^^m` equivale a `b^m`.

```
(%i12) x ^^ 3;
                           [  3833   1719 ]
(%o12)                     [              ]
                           [ - 4584  395  ]
(%i13) %e ^^ y;
                            [   %pi    %e ]
                            [ %e     %e   ]
(%o13)                      [              ]
                            [    a     b  ]
                            [  %e    %e   ]
```

- Una matriz elevada al exponente -1 con el operador de exponenciación no conmutativa equivale a la matriz inversa, si existe.

```
(%i14) x ^^ -1;
                           [ 11      3  ]
                           [ ---  - --- ]
                           [ 211    211 ]
(%o14)                     [             ]
                           [  8    17    ]
                           [ ---   ---   ]
                           [ 211   211   ]
(%i15) x . (x ^^ -1);
                              [ 1  0 ]
(%o15)                        [      ]
                              [ 0  1 ]
```

`matrixmap` (*f*, *M*) [Función]

Devuelve una matriz con el elemento `i,j` igual a *f*(*M*`[i,j]`).

Véanse también `map`, `fullmap`, `fullmapl` y `apply`.

`matrixp` (*expr*) [Función]

Devuelve `true` si *expr* es una matriz, en caso contrario `false`.

`matrix_element_add` [Variable opcional]

Valor por defecto: `+`

La variable `matrix_element_add` guarda el símbolo del operador a ejecutar en lugar de la suma en el producto matricial; a `matrix_element_add` se le puede asignar cualquier operador n-ario (esto es, una función que admite cualquier número de argumentos). El valor asignado puede ser el nombre de un operador encerrado entre apóstrofos, el nombre de una función o una expresión lambda.

Véanse también `matrix_element_mult` y `matrix_element_transpose`.

Ejemplo:

```
(%i1) matrix_element_add: "*"$
(%i2) matrix_element_mult: "^"$
(%i3) aa: matrix ([a, b, c], [d, e, f]);
                             [ a  b  c ]
(%o3)                        [         ]
```

```
                              [ d  e  f ]
(%i4) bb: matrix ([u, v, w], [x, y, z]);
                              [ u  v  w ]
(%o4)                         [         ]
                              [ x  y  z ]
(%i5) aa . transpose (bb);
                        [  u  v  w   x  y  z ]
                        [ a  b  c   a  b  c  ]
(%o5)                   [                    ]
                        [  u  v  w   x  y  z ]
                        [ d  e  f   d  e  f  ]
```

`matrix_element_mult` [Variable opcional]

Valor por defecto: `*`

La variable `matrix_element_mult` guarda el símbolo del operador a ejecutar en lugar de la multiplicación en el producto matricial; a `matrix_element_mult` se le puede asignar cualquier operador binario. El valor asignado puede ser el nombre de un operador encerrado entre apóstrofos, el nombre de una función o una expresión lambda.

El operador `.` puede ser una opción útil en determinados contextos.

Véanse también `matrix_element_add` y `matrix_element_transpose`.

Ejemplo:

```
(%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` [Variable opcional]

Valor por defecto: `false`

La variable `matrix_element_transpose` es una operación que se aplica a cada elemento de una matriz a la que se le calcula la transpuesta. A `matrix_element_mult` se le puede asignar cualquier operador unitario. El valor asignado puede ser el nombre de un operador encerrador entre apóstrofos, el nombre de una función o una expresión lambda.

Cuando `matrix_element_transpose` es igual a `transpose`, la función `transpose` se aplica a cada elemento. Cuando `matrix_element_transpose` es igual a `nonscalars`, la función `transpose` se aplica a todos los elementos no escalares. Si alguno de los elementos es un átomo, la opción `nonscalars` se aplica `transpose` sólo si el átomo se declara no escalar, mientras que la opción `transpose` siempre aplica `transpose`.

La opción por defecto, `false`, significa que no se aplica ninguna operación.

Véanse también `matrix_element_add` y `matrix_element_mult`.

Ejemplos:

```
(%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*) [Función]

Devuelve la traza (esto es, la suma de los elementos de la diagonal principal) de la matriz cuadrada *M*.

Para disponer de esta función es necesario cargar el paquete haciendo `load ("nchrpl")`.

`minor` (*M*, *i*, *j*) [Función]

Devuelve el menor `(i, j)` de la matriz *M*. Esto es, la propia matriz *M*, una vez extraídas la fila *i* y la columna *j*.

`ncharpoly` (*M*, *x*) [Función]

Devuelve el polinomio característico de la matriz *M* respecto de la variable *x*. Es una alternativa a la función `charpoly` de Maxima.

La función `ncharpoly` opera calculando trazas de las potencias de la matriz dada, que son iguales a las sumas de las potencias de las raíces del polinomio característico. A partir de estas cantidades se pueden calcular las funciones simétricas de las raíces, que no son otra cosa sino los coeficientes del polinomio característico. La función `charpoly` opera calculando el determinante de by `x * ident [n] - a`. La función `ncharpoly` es m'as eficiente en el caso de matrices grandes y densas.

Para disponer de esta función es necesario cargar el paquete haciendo `load ("nchrpl")`.

`newdet` (*M*) [Función]

Calcula el determinante de la matriz *M* por el algoritmo del árbol menor de Johnson-Gentleman. El resultado devuelto por `newdet` tiene formato CRE.

`permanent` (*M*) [Función]

Calcula la permanente de la matriz *M* por el algoritmo del árbol menor de Johnson-Gentleman. La permanente es como un determinante pero sin cambios de signo. El resultado devuelto por `permanent` tiene formato CRE.

Véase también `newdet`.

`rank` (*M*) [Función]

Calcula el rango de la matriz *M*. Esto es, el orden del mayor subdeterminante no singular de *M*.

La función *rango* puede retornar una respuesta errónea si no detecta que un elemento de la matriz equivalente a cero lo es.

`ratmx` [Variable opcional]

Valor por defecto: `false`

Si `ratmx` vale `false`, el determinante y la suma, resta y producto matriciales se calculan cuando las matrices se expresan en términos de sus elementos, pero no se calcula la inversión matricial en su representación general.

Si `ratmx` vale `true`, las cuatro operaciones citadas más arriba se calculan en el formato CRE y el resultado de la matriz inversa también se da en formato CRE. Esto puede hacer que se expandan los elementos de la matriz, dependiendo del valor de `ratfac`, lo que quizás no sea siempre deseable.

`row` (*M*, *i*) [Función]

Devuelve la *i*-ésima fila de la matriz *M*. El valor que devuelve tiene formato de matriz.

`rmxchar` [Variable opcional]

Valor por defecto: `]`

La variable `rmxchar` es el carácter que se dibuja al lado derecho de una matriz.

Véase también `lmxchar`.

`scalarmatrixp` [Variable opcional]

Valor por defecto: `true`

Si `scalarmatrixp` vale `true`, entonces siempre que una matriz 1 x 1 se produce como resultado del cálculo del producto no conmutativo de matrices se cambia al formato escalar.

Si `scalarmatrixp` vale `all`, entonces todas las matrices 1 x 1 se simplifican a escalares.

Si `scalarmatrixp` vale `false`, las matrices 1 x 1 no se convierten en escalares.

`setelmx (`*x*`,` *i*`,` *j*`,` *M*`)` [Función]

Asigna el valor *x* al (*i*, *j*)-ésimo elemento de la matriz *M* y devuelve la matriz actualizada.

La llamada *M* `[`*i*`,` *j*`]:` *x* hace lo mismo, pero devuelve *x* en lugar de *M*.

`similaritytransform (`*M*`)` [Función]
`simtran (`*M*`)` [Función]

La función `similaritytransform` calcula la transformada de similitud de la matriz `M`. Devuelve una lista que es la salida de la instrucción `uniteigenvectors`. Además, si la variable `nondiagonalizable` vale `false` entonces se calculan dos matrices globales `leftmatrix` y `rightmatrix`. Estas matrices tienen la propiedad de que `leftmatrix .` *M* `. rightmatrix` es una matriz diagonal con los valores propios de *M* en su diagonal. Si `nondiagonalizable` vale `true` entonces no se calculan estas matrices.

Si la variable `hermitianmatrix` vale `true` entonces `leftmatrix` es el conjugado complejo de la transpuesta de `rightmatrix`. En otro caso `leftmatrix` es la inversa de `rightmatrix`.

Las columnas de la matriz `rightmatrix` son los vectores propios de *M*. Las otras variables (véanse `eigenvalues` y `eigenvectors`) tienen el mismo efecto, puesto que `similaritytransform` llama a las otras funciones del paquete para poder formar `rightmatrix`.

Estas funciones se cargan con `load ("eigen")`.

El nombre `simtran` es sinónimo de `similaritytransform`.

`sparse` [Variable opcional]

Valor por defecto: `false`

Si `sparse` vale `true` y si `ratmx` vale `true`, entonces `determinant` utilizará rutinas especiales para calcular determinantes dispersos.

`submatrix (`*i_1*`, ...,` *i_m*`,` *M*`,` *j_1*`, ...,` *j_n*`)` [Función]
`submatrix (`*i_1*`, ...,` *i_m*`,` *M*`)` [Función]
`submatrix (`*M*`,` *j_1*`, ...,` *j_n*`)` [Función]

Devuelve una nueva matriz formada a partir de la matriz *M* pero cuyas filas *i_1*, ..., *i_m* y columnas *j_1*, ..., *j_n* han sido eliminadas.

`transpose (`*M*`)` [Función]

Calcula la transpuesta de *M*.

Si *M* es una matriz, el valor devuelto es otra matriz *N* tal que `N[i,j] = M[j,i]`.

Si M es una lista, el valor devuelto es una matriz N de `length (m)` filas y 1 columna, tal que `N[i,1] = M[i]`.

En caso de no ser M ni matriz ni lista, se devuelve la expresión nominal `'transpose (M)`.

`triangularize (M)` [Función]

Devuelve la forma triangular superior de la matriz `M`, obtenida por eliminación gaussiana. El resultado es el mismo que el devuelto por `echelon`, con la salvedad de que el primer elemento no nulo de cada fila no se normaliza a 1.

Las funciones `lu_factor` y `cholesky` también triangularizan matrices.

```
(%i1) M: matrix ([3, 7, aa, bb], [-1, 8, 5, 2], [9, 2, 11, 4]);
                        [  3   7  aa  bb ]
                        [                ]
(%o1)                   [ - 1  8  5   2  ]
                        [                ]
                        [  9   2  11  4  ]
(%i2) triangularize (M);
             [ - 1   8         5            2      ]
             [                                     ]
(%o2)        [  0   - 74     - 56         - 22     ]
             [                                     ]
             [  0    0    626 - 74 aa  238 - 74 bb ]
```

`uniteigenvectors (M)` [Función]
`ueivects (M)` [Función]

Calcula los vectores propios unitarios de la matriz M. El valor que devuelve es una lista de listas, la primera de las cuales es la salida de la función `eigenvalues` y el resto de sublistas son los vectores propios unitarios de la matriz correspondiente a esos valores propios, respectivamente.

Las variables citadas en la descripción de la función `eigenvectors` tienen los mismos efectos en `uniteigenvectors`.

Si `knowneigvects` vale `true`, el paquete `eigen` da por supuesto que el usuario conoce los vectores propios de la matriz y que están guardados en la variable global `listeigvects`, en tal caso el contenido de `listeigvects` debe ser una lista de estructura similar a la que devuelve la función `eigenvectors`.

Si `knowneigvects` vale `true` y la lista de vectores propios está en la variable `listeigvects`, el valor de la variable `nondiagonalizable` puede que no sea el correcto. Si tal es el caso, debe asignarsele el valor correcto.

Para utilizar esta fucnión es necesario cargarla haciendo `load ("eigen")`.

El nombre `ueivects` es sinónimo de `uniteigenvectors`.

`unitvector (x)` [Función]
`uvect (x)` [Función]

Devuelve $x/norm(x)$, esto es, el vector unitario de igual dirección y sentido que x.

`load ("eigen")` loads this function.

Para utilizar esta fucnión es necesario cargarla haciendo `load ("eigen")`.

El nombre `uvect` es sinónimo de `unitvector`.

`vectorpotential` (*givencurl*) [Función]
Devuelve el vector potencial de un vector rotacional en el sistema de coordenadas actual. `potentialzeroloc` tiene un rol similar al de `potential`, pero el orden del miembro izquierdo de las ecuaciones debe ser una permutación cíclica de las coordenadas.

`vectorsimp` (*expr*) [Función]
Realiza simplificaciones y expansiones de acuerdo con los valores de las siguientes variables globales:

`expandall`, `expanddot`, `expanddotplus`, `expandcross`, `expandcrossplus`, `expandcrosscross`, `expandgrad`, `expandgradplus`, `expandgradprod`, `expanddiv`, `expanddivplus`, `expanddivprod`, `expandcurl`, `expandcurlplus`, `expandcurlcurl`, `expandlaplacian`, `expandlaplacianplus` y `expandlaplacianprod`.

Todas estas variables tienen por defecto el valor `false`. El sufijo `plus` se refiere al uso de la suma o la distributividad. El sufijo `prod` se refiere a la expansión de operadores que realizan cualquier tipo de producto.

`expandcrosscross`
Simplifica $p\ (q\ r)$ en $(p.r) * q - (p.q) * r$.

`expandcurlcurl`
Simplifica *curlcurlp* en $graddivp + divgradp$.

`expandlaplaciantodivgrad`
Simplifica *laplacianp* en *divgradp*.

`expandcross`
Activa `expandcrossplus` y `expandcrosscross`.

`expandplus`
Activa `expanddotplus`, `expandcrossplus`, `expandgradplus`, `expanddivplus`, `expandcurlplus` y `expandlaplacianplus`.

`expandprod`
Activa `expandgradprod`, `expanddivprod` y `expandlaplacianprod`.

Estas variables están declaradas como `evflag`.

`zeromatrix` (*m*, *n*) [Función]
Devuelve una matriz rectangular m por n con todos sus elementos iguales a cero.

`[` [Símbolo especial]
`[` [Símbolo especial]
Los símbolos `[` y `]` marcan el comienzo y final, respectivamente, de una lista.

Los símbolos `[` y `]` también se utilizan para indicar los subíndices de los elementos de una lista, arreglo o función arreglo.

Ejemplos:

```
(%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
```

24 Afines

24.1 Funciones y variables para Afines

`fast_linsolve` ([*expr_1*, ..., *expr_m*], [*x_1*, ..., *x_n*]) [Función]

Resuelve las ecuaciones lineales simultáneas *expr_1*, ..., *expr_m* para las variables *x_1*, ..., *x_n*. Cada *expr_i* puede ser una ecuación o una expresión general; en caso de tratarse de una expresión general, será tratada como una ecuación de la forma *expr_i* `= 0`.

El valor que devuelve es una lista de ecuaciones de la forma `[`*x_1* `= a_1, ..., x_n =` *a_n*`]` donde todas las *a_1*, ..., *a_n* están exentas de *x_1*, ..., *x_n*.

La función `fast_linsolve` es más rápida que `linsolve` para sistemas de ecuaciones con coeficientes dispersos.

Antes de utilizar esta función ejecútese `load("affine")`.

`grobner_basis` ([*expr_1*, ..., *expr_m*]) [Función]

Devuelve una base de Groebner para las ecuaciones *expr_1*, ..., *expr_m*. La función `polysimp` puede ser entonces utilizada para simplificar otras funciones relativas a las ecuaciones.

```
grobner_basis ([3*x^2+1, y*x])$

polysimp (y^2*x + x^3*9 + 2) ==> -3*x + 2
```

`polysimp(f)` alcanza 0 si y sólo si *f* está en el ideal generado por *expr_1*, ..., *expr_m*, es decir, si y sólo si *f* es una combinación polinómica de los elementos de *expr_1*, ..., *expr_m*.

Antes de utilizar esta función ejecútese `load("affine")`.

`set_up_dot_simplifications` (*eqns*, *check_through_degree*) [Función]

`set_up_dot_simplifications` (*eqns*) [Función]

Las *eqns* son ecuaciones polinómicas de variables no conmutativas. El valor de `current_variables` es la lista de variables utilizadas para el cálculo de los grados. Las ecuaciones deben ser homogéneas, al objeto de completar el procedimiento.

El grado es el devuelto por `nc_degree`. Éste a su vez depende de los pesos de las variables individuales.

Antes de utilizar esta función ejecútese `load("affine")`.

`declare_weights` (*x_1*, *w_1*, ..., *x_n*, *w_n*) [Función]

Asigna los pesos *w_1*, ..., *w_n* a *x_1*, ..., *x_n*, respectivamente. Estos pesos son los utilizados en el cálculo de `nc_degree`.

Antes de utilizar esta función ejecútese `load("affine")`.

`nc_degree` (*p*) [Función]

Devuelve el grado de un polinomio no conmutativo *p*. Véase `declare_weights`.

Antes de utilizar esta función ejecútese `load("affine")`.

`dotsimp` (*f*) [Función]

Devuelve 0 si y sólo si *f* está en el ideal generado por las ecuaciones, esto es, si y sólo si *f* es una combinación lineal de los elementos de las ecuaciones.

Antes de utilizar esta función ejecútese `load("affine")`.

`fast_central_elements` ([*x_1*, ..., *x_n*], *n*) [Función]

Si se ha ejecutado `set_up_dot_simplifications` con antelación, obtiene los polinomios centrales de grado *n* de variables *x_1*, ..., *x_n*.

Por ejemplo:

```
set_up_dot_simplifications ([y.x + x.y], 3);
fast_central_elements ([x, y], 2);
[y.y, x.x];
```

Antes de utilizar esta función ejecútese `load("affine")`.

`check_overlaps` (*n*, *add_to_simps*) [Función]

Revisa la superposición hasta el grado *n*, asegurándose de que el usuario tiene suficientes reglas de simplificación en cada grado para que `dotsimp` trabaje correctamente. Este proceso puede acelerarse si se conoce de antemano cuál es la dimensión del espacio de monomios. Si éste es de dimensión global finita, entonces debería usarse `hilbert`. Si no se conoce la dimensiones de los monomios, no se debería especificar una `rank_function`. Un tercer argumento opcional es `reset`.

Antes de utilizar esta función ejecútese `load("affine")`.

`mono` ([*x_1*, ..., *x_n*], *n*) [Función]

Devuelve la lista de monomios independientes.

Antes de utilizar esta función ejecútese `load("affine")`.

`monomial_dimensions` (*n*) [Función]

Calcula el desarrollo de Hilbert de grado *n* para el algebra actual.

Antes de utilizar esta función ejecútese `load("affine")`.

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

Hace una lista de los coeficientes de los polinomios no conmutativos *p_1*, ..., *p_n* de los monomios no conmutativos *m_1*, ..., *m_n*. Los coeficientes deben escalares. Hágase uso de `list_nc_monomials` para construir la lista de monomios.

Antes de utilizar esta función ejecútese `load("affine")`.

`list_nc_monomials` ([*p_1*, ..., *p_n*]) [Función]

`list_nc_monomials` (*p*) [Función]

Devuelve una lista de los monomios no conmutativos que aparecen en el polinomio *p* o una lista de polinomios en *p_1*, ..., *p_n*.

Antes de utilizar esta función ejecútese `load("affine")`.

`all_dotsimp_denoms` [Variable]

Valor por defecto: `false`

Cuando `all_dotsimp_denoms` es una lista, los denominadores encontrados por `dotsimp` son añadidos a la lista. La variable `all_dotsimp_denoms` puede inicializarse como una lista vacía `[]` antes de llamar a `dotsimp`.

Por defecto, `dotsimp` no recolecta los denominadores.

25 itensor

25.1 Introducción a itensor

Maxima implementa dos tipos diferentes de manipulación simbólica de tensores: la manipulación de componentes (paquete `ctensor`) y la manipulación indexada (paquete `itensor`).

Véase más abajo la nota sobre 'notación tensorial'.

La manipulación de componentes significa que los objetos geométricos tensoriales se representan como arreglos (arrays) o matrices. Operaciones tensoriales como la contracción o la diferenciación covariante se llevan a cabo sumando índices mudos con la sentencia `do`. Esto es, se realizan operaciones directamente con las componentes del tensor almacenadas en un arreglo o matriz.

La manipulación indexada de tensores se lleva a cabo mediante la representación de los tensores como funciones de sus índices covariantes, contravariantes y de derivadas. Operaciones tensoriales como la contracción o la diferenciación covariante se realizan manipulando directamente los índices, en lugar de sus componentes asociadas.

Estas dos técnicas para el tratamiento de los procesos diferenciales, algebraicos y analíticos en el contexto de la geometría riemanniana tienen varias ventajas y desventajas que surgen según la naturaleza y dificultad del problema que está abordando el usuario. Sin embargo, se deben tener presentes las siguientes características de estas dos técnicas:

La representación de los tensores y sus operaciones en términos de sus componentes facilita el uso de paquete `ctensor`. La especificación de la métrica y el cálculo de los tensores inducidos e invariantes es inmediato. Aunque toda la potencia de simplificación de Maxima se encuentra siempre a mano, una métrica compleja con dependencias funcionales y de coordenadas intrincada, puede conducir a expresiones de gran tamaño en las que la estructura interna quede oculta. Además, muchos cálculos requieren de expresiones intermedias que pueden provocar la detención súbita de los programas antes de que se termine el cálculo. Con la experiencia, el usuario podrá evitar muchas de estas dificultades.

Devido a la forma en que los tensores y sus operaciones se representan en términos de operaciones simbólicas con sus índices, expresiones que serían intratables en su representación por componentes pueden en ocasiones simplificarse notablemente utilizando las rutinas especiales para objetos simétricos del paquete `itensor`. De esta manera, la estructura de expresiones grandes puede hacerse más transparente. Por otro lado, debido a la forma especial de la representación indexada de tensores en `itensor`, en algunos casos el usuario encontrará dificultades con la especificación de la métrica o la definición de funciones.

El paquete `itensor` puede derivar respecto de una variable indexada, lo que permite utilizar el paquete cuando se haga uso del formalismo de lagrangiano y hamiltoniano. Puesto que es posible derivar un campo lagrangiano respecto de una variable de campo indexada, se puede hacer uso de Maxima para derivar las ecuaciones de Euler-Lagrange correspondientes en forma indexada. Estas ecuaciones pueden traducirse a componentes tensoriales (`ctensor`) con la función `ic_convert`, lo que permite resolver las ecuaciones de campo en cualquier sistema de coordenadas, o obtener las ecuaciones de movimiento en forma hamiltoniana. Véanse dos ejemplos en `einhil.dem` y `bradic.dem`; el primero utiliza la acción de Einstein-Hilbert para derivar el campo tensorial de Einstein en el caso homogéneo e isotrópico (ecuaciones de Friedmann), así como en el caso esferosimétrico estático (solución

de Schwarzschild); el segundo demuestra cómo calcular las ecuaciones de Friedmann a partir de la acción de la teoría de la gravedad de Brans-Dicke, y también muestra cómo derivar el hamiltoniano asociado con la teoría del campo escalar.

25.1.1 Notación tensorial

Hasta ahora, el paquete `itensor` de Maxima utilizaba una notación que algunas veces llevaba a una ordenación incorrecta de los índices. Por ejemplo:

```
(%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
```

Este resultado no es correcto a menos que `a` sea un tensor simétrico. La razón por la que esto ocurre es que aunque `itensor` mantenga correctamente el orden dentro del conjunto de índices covariantes y contravariantes, una vez un índice sea aumentado o disminuido, su posición relativa al otro conjunto de índices se pierde.

Para evitar este problema, se ha desarrollado una notación totalmente compatible con la anterior.En esta notación, los índices contravariantes se insertan en las posiciones correctas en la lista de índices covariantes, pero precedidos del signo negativo.

En esta notación, el ejemplo anterior da el resultado correcto:

```
(%i5) ishow(g([-j,-k],[])*g([-i,-l],[])*a([i,j],[]))$
                                  i l       j k
(%t5)                            g    a    g
                                       i j
(%i6) ishow(contract(%))$
                                       l k
(%t6)                                  a
```

El único código que hace uso de esta notación es la función `lc2kdt`.

Devido a que este código es nuevo, puede contener errores.

25.1.2 Manipulación indexada de tensores

El paquete `itensor` se carga haciendo `load("itensor")`. Para acceder a las demos se hará `demo(tensor)`.

En el paquete `itensor` un tensor se representa como un objeto indexado, esto es, como una función de tres grupos de índices: los covariantes, los contravariantes y los de derivadas. Los índices covariantes se especifican mediante una lista que será el primer argumento del objeto indexado, siendo los índices contravariantes otra lista que será el segundo argumento del mismo objeto indexado. Si al objeto indexado le falta cualquiera de estos grupos de índices, entonces se le asignará al argumento correspondiente la lista vacía `[]`. Así, `g([a,b],[c])` representa un objeto indexado llamado `g`, el cual tiene dos índices covariantes `(a,b)`, un índice contravariante (`c`) y no tiene índices de derivadas.

Los índices de derivadas, si están presentes, se añaden como argumentos adicionales a la función simbólica que representa al tensor. Se pueden especificar explícitamente por el usuario o pueden crearse durante el proceso de diferenciación respecto de alguna coordenada. Puesto que la diferenciación ordinaria es conmutativa, los índices de derivadas se ordenan alfanuméricamente, a menos que la variable `iframe_flag` valga `true`, indicando que se está utilizando una métrica del sistema de referencia. Esta ordenación canónica hace posible que Maxima reconozca, por ejemplo, que `t([a],[b],i,j)` es lo mismo que `t([a],[b],j,i)`. La diferenciación de un objeto indexado con respecto de alguna coordenada cuyo índice no aparece como argumento de dicho objeto indexado, dará como resultado cero. Esto se debe a que Maxima no sabe si el tensor representado por el objeto indexado depende implícitamente de la coordenada correspondiente. Modificando la función `diff` de Maxima en `itensor`, se da por hecho que todos los objetos indexados dependen de cualquier variable de diferenciación, a menos que se indique lo contrario. Esto hace posible que la convención sobre la sumación se extienda a los índices de derivadas. El paquete `itensor` trata a los índices de derivadas como covariantes.

Las siguientes funciones forman parte del paquete `itensor` para la manipulación indexada de vectores. En lo que respecta a las rutinas de simplificación, no se considera en general que los objetos indexados tengan propiedades simétricas. Esto puede cambiarse reasignando a la variable `allsym[false]` el valor `true`, con lo cual los objetos indexados se considerarán simétricos tanto respecto de sus índices covariantes como contravariantes.

En general, el paquete `itensor` trata a los tensores como objetos opacos. Las ecuaciones tensoriales se manipulan en base a reglas algebraicas, como la simetría y la contracción. Además, en el paquete `itensor` hay funciones para la diferenciación covariante, la curvatura y la torsión. Los cálculos se pueden realizar respecto de una métrica del sistema de referencia móvil, dependiendo de las asignaciones dadas a la variable `iframe_flag`.

La siguiente sesión de ejemplo demuestra cómo cargar el paquete `itensor`, especificar el nombre de la métrica y realizar algunos cálculos sencillos.

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) imetric(g);
(%o2)                                done
(%i3) components(g([i,j],[]),p([i,j],[])*e([],[]))$
(%i4) ishow(g([k,l],[]))$
(%t4)                               e p
                                       k l
(%i5) ishow(diff(v([i],[]),t))$
(%t5)                                  0
(%i6) depends(v,t);
(%o6)                               [v(t)]
(%i7) ishow(diff(v([i],[]),t))$
                                    d
(%t7)                               -- (v )
                                    dt   i
(%i8) ishow(idiff(v([i],[]),j))$
(%t8)                                v
                                      i,j
```

```
(%i9) ishow(extdiff(v([i],[]),j))$
(%t9)                              v    - v
                                    j,i    i,j
                                   -----------
                                        2
(%i10) ishow(liediff(v,w([i],[])))$
                                  %3          %3
(%t10)                           v   w     + v   w
                                  i,%3    ,i  %3
(%i11) ishow(covdiff(v([i],[]),j))$
                                                %4
(%t11)                           v    - v   ichr2
                                  i,j    %4      i j
(%i12) ishow(ev(%,ichr2))$
                 %4 %5
(%t12) v    - (g      v   (e p       + e   p     - e p       - e    p
        i,j            %4     j %5,i    ,i  j %5      i j,%5    ,%5  i j

                                             + e p       + e   p    ))/2
                                                  i %5,j    ,j  i %5
(%i13) iframe_flag:true;
(%o13)                                    true
(%i14) ishow(covdiff(v([i],[]),j))$
                                               %6
(%t14)                           v    - v   icc2
                                  i,j    %6     i j
(%i15) ishow(ev(%,icc2))$
                                               %6
(%t15)                           v    - v   ifc2
                                  i,j    %6     i j
(%i16) ishow(radcan(ev(%,ifc2,ifc1)))$
              %6 %7                       %6 %7
(%t16) - (ifg      v   ifb       + ifg      v   ifb       - 2 v
                   %6    j %7 i             %6    i j %7      i,j

                                               %6 %7
                                          - ifg      v   ifb      )/2
                                                      %6    %7 i j
(%i17) ishow(canform(s([i,j],[])-s([j,i])))$
(%t17)                               s    - s
                                      i j    j i
(%i18) decsym(s,2,0,[sym(all)],[]);
(%o18)                                    done
(%i19) ishow(canform(s([i,j],[])-s([j,i])))$
(%t19)                                     0
(%i20) ishow(canform(a([i,j],[])+a([j,i])))$
(%t20)                               a    + a
```

```
                                      j i    i j
(%i21) decsym(a,2,0,[anti(all)],[]);
(%o21)                                done
(%i22) ishow(canform(a([i,j],[])+a([j,i])))$
(%t22)                                 0
```

25.2 Funciones y variables para itensor

25.2.1 Trabajando con objetos indexados

`dispcon` (*tensor_1*, *tensor_2*, ...) [Función]
`dispcon` (*all*) [Función]

Muestra las propiedades contractivas de sus argumentos tal como fueron asignadas por `defcon`. La llamada `dispcon (all)` muestra todas propiedades contractivas que fueron definidas.

`entertensor` (*nombre*) [Función]

Permite crear un objeto indexado llamado *nombre*, con cualquier número de índices tensoriales y de derivadas. Se admiten desde un único índice hasta una lista de índices. Véase el ejemplo en la descripción de `covdiff`.

`changename` (*anterior*, *nuevo*, *expr*) [Función]

Cambia el nombre de todos los objetos indexados llamados *anterior* a *new* en *expr*. El argumento *anterior* puede ser un símbolo o una lista de la forma `[`*nombre*`,` *m*`,` *n*`]`, en cuyo caso sólo los objetos indexados de llamados *nombre* con *m* índices covariantes y *n* contravariantes se renombrarán como *nuevo*.

`listoftens` [Función]

Hace un listado de todos los tensores y sus índices en una expresión tensorial. Por ejemplo,

```
(%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*) [Función]

Muestra *expr* con todos los objetos indexados que contiene, junto con los correspondientes índices covariantes (como subíndices) y contravariantes (como superíndices). Los índices de derivadas se muestran como subíndices, separados de los covariantes por una coma; véanse los múltiples ejemplos de este documento.

`indices` (*expr*) [Función]

Devuelve una lista con dos elementos. El primer elemento es una lista con los índices libres, aquellos que aparecen una sola vez. El segundo elemento es una lista con los índices mudos en *expr*, aquellos que aparecen exactamente dos veces. Por ejemplo,

```
(%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]]
```

Un producto tensorial que contenga el mismo índice más de dos veces es sintácticamente incorrecto. La función `indices` intenta tratar estas expresiones de una forma razonable; sin embargo, cuando se la obliga a manipular una expresión incorrecta puede tener un comportamiento imprevisto.

`rename` (*expr*) [Función]

`rename` (*expr*, *count*) [Función]

Devuelve una expresión equivalente a *expr* pero con los índices mudos de cada término elegidos del conjunto `[%1, %2,...]` si el segundo argumento opcional se omite. En otro caso, los índices mudos son indexados empezando con el valor *count*. Cada índice mudo en un producto será diferente. En el caso de las sumas, la función `rename` operará sobre cada término de la suma reinicializando el contador con cada término. De esta manera `rename` puede servir como simplificador tensorial. Además, los índices se ordenarán alfanuméricamente, si la variable `allsym` vale `true`, respecto de los índices covariantes y contravariantes dependiendo del valor de `flipflag`. Si `flipflag` vale `false`, entonces los índices se renombrarán de acuerdo con el orden de los índices contravariantes. Si `flipflag` vale `true`, entonces los índices se renombrarán de acuerdo con el orden de los índices covariantes. Suele acontecer que el efecto combinado de los dos cambios de nombre reduzcan la expresión más de lo que que pueda reducir cualquiera de ellas por separado.

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) allsym:true;
(%o2)                                  true
(%i3) g([],[%4,%5])*g([],[%6,%7])*ichr2([%1,%4],[%3])*
ichr2([%2,%3],[u])*ichr2([%5,%6],[%1])*ichr2([%7,r],[%2])-
g([],[%4,%5])*g([],[%6,%7])*ichr2([%1,%2],[u])*
ichr2([%3,%5],[%1])*ichr2([%4,%6],[%3])*ichr2([%7,r],[%2]),noeval$
(%i4) expr:ishow(%)$

       %4 %5  %6 %7      %3         u          %1         %2
(%t4) g      g      ichr2      ichr2      ichr2      ichr2
```

```
                                %1 %4      %2 %3      %5 %6      %7 r

          %4 %5  %6 %7      u          %1         %3         %2
      - g      g      ichr2      ichr2      ichr2      ichr2
                                %1 %2      %3 %5      %4 %6      %7 r
(%i5) flipflag:true;
(%o5)                                    true
(%i6) ishow(rename(expr))$
        %2 %5  %6 %7      %4         u          %1         %3
(%t6) g      g      ichr2      ichr2      ichr2      ichr2
                                %1 %2      %3 %4      %5 %6      %7 r

          %4 %5  %6 %7      u          %1         %3         %2
      - g      g      ichr2      ichr2      ichr2      ichr2
                                %1 %2      %3 %4      %5 %6      %7 r
(%i7) flipflag:false;
(%o7)                                    false
(%i8) rename(%th(2));
(%o8)                                      0
(%i9) ishow(rename(expr))$
        %1 %2  %3 %4      %5         %6         %7         u
(%t9) g      g      ichr2      ichr2      ichr2     ichr2
                                %1 %6      %2 %3      %4 r       %5 %7

          %1 %2  %3 %4      %6         %5         %7         u
      - g      g      ichr2      ichr2      ichr2     ichr2
                                %1 %3      %2 %6      %4 r       %5 %7
```

`show` (*expr*) [Función]

Muestra `expr` con sus objetos indexados que tengan índices covariantes como subíndices y los contravariantes como superíndices. Los índices derivados se muestran como subíndices, separados por una coma de los covariantes.

`flipflag` [Variable opcional]

Valor por defecto: `false`

Si vale `false` los índices se renombrarán de acuerdo con el orden de los índices covariantes, si `true` se renombrarán de acuerdo con el orden de los índices covariantes.

Si `flipflag` vale `false`, entonces `rename` construye una lista con los índices contravariantes según van apareciendo de izquierda a derecha; si vale `true`, entonces va formando la lista con los covariantes. Al primer índice mudo se le da el nombre `%1`, al siguiente `%2`, etc. Finalmente se hace la ordenación. Véase el ejemplo en la descripción de la función `rename`.

`defcon` (*tensor_1*) [Función]

`defcon` (*tensor_1*, *tensor_2*, *tensor_3*) [Función]

Le asigna a gives *tensor_1* la propiedad de que la contracción de un producto de *tensor_1* por *tensor_2* da como resultado un *tensor_3* con los índices apropiados.

Si sólo se aporta un argumento, *tensor_1*, entonces la contracción del producto de *tensor_1* por cualquier otro objeto indexado que tenga los índices apropiados, por ejemplo `my_tensor`, dará como resultado un objeto indexado con ese nombre, `my_tensor`, y con un nuevo conjunto de índices que reflejen las contracciones realizadas. Por ejemplo, si `imetric:g`, entonces `defcon(g)` implementará el aumento o disminución de los índices a través de la contracción con el tensor métrico. Se puede dar más de un `defcon` para el mismo objeto indexado, aplicándose el último. La variable `contractions` es una lista con aquellos objetos indexados a los que se le han dado propiedades de contracción con `defcon`.

`remcon` (*tensor_1*, ..., *tensor_n*) [Función]
`remcon` (*all*) [Función]

Borra todas las propiedades de contracción de *tensor_1*, ..., *tensor_n*). La llamada `remcon(all)` borra todas las propiedades de contracción de todos los objetos indexados.

`contract` (*expr*) [Función]

Lleva a cabo las contracciones tensoriales en *expr*, la cual puede ser cualquier combinación de sumas y productos. Esta función utiliza la información dada a la función `defcon`. Para obtener mejores resultados, `expr` debería estar completamente expandida. La función `ratexpand` es la forma más rápida de expandir productos y potencias de sumas si no hay variables en los denominadores de los términos.

`indexed_tensor` (*tensor*) [Función]

Debe ejecutarse antes de asignarle componentes a un *tensor* para el que ya existe un valor, como `ichr1`, `ichr2` o `icurvature`. Véase el ejemplo de la descripción de `icurvature`.

`components` (*tensor*, *expr*) [Función]

Permite asignar un valor indexado a la expresión *expr* dando los valores de las componentes de *tensor*. El tensor debe ser de la forma `t([...],[...])`, donde cualquiera de las listas puede estar vacía. La expresión *expr* puede ser cualquier objeto indexado que tenga otros objetos con los mismos índices libres que *tensor*. Cuando se utiliza para asignar valores al tensor métrico en el que las componentes contengan índices mudos, se debe tener cuidado en no generar índices mudos múltiples. Se pueden borrar estas asignaciones con la función `remcomps`.

Es importante tener en cuenta que `components` controla la valencia del tensor, no el orden de los índices. Así, asignando componentes de la forma `x([i,-j],[])`, `x([-j,i],[])` o `x([i],[j])` todos ellos producen el mismo resultado, la asignación de componentes a un tensor de nombre `x` con valencia `(1,1)`.

Las componentes se pueden asignar a una expresión indexada de cuatro maneras, dos de las cuales implican el uso de la instrucción `components`:

1) Como una expresión indexada. Por ejemplo:

```
(%i2) components(g([],[i,j]),e([],[i])*p([],[j]))$
(%i3) ishow(g([],[i,j]))$
                                          i  j
```

```
(%t3)                                  e  p
```

2) Como una matriz:

```
(%i5) lg:-ident(4)$lg[1,1]:1$lg;
                              [ 1   0    0    0  ]
                              [                   ]
                              [ 0  - 1   0    0  ]
(%o5)                         [                   ]
                              [ 0   0   - 1   0  ]
                              [                   ]
                              [ 0   0    0   - 1 ]

(%i6) components(g([i,j],[]),lg);
(%o6)                                  done
(%i7) ishow(g([i,j],[]))$
(%t7)                                  g
                                        i j
(%i8) g([1,1],[]);
(%o8)                                   1
(%i9) g([4,4],[]);
(%o9)                                  - 1
```

3) Como una función. Se puede utilizar una función de Maxima para especificar las componentes de un tensor en base a sus índices. Por ejemplo, el código siguiente asigna `kdelta` a `h` si `h` tiene el mismo número de índices covariantes y contravariantes y no tiene índices de derivadas, asignándole `g` en otro caso:

```
(%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) Utilizando los patrones de Maxima, en particular las funciones `defrule` y `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*) [Función]

Borra todos los valores de *tensor* que han sido asignados con la función `components`.

showcomps (*tensor*) [Función]

Muestra las componentes de un tensor definidas con la instrucción `components`. Por ejemplo:

```
(%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
```

La función `showcomps` también puede mostrar las componentes de tensores de rango mayor de 2.

`idummy` () [Función]

Incrementa `icounter` y devuelve un índice de la forma `%n` siendo `n` un entero positivo. Esto garantiza que índices mudos que sean necesarios para formar expresiones no entren en conflico con índices que ya están en uso. Véase el ejemplo de la descripción de `indices`.

`idummyx` [Variable opcional]

Valor por defecto: `%`

Es el prefijo de los índices mudos. Véase `indices`.

`icounter` [Variable opcional]

Valor por defecto: `1`

Determina el sufijo numérico a ser utilizado en la generación del siguiente índice mudo. El prefijo se determina con la opción `idummy` (por defecto: %).

`kdelta` (*L1*, *L2*) [Función]

Es la función delta generalizada de Kronecker definida en el paquete `itensor` siendo *L1* la lista de índices covariantes y *L2* la lista de índices contravariantes. La función `kdelta([i],[j])` devuelve el valor de la delta ordinaria de Kronecker. La instrucción `ev(`*expr*`,kdelta)` provoca la evaluación de una expresión que contenga `kdelta([],[])`.

En un abuso de la notación, `itensor` también permite a `kdelta` tener 2 índices covariantes y ninguno contravariante, o 2 contravariantes y ninguno covariante. Esto es una funcionalidad del paquete, loque no implica que `kdelta([i,j],[])` sea un objeto tensorial de pleno derecho.

`kdels` (*L1*, *L2*) [Función]

Función delta de Kronecker simetrizada, utilizada en algunos cálculos. Por ejemplo:

```
(%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*) [Función]

Es el tensor de permutación de Levi-Civita, el cual devuelve 1 si la lista *L* con una permutación par de enteros, -1 si es en una permutación impar y 0 si algunos de los índices de *L* están repetidos.

`lc2kdt` (*expr*) [Función]

Simplifica expresiones que contengan el símbolo de Levi-Civita, convirtiéndolas en expresiones con la delta de Kronecker siempre que sea posible. La diferencia principal entre esta función y la simple evaluación del símbolo de Levi-Civita consiste en que de esta última forma se obtienen expresiones de Kronecker con índices numéricos, lo que impide simplificaciones ulteriores. La función `lc2kdt` evita este problema, dando resultados con son más fáciles de simplificar con `rename` o `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
```

La función `lc2kdt` en ocasiones hace uso del tensor métrico. Si el tensor métrico no fue previamente definido con `imetric`, se obtiene un mensaje de error.

```
(%i7) expr:ishow('levi_civita([],[i,j])
                *'levi_civita([],[k,l])*a([j,k],[]))$
                                    i j            k l
(%t7)                 levi_civita    levi_civita    a
                                                       j k
(%i8) ishow(lc2kdt(expr))$
Maxima encountered a Lisp error:

 Error in $IMETRIC [or a callee]:
 $IMETRIC [or a callee] requires less than two arguments.

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
(%i9) imetric(g);
(%o9)                                done
(%i10) ishow(lc2kdt(expr))$
         %3 i       k   %4 j       l     %3 i       l   %4 j
(%t10) (g     kdelta   g     kdelta   - g     kdelta   g
                    %3             %4               %3
              k
        kdelta  ) a
              %4   j k
(%i11) ishow(contract(expand(%)))$
                                   l i    l i  j
(%t11)                            a    - g    a
                                                j
```

`lc_l` [Función]

Regla de simplificación utilizada en expresiones que contienen el símbolo de `levi_civita` sin evaluar. Junto con `lc_u`, puede utilizarse para simplificar muchas expresiones de forma más eficiente que la evaluación de `levi_civita`. Por ejemplo:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2)  el1:ishow('levi_civita([i,j,k],[])*a([],[i])*a([],[j]))$
                              i  j
(%t2)                        a  a  levi_civita
                                                 i j k
(%i3) el2:ishow('levi_civita([],[i,j,k])*a([i])*a([j]))$
```

```
                                              i j k
(%t3)                         levi_civita      a  a
                                                i  j
(%i4) canform(contract(expand(applyb1(el1,lc_l,lc_u))));
(%t4)                                         0
(%i5) canform(contract(expand(applyb1(el2,lc_l,lc_u))));
(%t5)                                         0
```

`lc_u` [Función]

Regla de simplificación utilizada en expresiones que contienen el símbolo de `levi_civita` sin evaluar. Junto con `lc_l`, puede utilizarse para simplificar muchas expresiones de forma más eficiente que la evaluación de `levi_civita`. Véase `lc_l`.

`canten` (*expr*) [Función]

Simplifica *expr* renombrando (véase `rename`) y permutando índices mudos. La función `rename` se restringe a sumas de productos de tensores en los cuales no hay derivadas, por lo que está limitada y sólo debería utilizarse si `canform` no es capaz de de llevar a cabo la simplificación requerida.

La función `canten` devuelve un resultado matemáticamente correcto sólo si su argumento es una expresión completamente simétrica respecto de sus índices. Por esta razón, `canten` devuelve un error si `allsym` no vale `true`.

`concan` (*expr*) [Función]

Similar a `canten` pero también realiza la contracción de los índices.

25.2.2 Simetrías de tensores

`allsym` [Variable opcional]

Valor por defecto: `false`

Si vale `true` entonces todos los objetos indexados se consideran simétricos respecto de todos sus índices covariantes y contravariantes. Si vale `false` entonces no se tienen en cuenta ningún tipo de simetría para estos índices. Los índices de derivadas se consideran siempre simétricos, a menos que la variable `iframe_flag` valga `true`.

`decsym` (*tensor*, *m*, *n*, [*cov_1*, *cov_2*, ...], [*contr_1*, *contr_2*, ...]) [Función]

Declara propiedades de simetría para el *tensor* de *m* índices covariantes y *n* contravariantes. Los *cov_i* y *contr_i* son seudofunciones que expresan relaciones de simetría entre los índices covariantes y contravariantes, respectivamente. Éstos son de la forma `symoper(index_1, index_2,...)` donde `symoper` es uno de `sym`, `anti` o `cyc` y los *index_i* son enteros que indican la posición del índice en el *tensor*. Esto declarará a *tensor* simétrico, antisimétrico o cíclico respecto de *index_i*. La llamada `symoper(all)` indica que todos los índices cumplen la condición de simetría. Por ejemplo, dado un objeto `b` con 5 índices covariantes, `decsym(b,5,3,[sym(1,2),anti(3,4)],[cyc(all)])` declara `b` simétrico en el primer y segundo índices covariantes, antisimétrico en su tercer y cuarto índices también covariantes y cíclico en todos sus índices contravariantes. Cualquiera de las listas de declaración de simetrías puede ser nula. La función que realiza las simplificaciones es `canform`, como se ilustra en el siguiente ejemplo,

```
(%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*) [Función]

Borra todas las propiedades de simetría del *tensor* que tiene *m* índices covariantes y *n* contravariantes.

`canform` (*expr*) [Función]

`canform` (*expr*, *rename*) [Función]

Simplifica *expr* renombrando índices mudos y reordenando todos los índices según las condiciones de simetría que se le hayan impuesto. Si `allsym` vale `true` entonces todos los índices se consideran simétricos, en otro caso se utilizará la información sobre simetrías suministrada por `decsym`. Los índices mudos se renombran de la misma manera que en la función `rename`. Cuando `canform` se aplica a una expresión grande el cálculo puede llevar mucho tiempo. Este tiempo se puede acortar llamando primero a `rename`. Véase también el ejemplo de la descripción de `decsym`. La función `canform` puede que no reduzca completamente una expresión a su forma más sencilla, pero en todo caso devolverá un resultado matemáticamente correcto.

Si al parámetro opcional *rename* se le asigna el valor `false`, no se renombrarán los índices mudos.

25.2.3 Cálculo tensorial indexado

`diff` (*expr*, *v_1*, [*n_1*, [*v_2*, *n_2*] ...]) [Función]

Se trata de la función de Maxima para la diferenciación, ampliada para las necesidades del paquete `itensor`. Calcula la derivada de *expr* respecto de *v_1 n_1* veces, respecto de *v_2 n_2* veces, etc. Para el paquete de tensores,la función ha sido modificada de manera que *v_i* puedan ser enteros desde 1 hasta el valor que tome la variable `dim`. Esto permite que la derivación se pueda realizar con respecto del *v_i*-ésimo miembro de la lista `vect_coords`. Si `vect_coords` guarda una variable atómica, entonces esa variable será la que se utilice en la derivación. Con esto se hace posible la utilización de una lista con nombres de coordenadas subindicadas, como `x[1]`, `x[2]`, ...

El paquete sobre tensores amplía las capacidades de `diff` con el fin de poder calcular derivadas respecto de variables indexadas. En particular, es posible derivar expresiones que contengan combinaciones del tensor métrico y sus derivadas respecto del tensor métrico y su primera y segunda derivadas. Estos métodos son particularmente útiles cuando se consideran los formalismos lagrangianos de la teoría gravitatoria, permitiendo obtener el tensor de Einstein y las ecuaciones de campo a partir del principio de acción.

`idiff` (*expr*, *v_1*, [*n_1*, [*v_2*, *n_2*] ...]) [Función]

Diferenciación inicial. Al contrario que `diff`, que deriva respecto de una variable independiente, `idiff` puede usarse para derivar respecto de una coordenada.

La función `idiff` también puede derivar el determinante del tensor métrico. Así, si `imetric` toma el valor `G` entonces `idiff(determinant(g),k)` devolverá `2*determinant(g)*ichr2([%i,k],[%i])` donde la índice mudo `%i` se escoge de forma apropiada.

`liediff` (*v*, *ten*) [Función]

Calcula la derivada de Lie de la expresión tensorial *ten* respecto de campo vectorial *v*. La expresión *ten* debe ser cualquier tensor indexado; *v* debe ser el nombre (sin índices) de un campo vectorial. Por ejemplo:

```
(%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*) [Función]

Calcula todas las instrucciones `idiff` que aparezcan en la expresión tensorial *ten*.

`undiff` (*expr*) [Función]

Devuelve una expresión equivalente a *expr* pero con todas las derivadas de los objetos indexados reemplazadas por la forma nominal de la función `idiff`.

`evundiff` (*expr*) [Función]

Equivale a `undiff` seguido de `ev` y `rediff`.

La razón de esta operación es evaluar de forma sencilla expresiones que no pueden ser directamente evaluadas en su forma derivada. Por ejemplo, lo siguiente provoca un error:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) icurvature([i,j,k],[l],m);
Maxima encountered a Lisp error:

 Error in $ICURVATURE [or a callee]:
 $ICURVATURE [or a callee] requires less than three arguments.

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
```

Sin embargo, si `icurvature` se da en forma nominal, puede ser evaluada utilizando `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: en versiones antiguas de Maxima, las formas derivadas de los símbolos de Christoffel no se podían evaluar. Este fallo ha sido subsanado, de manera que `evundiff` ya no se necesita en expresiones como esta:

```
(%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*, ...) [Función]
Iguala a cero en la expresión *expr* todas las apariciones de *tensor_i* que no tengan índices de derivadas.

`flushd` (*expr*, *tensor_1*, *tensor_2*, ...) [Función]
Iguala a cero en la expresión *expr* todas las apariciones de *tensor_i* que tengan índices de derivadas

`flushnd` (*expr*, *tensor*, *n*) [Función]
Iguala a cero en *expr* todas las apariciones del objeto diferenciado *tensor* que tenga *n* o más índices de derivadas, como demuestra el siguiente ejemplo:

```
(%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*, ...) [Función]
Le da a *tensor_i* la propiedad de diferenciación coordenada, que la derivada del vector contravariante cuyo nombre es uno de los *tensor_i* es igual a la delta de Kronecker. Por ejemplo, si se ha hecho `coord(x)` entonces `idiff(x([],[i]),j)` da `kdelta([i],[j])`. La llamada `coord` devuelve una lista de todos los objetos indexados con esta propiedad.

`remcoord` (*tensor_1*, *tensor_2*, ...) [Función]
`remcoord` (*all*) [Función]
Borra todas las propiedades de diferenciación coordenada de `tensor_i` que hayan sido establecidas por la función `coord`. La llamada `remcoord(all)` borra esta propiedad de todos los objetos indexados.

`makebox` (*expr*) [Función]
Muestra *expr* de la misma manera que lo hace `show`; sin embargo, cualquier tensor de d'Alembert que aparezca en *expr* estará indicado por `[]`. Por ejemplo, `[]p([m],[n])` representa `g([],[i,j])*p([m],[n],i,j)`.

`conmetderiv` (*expr*, *tensor*) [Función]
Simplifica expresiones que contengan derivadas ordinarias tanto de las formas covariantes como contravariantes del tensor métrico. Por ejemplo, `conmetderiv` puede

relacionar la derivada del tensor métrico contravariante con los símbolos de Christoffel, como se ve en el ejemplo:

```
(%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*) [Función]
`simpmetderiv` (*expr*[, *stop*]) [Función]

Simplifica expresiones que contienen productos de las derivadas del tensor métrico. La función `simpmetderiv` reconoce dos identidades:

```
  ab        ab           ab                  a
 g   g   + g   g     = (g   g  )   = (kdelta )   = 0
  ,d  bc        bc,d         bc ,d          c ,d
```

de donde

```
  ab          ab
 g   g   = - g   g
  ,d  bc          bc,d
```

y

```
  ab          ab
 g   g     = g   g
  ,j  ab,i    ,i  ab,j
```

que se deduce de las simetrías de los símbolos de Christoffel.

La función `simpmetderiv` tiene un argumento opcional, el cual detiene la función después de la primera sustitución exitosa en un expresión producto. La función `simpmetderiv` también hace uso de la variable global *flipflag* que determina cómo aplicar una ordenación "canónica" a los índices de los productos.

Todo esto se puede utilizar para conseguir buenas simplificaciones que serían difíciles o imposibles de conseguir, lo que se demuestra en el siguiente ejemplo, que utiliza explícitamente las simplificaciones parciales de `simpmetderiv`:

```
(%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
```

Véase también `weyl.dem` para un ejemplo que utiliza `simpmetderiv` y `conmetderiv` para simplificar contracciones del tensor de Weyl.

`flush1deriv` (*expr*, *tensor*) [Función]

Iguala a cero en `expr` todas las apariciones de `tensor` que tengan exactamente un índice derivado.

25.2.4 Tensores en espacios curvos

`imetric` (*g*) [Función]

`imetric` [Variable de sistema]

Especifica la métrica haciendo la asignación de la variable `imetric:`*g*, además las propiedades de contracción de la métrica *g* se fijan ejecutando las instrucciones `defcon(`*g*`)`, `defcon(`*g*`,`*g*`,kdelta)`. La variable `imetric`, a la que no se le asigna ningún valor por defecto, tiene el valor de la métrica que se le haya asignado con la instrucción `imetric(`*g*`)`.

idim (*n*) [Función]

Establece las dimensiones de la métrica. También inicializa las propiedades de antisimetría de los símbolos de Levi-Civita para la dimensión dada.

ichr1 ([*i*, *j*, *k*]) [Función]

Devuelve el símbolo de Christoffel de primera especie dado por la definición

```
(g      + g      - g     )/2 .
  ik,j     jk,i     ij,k
```

Para evaluar los símbolos de Christoffel de una métrica determinada, a la variable `imetric` hay que asignarle un nombre como en el ejemplo de la descripción de `chr2`.

ichr2 ([*i*, *j*], [*k*]) [Función]

Devuelve el símbolo de Christoffel de segunda especie dado por la definición

```
                          ks
ichr2([i,j],[k]) = g    (g      + g      - g     )/2
                           is,j     js,i     ij,s
```

icurvature ([*i*, *j*, *k*], [*h*]) [Función]

Devuelve el tensor de curvatura de Riemann en términos de los símbolos de Christoffel de segunda especie (`ichr2`). Se utiliza la siguiente notación:

```
                  h             h            h         %1         h
icurvature     = - ichr2      - ichr2     ichr2    + ichr2
                  i j k         i k,j        %1 j      i k        i j,k
                                 h           %1
                           + ichr2      ichr2
                                 %1 k        i j
```

covdiff (*expr*, *v_1*, *v_2*, ...) [Función]

Devuelve la derivada covariante de *expr* respecto de las variables *v_i* en términos de los símbolos de Christoffel de segunda especie (`ichr2`). Para evaluarlos debe hacerse `ev(`*expr*`,ichr2)`.

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) entertensor()$
Enter tensor name: a;
Enter a list of the covariant indices: [i,j];
Enter a list of the contravariant indices: [k];
Enter a list of the derivative indices: [];
                                      k
(%t2)                                a
                                      i j
(%i3) ishow(covdiff(%,s))$
              k         %1     k         %1     k
(%t3)     - a     ichr2    - a     ichr2    + a
               i %1      j s    %1 j      i s    i j,s
```

```
             k     %1
      + ichr2     a
             %1 s  i j
(%i4) imetric:g;
(%o4)                                    g
(%i5) ishow(ev(%th(2),ichr2))$
         %1 %4  k
        g      a     (g       - g       + g      )
                i %1   s %4,j    j s,%4    j %4,s
(%t5) - ----------------------------------------
                            2
    %1 %3  k
   g      a     (g       - g       + g      )
           %1 j   s %3,i    i s,%3    i %3,s
 - ----------------------------------------
                       2
    k %2  %1
   g     a    (g        - g        + g       )
          i j   s %2,%1    %1 s,%2    %1 %2,s     k
 + ---------------------------------------- + a
                       2                         i j,s
```

`lorentz_gauge` (*expr*) [Función]

Impone la condición de Lorentz sustituyendo por 0 todos los objetos indexados de *expr* que tengan un índice derivado idéntico a un índice contravariante.

`igeodesic_coords` (*expr*, *nombre*) [Función]

Elimina los símbolos no diferenciados de Christoffel y las primeras derivadas del tensor métrico de *expr*. El argumento *nombre* de la función `igeodesic_coords` se refiere a la métrica *nombre* si aparece en *expr*, mientras que los coeficientes de conexión deben tener los nombres `ichr1` y/o `ichr2`. El siguiente ejemplo hace la verificación de la identidad cíclica satisfecha por el tensor de curvatura de Riemann haciendo uso de la función `igeodesic_coords`.

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(icurvature([r,s,t],[u]))$
             u            u         %1         u
(%t2) - ichr2      - ichr2     ichr2    + ichr2
             r t,s        %1 s      r t        r s,t

                                                u         %1
                                           + ichr2     ichr2
                                                %1 t      r s
(%i3) ishow(igeodesic_coords(%,ichr2))$
                                     u            u
```

```
(%t3)                          ichr2      - ichr2
                                    r s,t        r t,s
(%i4) ishow(igeodesic_coords(icurvature([r,s,t],[u]),ichr2)+
           igeodesic_coords(icurvature([s,t,r],[u]),ichr2)+
           igeodesic_coords(icurvature([t,r,s],[u]),ichr2))$
            u            u            u            u
(%t4) - ichr2      + ichr2      + ichr2      - ichr2
            t s,r        t r,s        s t,r        s r,t

                                                 u            u
                                           - ichr2      + ichr2
                                                 r t,s        r s,t
(%i5) canform(%);
(%o5)                                  0
```

25.2.5 Sistemas de referencia móviles

Maxima puede hacer cálculos utilizando sistemas de referencia móviles, los cuales pueden ser ortonormales o cualesquiera otros.

Para utilizar sistemas de referencia, primero se debe asignar a la variable `iframe_flag` el valor `true`. Con esto se hace que los símbolos de Christoffel, `ichr1` y `ichr2`, sean reemplazados por los coeficientes `icc1` y `icc2` en los cálculos, cambiando así el comportamiento de `covdiff` y `icurvature`.

El sistema de referencia se define con dos tensores: el campo del sistema de referencia inverso (`ifri`, la base dual tetrad) y la métrica del sistema de referencia `ifg`. La métrica del sistema de referencia es la matriz identidad en los sistemas de referencia ortonormales, o la métrica de Lorentz en sistemas de referencia ortonormales en el espacio-tiempo de Minkowski. El campo del sistema de referencia inverso define la base del sistema de referencia con vectores unitarios. Las propiedades contractivas se definen para el campo y la métrica del sistema de referencia.

Si `iframe_flag` vale `true`, muchas expresiones de `itensor` utilizan la métrica `ifg` en lugar de la métrica definida por `imetric` para incrementar y reducir índices.

IMPORTANTE: Asignando a la variable `iframe_flag` el valor `true` NO deshace las propiedades contractivas de una métrica establecidas con una llamada a `defcon` o a `imetric`. Si se utiliza el campo del sistema de referencia, es mejor definir la métrica asignando su nombre a la variable `imetric` y NO hacer una llamada a la función `imetric`.

Maxima utiliza estos dos tensores para definir los coeficientes del sistema de referencia: `ifc1` y and `ifc2`, los cuales forman parte de los coeficientes de conexión `icc1` y `icc2`, tal como demuestra el siguiente ejemplo:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) iframe_flag:true;
(%o2)                              true
(%i3) ishow(covdiff(v([],[i]),j))$
```

```
                                i        i     %1
(%t3)                          v   + icc2     v
                                ,j       %1 j
(%i4) ishow(ev(%,icc2))$
                                %1     i       i
(%t4)                          v   ifc2     + v
                                       %1 j    ,j
(%i5) ishow(ev(%,ifc2))$
                           %1    i %2                i
(%t5)                     v   ifg     ifc1        + v
                                          %1 j %2    ,j
(%i6) ishow(ev(%,ifc1))$
            %1    i %2
          v   ifg     (ifb        - ifb        + ifb       )
                           j %2 %1      %2 %1 j      %1 j %2     i
(%t6)      ------------------------------------------------- + v
                                  2                              ,j
(%i7) ishow(ifb([a,b,c]))$
                                                    %3    %4
(%t7)               (ifri        - ifri       ) ifr   ifr
                         a %3,%4       a %4,%3     b     c
```

Se utiliza un método alternativo para calcular el sistema de referencia `ifb` si la variable `iframe_bracket_form` vale `false`:

```
(%i8) block([iframe_bracket_form:false],ishow(ifb([a,b,c])))$
                                 %6    %5        %5      %6
(%t8)              ifri     (ifr   ifr     - ifr     ifr  )
                       a %5     b     c,%6      b,%6    c
```

`ifb` [Variable]

Es el sistema de referencia soporte. La contribución de la métrica del campo a los coeficientes de conexión se expresa utilizando:

$$ifc1_{abc} = \frac{-ifb_{cab} + ifb_{bca} + ifb_{abc}}{2}$$

El sistema de referencia soporte se define en términos del campo y la métrica del sistema de referencia. Se utilizan dos métodos alternativos dependiendo del valor de `frame_bracket_form`. Si vale `true` (que es el valor por defecto) o si `itorsion_flag` vale `true`:

```
                d      e                                        f
```

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

En otro caso:

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

`icc1` [Variable]

Coeficientes de conexión de primera especie. Se definen en `itensor` como

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

En esta expresión, si `iframe_flag` vale `true`, el símbolo de Christoffel `ichr1` se reemplaza por el coeficiente de conexión del sistema de referencia `ifc1`. Si `itorsion_flag` vale `false`, `ikt1` será omitido. También se omite si se utiliza una base, ya que la torsión ya está calculada como parte del sistema de referencia.

`icc2` [Variable]

Coeficientes de conexión de segunda especie. Se definen en `itensor` como

```
    c         c        c         c
icc2   = ichr2   - ikt2   - inmc2
    ab        ab       ab        ab
```

En esta expresión, si la variable `iframe_flag` vale `true`, el símbolo de Christoffel `ichr2` se reemplaza por el coeficiente de conexión del sistema de referencia `ifc2`. Si `itorsion_flag` vale `false`, `ikt2` se omite. También se omite si se utiliza una base de referncia. Por último, si `inonmet_flag` vale `false`, se omite `inmc2`.

`ifc1` [Variable]

Coeficiente del sistema de referencia de primera especie, también conocido como coeficientes de rotación de Ricci. Este tensor represnta la contribución de la métrica del sistema de referencia al coeficiente de conexión de primera especie, definido como

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

`ifc2` [Variable]

Coeficiente del sistema de referencia de segunda especie. Este tensor representa la contribución de la métrica del sistema de referencia al coeficiente de conexión de segunda especie, definido como

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

`ifr` [Variable]

El campo del sistema de referencia. Se contrae con el campo inverso `ifri` para formar la métrica del sistema de referencia, `ifg`.

`ifri` [Variable]

Campo inverso del sistema de referencia. Especifica la base del sistema de referencia (vectores de la base dual).

`ifg` [Variable]

La métrica del sistema de referencia. Su valor por defecto es `kdelta`, pero puede cambiarse utilizando `components`.

`ifgi` [Variable]

La métrica inversa del sistema de referencia. Se contrae con la métrica `ifg` para dar `kdelta`.

`iframe_bracket_form` [Variable opcional]

Valor por defecto: `true`

Especifica cómo se calcula `ifb`.

25.2.6 Torsión y no metricidad

Maxima trabaja con conceptos como la torsión y la no metricidad. Cuando la variable `itorsion_flag` vale `true`, la contribución de la torsión se añade a los coeficientes de conexión. También se añaden las componentes de no metricidad cuando `inonmet_flag` vale `true`.

`inm` [Variable]

Vector de no metricidad. La no metricidad conforme se define a partir de la derivada covariante del tensor métrico. La derivada covariante del tensor métrico, que normalmente es nula, se calcula, cuando `inonmet_flag` vale `true`, como

```
g     =- g  inm
 ij;k     ij   k
```

`inmc1` [Variable]

Permutación covariante de las componentes del vector de no metricidad. Se define como

```
g   inm  - inm  g   - g   inm
```

```
             ab    c      a  bc    ac    b
inmc1    = ------------------------------
     abc                 2
```

(Sustitúyase g por ifg si se utiliza una métrica para el sistema de referencia.)

inmc2 [Variable]

Permutación contravariante de las componentes del vector de no metricidad. Se utiliza en los coeficientes de conexión si inonmet_flag vale true. Se define como

```
                       c         c         cd
           -inm  kdelta  - kdelta  inm  + g   inm  g
     c        a       b         a    b          d  ab
inmc2   = ----------------------------------------------
     ab                       2
```

(Sustitúyase g por ifg si se utiliza una métrica para el sistema de referencia.)

ikt1 [Variable]

Permutación covariante del tensor de permutación, también conocido como contorsión. Se define como

```
                  d           d       d
           -g   itr  - g    itr   - itr   g
             ad    cb    bd    ca      ab  cd
ikt1    = ----------------------------------
    abc                  2
```

(Sustitúyase g por ifg si se utiliza una métrica para el sistema de referencia.)

ikt2 [Variable]

Permutación contravariante del tensor de permutación, también conocido como contorsión. Se define como

```
    c     cd
ikt2   = g   ikt1
    ab           abd
```

(Sustitúyase g por ifg si se utiliza una métrica para el sistema de referencia.)

itr [Variable]

Tensor de torsión. Para una métrica con torsión, la diferenciación covariante iterada de una función escalar no conmuta, tal como demuestra el siguiente ejemplo:

```
(%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
```

25.2.7 Álgebra exterior

Con el paquete `itensor` se pueden realizar operaciones en campos tensoriales covariantes antisimétricos. Un campo tensorial totalmente antisimétrrico de rango (0,L) se corresponde con una L-forma diferencial. Sobre estos objetos se define una operación que se llama producto exterior.

Desafortunadamente no hay consenso entre los autores a la hora de definir el producto exterior. Algunos autores prefieren una definición que se corresponde con la noción de antisimetrización, con lo que el producto externo de dos campos vectoriales se definiría como

```
                a a  - a a
                 i j    j i
      a  /\ a  = -----------
       i     j        2
```

De forma más general, el producto de una p-forma por una q-forma se definiría 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
```

donde `D` es la delta de Kronecker.

Otros autores, sin embargo, prefieren una definición "geométrica" que se corresponde con la noción del elemento de volumen,

```
a  /\ a  = a a  - a a
 i     j    i j    j i
```

y, en el caso general,

```
                           1    k1..kp l1..lq
A       /\ B       = ----- D              A       B
 i1..ip     j1..jq   p! q!  i1..ip j1..jq  k1..kp  l1..lq
```

Puesto que `itensor` un paquete de álgebra tensorial, la primera de estas dos definiciones parece la más natural. Sin embargo, muchas aplicaciones hacen uso de la segunda definición. Para resolver el dilema, se define una variable que controla el comportamiento del producto exteriort: si `igeowedge_flag` vale `false` (el valor por defecto), se utiliza la primera definición, si vale `true`, la segunda.

`~` [Operador]

El operador del producto exterior se representa por el símbolo `~`. Este es un operador binario. Sus argumentos deben ser expresiones que tengan escalares, tensores covariantes de rango uno o tensores covariantes de rango `l` que hayan sido declarados antisimétricos en todos los índices covariantes.

El comportamiento del operador del producto exterior se controla con la variable `igeowedge_flag`, como en el ejemplo siguiente:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(a([i])~b([j]))$
                                    a  b  - b  a
                                     i  j    i  j
```

```
(%t2)                                 -------------
                                            2
(%i3) decsym(a,2,0,[anti(all)],[]);
(%o3)                                    done
(%i4) ishow(a([i,j])~b([k]))$
                             a    b  + b  a    - a    b
                              i j  k    i  j k    i k  j
(%t4)                        ---------------------------
                                            3
(%i5) igeowedge_flag:true;
(%o5)                                    true
(%i6) ishow(a([i])~b([j]))$
(%t6)                                 a  b  - b  a
                                       i  j    i  j
(%i7) ishow(a([i,j])~b([k]))$
(%t7)                        a    b  + b  a    - a    b
                              i j  k    i  j k    i k  j
```

| [Operador]

La barra vertical | representa la operación "contracción con un vector". Cuando un tensor covariante totalmente antisimétrico se contrae con un vector contravariante, el resultado no depende del índice utilizado para la contracción. Así, es posible definir la operación de contracción de forma que no se haga referencia al índice.

En el paquete `itensor` la contracción con un vector se realiza siempre respecto del primer índice de la ordenación literal. Ejemplo:

```
(%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
```

Nótese que es primordial que los tensores utilizados junto con el operador | se declaren totalmente antisimétricos en sus índices covariantes. De no ser así, se pueden obtener resultados incorrectos.

`extdiff` (*expr*, *i*) [Función]

Calcula la derivada exterior de *expr* con respecto del índice *i*. La derivada exterior se define formalmente como el producto exterior del operador de la derivada parcial y una forma diferencial. Por lo tanto, esta operación también se ve afectada por el valor que tome la variable `igeowedge_flag`. Ejemplo:

```
(%i1) load("itensor");
```

```
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(extdiff(v([i]),j))$
                                    v    - v
                                     j,i    i,j
(%t2)                               -----------
                                         2
(%i3) decsym(a,2,0,[anti(all)],[]);
(%o3)                                  done
(%i4) ishow(extdiff(a([i,j]),k))$
                              a      - a      + a
                               j k,i    i k,j    i j,k
(%t4)                         ------------------------
                                        3
(%i5) igeowedge_flag:true;
(%o5)                                  true
(%i6) ishow(extdiff(v([i]),j))$
(%t6)                               v    - v
                                     j,i    i,j
(%i7) ishow(extdiff(a([i,j]),k))$
(%t7)                     - (a      - a      + a     )
                             k j,i    k i,j    j i,k
```

hodge (*expr*) [Función]

Calcula el dual de Hodge *expr*. Por ejemplo:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) imetric(g);
(%o2)                             done
(%i3) idim(4);
(%o3)                             done
(%i4) icounter:100;
(%o4)                              100
(%i5) decsym(A,3,0,[anti(all)],[])$

(%i6) ishow(A([i,j,k],[]))$
(%t6)                            A
                                  i j k
(%i7) ishow(canform(hodge(%)))$
                           %1 %2 %3 %4
               levi_civita            g        A
                                       %1 %102  %2 %3 %4
(%t7)          ----------------------------------------
                                  6
(%i8) ishow(canform(hodge(%)))$
                  %1 %2 %3 %8            %4 %5 %6 %7
```

```
(%t8) levi_civita            levi_civita             g
                                                      %1 %106
                             g        g        g      A          /6
                              %2 %107  %3 %108  %4 %8  %5 %6 %7
(%i9) lc2kdt(%)$

(%i10) %,kdelta$

(%i11) ishow(canform(contract(expand(%))))$
(%t11)                      - A
                               %106 %107 %108
```

`igeowedge_flag` [Variable opcional]

Valor por defecto: `false`

Controla el comportamiento del producto exterior y de la derivada exterior. Cuando vale `false`, la noción de formas diferenciales se corresponde con el de campo tensorial covariante totalmente antisimétrico. Cuando vale `true`, las formas diferenciales se corresponden con la idea de elemento de volumen.

25.2.8 Exportando expresiones en TeX

El paquete `itensor` dispone de soporte limitado para exportar expresiones con tensores a TeX. Puesto que las expresiones de `itensor` son llamadas a funciones, puede que la instrucción habitual en Maxima, `tex`, no devuleva los resultados esperados. Se puede utlizar el comando `tentex`, que tratará de traducir expresiones tensoriales a objetos de TeX correctamente indexados.

`tentex` (*expr*) [Función]

Para utilizar la función `tentex`, primero se debe cargar `tentex`, tal como muestra el siguiente ejemplo:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) load("tentex");
(%o2)       /share/tensor/tentex.lisp
(%i3) idummyx:m;
(%o3)                                  m
(%i4) ishow(icurvature([j,k,l],[i]))$
            m1       i           m1       i           i
(%t4)  ichr2    ichr2     - ichr2    ichr2     - ichr2
            j k      m1 l        j l      m1 k        j l,k

                                                     i
                                                + ichr2
                                                       j k,l
(%i5) tentex(%)$
$$\Gamma_{j\,k}^{m_1}\,\Gamma_{l\,m_1}^{i}-\Gamma_{j\,l}^{m_1}\,
```

```
\Gamma_{k\,m_1}^{i}-\Gamma_{j\,l,k}^{i}+\Gamma_{j\,k,l}^{i}$$
```

Nótese la asignación de la variable `idummyx` para evitar la aparición del símbolo del porcentaje en la expresión en TeX, que puede dar errores de compilación.

Téngase en cuenta que esta versión de la función `tentex` es experimental.

25.2.9 Interactuando con ctensor

El paquete `itensor` genera código Maxima que luego puede ser ejecutado en el contexto del paquete `ctensor`. La función que se encarga de esta tarea es `ic_convert`.

`ic_convert` (*eqn*) [Function]

Convierte la ecuación *eqn* del entorno `itensor` a una sentencia de asignación de `ctensor`. Sumas implícitas sobre índices mudos se hacen explícitas mientras que objetos indexados se transforman en arreglos (los subíndices de los arreglos se ordenan poniendo primero los covariantes seguidos de los contravariantes. La derivada de un objeto indexado se reemplazará por por la forma nominal de `diff` tomada con respecto a `ct_coords` con el subíndice correspondiente al índice derivado. Los símbolos de Christoffel `ichr1 ichr2` se traducen a `lcs` y `mcs`, respectivamente. Además, se añaden bucles `do` para la sumación de todos los índices libres, de manera que la sentencia traducida pueda ser evaluada haciendo simplemente `ev`. Los siguientes ejemplos muestran las funcionalidades de esta función.

```
(%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
```

25.2.10 Palabras reservadas

Las siguientes palabras son utilizadas por el paquete `itensor` internamente, por lo que no deberían ser modificadas por el usuario:

Palabra	Comentarios
indices2()	Versión interna de indices()
conti	Lista los índices contravariantes
covi	Lista los índices covariantes
deri	Lista los índices de derivadas
name	Devuelve el nombre de un objeto indexado
concan	
irpmon	
lc0	
_lc2kdt0	
_lcprod	
_extlc	

26 ctensor

26.1 Introducción a ctensor

El paquete `ctensor` dispone de herramientas para manipular componentes de tensores. Para poder hacer uso de `ctensor` es necesario cargarlo previamente en memoria ejecutando `load("ctensor")`. Para comenzar una sesión interactiva con `ctensor`, ejecutar la función `csetup()`. Primero se le pregunta al usuario la dimensión de la variedad. Si la dimensión es 2, 3 o 4, entonces la lista de coordenadas será por defecto `[x,y]`, `[x,y,z]` o `[x,y,z,t]`, respectivamente. Estos nombres pueden cambiarse asignando una nueva lista de coordenadas a la variable `ct_coords` (que se describe más abajo), siendo el usuario advertido sobre este particular. Se debe tener cuidado en evitar que los nombres de las coordenadas entren en conflicto con los nombres de otros objetos en Maxima.

A continuación, el usuario introduce la métrica, bien directamente, o desde un fichero especificando su posición ordinal. La métrica se almacena en la matriz `lg`. Por último, la métrica inversa se obtiene y almacena en la matriz `ug`. También se dispone de la opción de efectuar todos los cálculos en serie de potencias.

Se desarrolla a continuación un ejemplo para la métrica estática, esférica y simétrica, en coordenadas estándar, que se aplicará posteriormente al problema de derivar las ecuaciones de vacío de Einstein (de las que se obtiene la solución de Schwarzschild). Muchas de las funciones de `ctensor` se mostrarán en los ejemplos para la métrica estándar.

```
(%i1) load("ctensor");
(%o1)      /share/tensor/ctensor.mac
(%i2) csetup();
Enter the dimension of the coordinate system:
4;
Do you wish to change the coordinate names?
n;
Do you want to
1. Enter a new metric?

2. Enter a metric from a file?

3. Approximate a metric with a Taylor series?
1;

Is the matrix  1. Diagonal  2. Symmetric  3. Antisymmetric  4. General
Answer 1, 2, 3 or 4
1;
Row 1 Column 1:
a;
Row 2 Column 2:
x^2;
Row 3 Column 3:
x^2*sin(y)^2;
Row 4 Column 4:
```

```
-d;

Matrix entered.
Enter functional dependencies with the DEPENDS function or 'N' if none
depends([a,d],x);
Do you wish to see the metric?
y;
                          [ a  0       0        0  ]
                          [                        ]
                          [     2                  ]
                          [ 0  x       0        0  ]
                          [                        ]
                          [         2    2         ]
                          [ 0  0   x  sin (y)   0  ]
                          [                        ]
                          [ 0  0       0       - d ]
(%o2)                                done
(%i3) christof(mcs);
                                              a
                                             x
(%t3)                          mcs        = ---
                                  1, 1, 1   2 a

                                              1
(%t4)                           mcs        = -
                                   1, 2, 2   x

                                              1
(%t5)                           mcs        = -
                                   1, 3, 3   x

                                             d
                                             x
(%t6)                          mcs        = ---
                                  1, 4, 4   2 d

                                               x
(%t7)                          mcs        = - -
                                  2, 2, 1     a

                                            cos(y)
(%t8)                         mcs        = ------
                                 2, 3, 3   sin(y)

                                                2
                                           x sin (y)
(%t9)                     mcs        = - ---------
```

```
                                3, 3, 1         a

(%t10)                  mcs         = - cos(y) sin(y)
                           3, 3, 2

                                                 d
                                                  x
(%t11)                        mcs         = ---
                                 4, 4, 1    2 a
(%o11)                           done
```

26.2 Funciones y variables para ctensor

26.2.1 Inicialización y preparación

`csetup` () [Función]

Es la función del paquete `ctensor` que inicializa el paquete y permite al usuario introducir una métrica de forma interactiva. Véase `ctensor` para más detalles.

`cmetric` (*dis*) [Función]

`cmetric` () [Función]

Es la función del paquete `ctensor` que calcula la métrica inversa y prepara el paquete para cálculos ulteriores.

Si `cframe_flag` vale `false`, la función calcula la métrica inversa `ug` a partir de la matriz `lg` definida por el usuario. Se calcula también la métrica determinante y se almacena en la variable `gdet`. Además, el paquete determina si la métrica es diagonal y ajusta el valor de `diagmetric` de la forma apropiada. Si el argumento opcional *dis* está presente y no es igual a `false`, el usuario podrá ver la métrica inversa.

Si `cframe_flag` vale `true`, la función espera que los valores de `fri` (la matriz del sistema de referencia inverso) y `lfg` (la matriz del sistema de referencia) estén definidos. A partir de ellos, se calculan la matriz del sistema de referencia `fr` y su métrica `ufg`.

`ct_coordsys` (*sistema_coordenadas*, *extra_arg*) [Función]

`ct_coordsys` (*sistema_coordenadas*) [Función]

Prepara un sistema de coordenadas predefinido y una métrica. El argumento *sistema_coordenadas* puede ser cualquiera de los siguientes símbolos:

Símbolo	Dim	Coordenadas	Descripción/comentarios
cartesian2d	2	[x,y]	Sistema de coordenadas cartesianas e
polar	2	[r,phi]	Sistema de coordenadas polares
elliptic	2	[u,v]	Sistema de coordenadas elípticas
confocalelliptic	2	[u,v]	Coordenadas elípticas confocales
bipolar	2	[u,v]	Sistema de coordenas bipolares
parabolic	2	[u,v]	Sistema de coordenadas parabólicas
cartesian3d	3	[x,y,z]	Sistema de coordenadas cartesianas e

```
polarcylindrical      3  [r,theta,z]       Polares en 2D con cilíndrica z
ellipticcylindrical   3  [u,v,z]           Elípticas en 2D con cilíndrica z
confocalellipsoidal   3  [u,v,w]           Elipsoidales confocales
bipolarcylindrical    3  [u,v,z]           Bipolares en 2D con cilíndrica z
paraboliccylindrical  3  [u,v,z]           Parabólicas en 2D con cilíndrica z
paraboloidal          3  [u,v,phi]         Coordenadas paraboloidales
conical               3  [u,v,w]           Coordenadas cónicas
toroidal              3  [u,v,phi]         Coordenadas toroidales
spherical             3  [r,theta,phi]     Sistema de coordenadas esféricas
oblatespheroidal      3  [u,v,phi]         Coordenadas esferoidales obleadas
oblatespheroidalsqrt  3  [u,v,phi]
prolatespheroidal     3  [u,v,phi]         Coordenadas esferoidales prolatas
prolatespheroidalsqrt 3  [u,v,phi]
ellipsoidal           3  [r,theta,phi]     Coordenadas elipsoidales
cartesian4d           4  [x,y,z,t]         Sistema de coordenadas cartesianas e
spherical4d           4  [r,theta,eta,phi] Sistema de coordenadas esféricas en
exteriorschwarzschild 4  [t,r,theta,phi]   Métrica de Schwarzschild
interiorschwarzschild 4  [t,z,u,v]         Métrica interior de Schwarzschild
kerr_newman           4  [t,r,theta,phi]   Métrica simétrica con carga axial
```

El argumento `sistema_coordenadas` puede ser también una lista de funciones de transformación, seguida de una lista que contenga los nombres de las coordenadas. Por ejemplo, se puede especificar una métrica esférica como se indica a continuación:

```
(%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
```

Las funciones de transformación se pueden utilizar también si `cframe_flag` vale `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) ]
```

El argumento opcional *extra_arg* puede ser cualquiera de los siguientes:

`cylindrical` indica a `ct_coordsys` que añada una coordenada cilíndrica más.

`minkowski` indica a `ct_coordsys` que añada una coordenada más con signatura métrica negativa.

`all` indica a `ct_coordsys` que llame a `cmetric` y a `christof(false)` tras activar la métrica.

Si la variable global `verbose` vale `true`, `ct_coordsys` muestra los valores de `dim`, `ct_coords`, junto con `lg` o `lfg` y `fri`, dependiendo del valor de `cframe_flag`.

`init_ctensor` () [Función]

Inicializa el paquete `ctensor`.

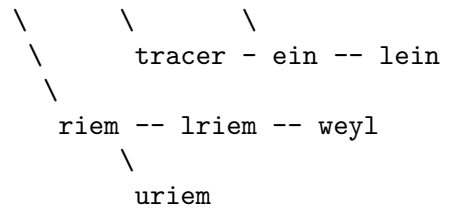
La función `init_ctensor` reinicializa el paquete `ctensor`. Borra todos los arreglos ("arrays") y matrices utilizados por `ctensor` y reinicializa todas las variables, asignando a `dim` el valor 4 y la métrica del sistema de referencia a la de Lorentz.

26.2.2 Los tensores del espacio curvo

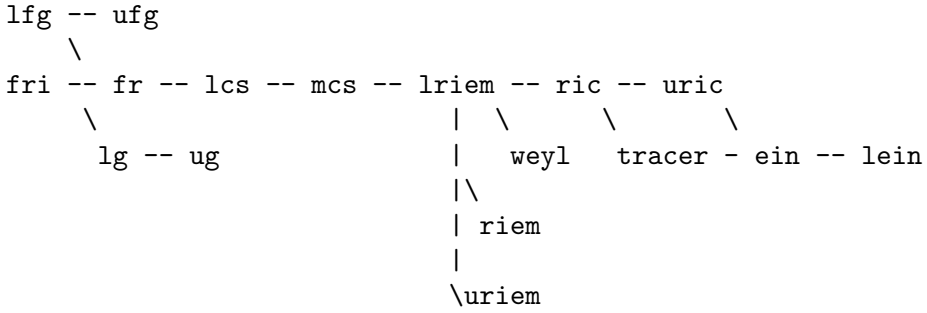
El propósito principal del paquete `ctensor` es calcular los tensores del espacio (-tiempo) curvo, en especial los tensores utilizados en relatividad general.

Cuando se utiliza una métrica, `ctensor` puede calcular los siguientes tensores:

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



El paquete `ctensor` también puede trabajar con sistemas de referencia móviles. Si `cframe_flag` vale `true`, se pueden calcular los siguientes tensores:



`christof` (*dis*) [Función]

Es una función del paquete `ctensor`. Calcula los símbolos de Christoffel de ambos tipos. El argumento *dis* determina qué resultados se mostrarán de forma inmediata. Los símbolos de Christoffel de primer y segundo tipo se almacenan en los arreglos `lcs[i,j,k]` y `mcs[i,j,k]`, respectivamente, y se definen simétricos en sus dos primeros índices. Si el argumento de `christof` es `lcs` o `mcs` entonces serán mostrados únicamente los valores no nulos de `lcs[i,j,k]` o `mcs[i,j,k]`, respectivamente. Si el argumento es `all` entonces se mostrarán los valores no nulos de `lcs[i,j,k]` y `mcs[i,j,k]`. Si el argumento vale `false` entonces no se mostrarán los elementos. El arreglo `mcs[i,j,k]` está definido de tal modo que el último índice es contravariante.

`ricci` (*dis*) [Función]

Es una función del paquete `ctensor`. La función `ricci` calcula las componentes covariantes (simétricas) `ric[i,j]` del tensor de Ricci. Si el argumento *dis* vale `true`, entonces se muestran las componentes no nulas.

`uricci` (*dis*) [Función]

Esta función calcula en primer lugar las componentes covariantes `ric[i,j]` del tensor de Ricci. Después se calcula el tensor de Ricci utilizando la métrica contravariante. Si el valor del argumento *dis* vale `true`, entonces se mostrarán directamente las componentes `uric[i,j]` (el índice i es covariante y el j contravariante). En otro caso, `ricci(false)` simplemente calculará las entradas del arreglo `uric[i,j]` sin mostrar los resultados.

`scurvature` () [Función]

Devuelve la curvatura escalar (obtenida por contracción del tensor de Ricci) de la variedad de Riemannian con la métrica dada.

`einstein` (*dis*) [Función]

Es una función del paquete `ctensor`. La función `einstein` calcula el tensor de Einstein después de que los símbolos de Christoffel y el tensor de Ricci hayan sido calculados (con las funciones `christof` y `ricci`). Si el argumento *dis* vale `true`, entonces se mostrarán los valores no nulos del tensor de Einstein `ein[i,j]`, donde `j` es el índice contravariante. La variable `rateinstein` causará la simplificación racional de estas componentes. Si `ratfac` vale `true` entonces las componentes también se factorizarán.

`leinstein` (*dis*) [Función]

Es el tensor covariante de Einstein. La función `leinstein` almacena los valores del tensor covariante de Einstein en el arreglo `lein`. El tensor covariante de Einstein se calcula a partir del tensor de Einstein `ein` multiplicándolo por el tensor métrico. Si el argumento *dis* vale `true`, entonces se mostrarán los valores no nulos del tensor covariante de Einstein.

`riemann` (*dis*) [Función]

Es una función del paquete `ctensor`. La función `riemann` calcula el tensor de curvatura de Riemann a partir de la métrica dada y de los símbolos de Christoffel correspondientes. Se utiliza el siguiente convenio sobre los índices:

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

Esta notación es consistente con la notación utilizada por el paquete `itensor` y su función `icurvature`. Si el argumento opcional *dis* vale `true`, se muestran las componentes no nulas únicas de `riem[i,j,k,l]`. Como en el caso del tensor de Einstein, ciertas variables permiten controlar al usuario la simplificación de las componentes del tensor de Riemann. Si `ratriemann` vale `true`, entonces se hará la simplificación racional. Si `ratfac` vale `true`, entonces se factorizarán todas las componentes.

Si la variable `cframe_flag` vale `false`, el tensor de Riemann se calcula directamente a partir de los símbolos de Christoffel. Si `cframe_flag` vale `true`, el tensor covariante de Riemann se calcula a partir de los coeficientes del campo.

`lriemann` (*dis*) [Función]

Es el tensor covariante de Riemann (`lriem[]`).

Calcula el tensor covariante de Riemann como un arreglo `lriem`. Si el argumento *dis* vale `true`, sólo se muestran los valores no nulos.

Si la variable `cframe_flag` vale `true`, el tensor covariante de Riemann se calcula directamente de los coeficientes del campo. En otro caso, el tensor de Riemann (3,1) se calcula en primer lugar.

Para más información sobre la ordenación de los índices, véase `riemann`.

`uriemann` (*dis*) [Función]

Calcula las componentes contravariantes del tensor de curvatura de Riemann como un arreglo `uriem[i,j,k,l]`. Éstos se muestran si *dis* vale `true`.

`rinvariant` () [Función]

Calcula la invariante de Kretchmann (`kinvariant`) obtenida por contracción de los tensores.

```
lriem[i,j,k,l]*uriem[i,j,k,l].
```

Este objeto no se simplifica automáticamente al ser en ocasiones muy grande.

`weyl` (*dis*) [Función]

Calcula el tensor conforme de Weyl. Si el argumento *dis* vale `true`, se le mostrarán al usuario las componentes no nulas `weyl[i,j,k,l]`. En otro caso, estas componentes serán únicamente calculadas y almacenadas. Si la variable `ratweyl` vale `true`, entonces las componentes se simplifican racionalmente; si `ratfac` vale `true` los resultados también se simplificarán.

26.2.3 Desarrollo de Taylor

El paquete `ctensor` puede truncar resultados e interpretarlos como aproximaciones de Taylor. Este comportamiento se controla con lavariable `ctayswitch`; cuando vale `true`, `ctensor` utiliza internamente la función `ctaylor` cuando simplifica resultados.

La función `ctaylor` es llamada desde las siguientes funciones del paquete `ctensor`:

```
Función      Comentarios
---------------------------------
christof()   Sólo para mcs
ricci()
uricci()
einstein()
riemann()
weyl()
checkdiv()
```

`ctaylor` () [Función]

La función `ctaylor` trunca su argumento convirtiéndolo en un desarrollo de Taylor por medio de la función `taylor` e invocando después a `ratdisrep`. Esto tiene el efecto de eliminar términos de orden alto en la variable de expansión `ctayvar`. El orden de los términos que deben ser eliminados se define `ctaypov`; el punto alrededor del cual se desarrolla la serie se especifica en `ctaypt`.

Como ejemplo, considérese una sencilla métrica que es una perturbación de la de Minkowski. Sin añadir restricciones, incluso una métrica diagonal produce expansiones del tensor de Einstein que pueden llegar a ser muy complejas:

```
(%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
```

Sin embargo, si se recalcula este ejemplo como una aproximación lineal en la variable `l`, se obtienen expresiones más sencillas:

```
(%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
```

Esta capacidad del paquete `ctensor` puede ser muy útil; por ejemplo, cuando se trabaja en zonas del campo gravitatorio alejadas del origen de éste.

26.2.4 Campos del sistema de referencia

Cuando la variable `cframe_flag` vale `true`, el paquete `ctensor` realiza sus cálculos utilizando un sistema de referencia móvil.

`frame_bracket` (*fr*, *fri*, *diagframe*) [Función]

Es el sistema de referencia soporte (`fb[]`).

Calcula el soporte del sistema de referencia de acuerdo con la siguiente definición:

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

26.2.5 Clasificación algebraica

Una nueva funcionalidad (Noviembre de 2004) de `ctensor` es su capacidad de obtener la clasificación de Petrov de una métrica espaciotemporal de dimensión 4. Para una demostración de esto véase el fichero `share/tensor/petrov.dem`.

`nptetrad` () [Función]

Calcula la cuaterna nula de Newman-Penrose (`np`). Véase `petrov` para un ejemplo.

La cuaterna nula se construye bajo la suposición de que se está utilizando una métrica tetradimensional ortonormal con signatura métrica (-,+,+,+). Los componentes de la cuaterna nula se relacionan con la inversa de la matriz del sistema de referencia de la siguiente manera:

```
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*) [Función]

Calcula los cinco coeficientes de Newman-Penrose `psi[0]`...`psi[4]`. Si `dis` vale `true`, se muestran estos coeficientes. Véase `petrov` para un ejemplo.

Estos coeficientes se calculan a partir del tensor de Weyl.

`petrov` () [Función]

Calcula la clasificación de Petrov de la métrica caracterizada por `psi[0]`...`psi[4]`.

Por ejemplo, lo que sigue demuestra cómo obtener la clasificación de Petrov para la 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
```

La función de clasificación de Petrov se basa en el algoritmo publicado en "Classifying geometries in general relativity: III Classification in practice" de Pollney, Skea, and d'Inverno, Class. Quant. Grav. 17 2885-2902 (2000). Excepto para algunos ejemplos sencillos, esta implementación no ha sido exhaustivamente probada, por lo que puede contener errores.

26.2.6 Torsión y no metricidad

El paquete `ctensor` es capaz de calcular e incluir coeficientes de torsión y no metricidad en los coeficientes de conexión.

Los coeficientes de torsión se calculan a partir de un tensor suministrado por el usuario, `tr`, el cual debe ser de rango (2,1). A partir de ahí, los coeficientes de torsión `kt` se calculan de acuerdo con las siguientes fórmulas:

```
              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
```

Los coeficientes de no metricidad se calculan a partir de un vector de no metricidad, `nm`, suministrado por el usuario. A partir de ahí, los coeficientes de no metricidad, `nmc`, se calculan como se indica a continuación:

```
              k    k        km
        -nm  D  - D  nm  + g   nm  g
   k       i  j    i   j         m  ij
nmc  = ------------------------------
   ij               2
```

donde D es la delta de Kronecker.

`contortion` (*tr*) [Función]
Calcula los coeficientes (2,1) de contorsión del tensor de torsión *tr*.

`nonmetricity` (*nm*) [Función]
Calcula los coeficientes (2,1) de no metricidad del vector de no metricidad *nm*.

26.2.7 Otras funcionalidades

`ctransform` (*M*) [Función]
Es una función del paquete `ctensor`. Realiza una transformación de coordenadas a partir de una matriz cuadrada simétrica M arbitraria. El usuario debe introducir las funciones que definen la transformación.

`findde` (*A*, *n*) [Función]
Devuelve la lista de las ecuaciones diferenciales que corresponden a los elementos del arreglo cuadrado n-dimensional. El argumento n puede ser 2 ó 3; `deindex` es una lista global que contiene los índices de A que corresponden a estas ecuaciones diferenciales. Para el tensor de Einstein (`ein`), que es un arreglo bidimensional, si se calcula para la métrica del ejemplo de más abajo, `findde` devuelve las siguientes ecuaciones diferenciales independientes:

```
(%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` () [Función]

Calcula el gradiente covariante de una función escalar permitiendo al usuario elegir el nombre del vector correspondiente, tal como ilustra el ejemplo que acompaña a la definición de la función `contragrad`.

`contragrad` () [Function]

Calcula el gradiente contravariante de una función escalar permitiendo al usuario elegir el nombre del vector correspondiente, tal como muestra el siguiente ejemplo para la 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` () [Función]

Calcula el tensor de d'Alembertian de la función escalar una vez se han declarado las dependencias. Por ejemplo:

```
(%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` () [Función]

Calcula la divergencia covariante del tensor de segundo rango (mixed second rank tensor), cuyo primer índice debe ser covariante, devolviendo las `n` componentes correspondientes del campo vectorial (la divergencia), siendo `n` = `dim`.

`cgeodesic` (*dis*) [Función]

Es una función del paquete `ctensor` que calcula las ecuaciones geodésicas del movimiento para una métrica dada, las cuales se almacenan en el arreglo `geod[i]`. Si el argumento *dis* vale `true` entonces se muestran estas ecuaciones.

`bdvac` (*f*) [Función]

Genera las componentes covariantes de las ecuaciones del campo vacío de la teoría gravitacional de Brans- Dicke gravitational. El campo escalar se especifica con el argumento *f*, el cual debe ser el nombre de una función no evaluada (precedida de apóstrofo) con dependencias funcionales, por ejemplo, `'p(x)`.

Las componentes del tensor covariante (second rank covariant field tensor) se almacenan en el arreglo `bd`.

`invariant1` () [Función]
Genera el tensor de Euler-Lagrange (ecuaciones de campo) para la densidad invariante de R^2. Las ecuaciones de campo son las componentes del arreglo `inv1`.

26.2.8 Utilidades

`diagmatrixp` (*M*) [Función]
Devuelve `true` si *M* es una matriz diagonal o un arreglo bidimensional.

`symmetricp` (*M*) [Función]
Devuelve `true` si *M* es una matriz simétrica o un arreglo bidimensional.

`ntermst` (*f*) [Función]
Permite hacerse una idea del tamaño del tensor *f*.

`cdisplay` (*ten*) [Función]
Muestra todos los elementos del tensor *ten* como arreglo multidimensional. Tensors de rango 0 y 1, así como otros tipos de variables, se muestran como en `ldisplay`. Tensors de rango 2 se muestran como matrices bidimensionales, mientras que tensores de mayor rango se muestran como listas de matrices bidimensionales. Por ejemplo, el tensor de Riemann de la métrica de Schwarzschild se puede ver 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                                       ]
```

$$
\text{riem}_{2,2} = \begin{bmatrix} r\ (r - 2\,m) & 0 & 0 & 0 \\ 0 & 0 & -\dfrac{m}{r^2\ (r - 2\,m)} & 0 \\ 0 & 0 & 0 & -\dfrac{m}{r^2\ (r - 2\,m)} \end{bmatrix}
$$

$$
\text{riem}_{2,3} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & \dfrac{m}{r^2\ (r - 2\,m)} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}
$$

$$
\text{riem}_{2,4} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \dfrac{m}{r^2\ (r - 2\,m)} \\ 0 & 0 & 0 & 0 \end{bmatrix}
$$

$$
\text{riem}_{3,1} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ m & 0 & 0 & 0 \\ \dfrac{}{r} & & & \\ 0 & 0 & 0 & 0 \end{bmatrix}
$$

$$
\begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}
$$

```
                      [           ]
                      [ 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 ]
```

```
                                [                       ]
                                [ 0        0        0  0 ]
                   riem     = [                       ]
                       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*) [Función]

Devuelve una nueva lista consistente en *L* sin su *n*-ésimo elemento.

26.2.9 Variables utilizadas por `ctensor`

`dim` [Variable opcional]

Valor por defecto: 4

Es la dimensión de la variedad, que por defecto será 4. La instrucción `dim: n` establecerá la dimensión a cualquier otro valor `n`.

`diagmetric` [Variable opcional]

Valor por defecto: `false`

Si `diagmetric` vale `true` se utilizarán rutinas especiales para calcular todos los objetos geométricos teniendo en cuenta la diagonalidad de la métrica, lo que redundará en una reducción del tiempo de cálculo. Esta opción se fija automáticamente por `csetup` si se especifica una métrica diagonal.

`ctrgsimp` [Variable opcional]

Provoca que se realicen simplificaciones trigonométricas cuando se calculan tensores. La variable `ctrgsimp` afecta únicamente a aquellos cálculos que utilicen un sistema de referencia móvil.

`cframe_flag` [Variable opcional]

Provoca que los cálculos se realicen respecto de un sistema de referencia móvil.

`ctorsion_flag` [Variable opcional]

Obliga a que se calcule también el tensor de contorsión junto con los coeficientes de conexión. El propio tensor de contorsión se calcula con la función `contortion` a partir del tensor `tr` suministrado por el usuario.

`cnonmet_flag` [Variable opcional]

Obliga a que se calculen también los coeficientes de no metricidad junto con los coeficientes de conexión. Los coeficientes de no metricidad se calculan con la función `nonmetricity` a partir del vector de no metricidad`nm` suministrado por el usuario.

`ctayswitch` [Variable opcional]

Si vale `true`, obliga a que ciertos cálculos de `ctensor` se lleven a cabo utilizando desarrollos de series de Taylor. Estos cálculos hacen referencia a las funciones `christof`, `ricci`, `uricci`, `einstein` y `weyl`.

`ctayvar` [Variable opcional]

Variable utilizada para desarrollos de Taylor cuando la variable `ctayswitch` vale `true`.

`ctaypov` [Variable opcional]

Máximo exponente utilizado en los desarrollos de Taylor cuando `ctayswitch` vale `true`.

`ctaypt` [Variable opcional]

Punto alrededor del cual se realiza un desarrollo de Taylor cuando `ctayswitch` vale `true`.

`gdet` [Variable opcional]

Es el determinante del tensor métrico `lg`, calculado por `cmetric` cuando `cframe_flag` vale `false`.

`ratchristof` [Variable opcional]

Obliga a que la función `christof` aplique la simplificación racional.

`rateinstein` [Variable opcional]

Valor por defecto: `true`

Si vale `true` entonces se hará la simplificación racional en los componentes no nulos de los tensores de Einstein; si `ratfac` vale `true` entonces las componentes también serán factorizadas.

`ratriemann` [Variable opcional]

Valor por defecto: `true`

Es una de las variables que controlan la simplificación de los tensores de Riemann; si vale `true`, entonces se llevará a cabo la simplificación racional; si `ratfac` vale `true` entonces las componentes también serán factorizadas.

`ratweyl` [Variable opcional]

Valor por defecto: `true`

Si vale `true`, entonces la función `weyl` llevará a cabo la simplificación racional de los valores del tensor de Weyl. si `ratfac` vale `true` entonces las componentes también serán factorizadas.

`lfg` [Variable]

Es la covariante de la métrica del sistema de referencia. Por defecto, está inicializada al sistema de referencia tetradimensional de Lorentz con signatura (+,+,+,-). Se utiliza cuando `cframe_flag` vale `true`.

`ufg` [Variable]

Es la métrica del sistema de referencia inverso. La calcula `lfg` cuando `cmetric` es invocada tomando `cframe_flag` el valor `true`.

`riem` [Variable]

Es el tensor (3,1) de Riemann. Se calcula cuando se invoca la función `riemann`. Para información sobre el indexado, véase la descripción de `riemann`.

Si `cframe_flag` vale `true`, `riem` se calcula a partir del tensor covariante de Riemann `lriem`.

`lriem` [Variable]

Es el tensor covariante de Riemann. Lo calcula la función `lriemann`.

`uriem` [Variable]

Es el tensor contravariante de Riemann. Lo calcula la función `uriemann`.

`ric` [Variable]

Es el tensor de Ricci. Lo calcula la función `ricci`.

`uric` [Variable]

Es el tensor contravariante de Ricci. Lo calcula la función `uricci`.

`lg` [Variable]

Es el tensor métrico. Este tensor se debe especificar (como matriz cuadrada de orden `dim`) antes de que se hagan otros cálculos.

`ug` [Variable]
Es la inversa del tensor métrico. Lo calcula la función `cmetric`.

`weyl` [Variable]
Es el tensor de Weyl. Lo calcula la función `weyl`.

`fb` [Variable]
Son los coeficientes del sistema de referencia soporte, tal como los calcula `frame_bracket`.

`kinvariant` [Variable]
Es la invariante de Kretchmann, tal como la calcula la función `rinvariant`.

`np` [Variable]
Es la cuaterna nula de Newman-Penrose, tal como la calcula la función `nptetrad`.

`npi` [Variable]
Es la cuaterna nula "raised-index Newman-Penrose". Lo calcula la función `nptetrad`. Se define como `ug.np`. El producto `np.transpose(npi)` es constante:

```
(%i39) trigsimp(np.transpose(npi));
                                [  0   - 1  0  0 ]
                                [                ]
                                [ - 1   0   0  0 ]
(%o39)                          [                ]
                                [  0    0   0  1 ]
                                [                ]
                                [  0    0   1  0 ]
```

`tr` [Variable]
Tensor de rango 3 suministrado por el usuario y que representa una torsión. Lo utiliza la función `contortion`.

`kt` [Variable]
Es el tensor de contorsión, calculado a partir de `tr` por la función `contortion`.

`nm` [Variable]
Vector de no metricidad suministrado por el usuario. Lo utiliza la función `nonmetricity`.

`nmc` [Variable]
Son los coeficientes de no metricidad, calculados a partir de `nm` por la función `nonmetricity`.

`tensorkill` [Variable del sistema]
Variable que indica si el paquete de tensores se ha inicializado. Utilizada por `csetup` y reinicializada por `init_ctensor`.

`ct_coords` [Variable opcional]
Valor por defecto: `[]`

La variable `ct_coords` contiene una lista de coordenadas. Aunque se define normalmente cuando se llama a la función `csetup`, también se pueden redefinir las coordenadas con la asignación `ct_coords: [j1, j2, ..., jn]` donde `j` es el nuevo nombre de las coordenadas. Véase también `csetup`.

26.2.10 Nombres reservados

Los siguientes nombres se utilizan internamente en el paquete `ctensor` y no deberían redefinirse:

```
  Nombre       Descripción
  ---------------------------------------
  _lg()        Toma el valor lfg si se utiliza métrica del sistema de referencia,
               lg en otro caso
  _ug()        Toma el valor ufg si se utiliza métrica del sistema de referencia,
               ug en otro caso
  cleanup()    Elimina elementos de la lista deindex
  contract4()  Utilizada por psi()
  filemet()    Utilizada por csetup() cuando se lee la métrica desde un fichero
  findde1()    Utilizada por findde()
  findde2()    Utilizada por findde()
  findde3()    Utilizada por findde()
  kdelt()      Delta de Kronecker (no generalizada)
  newmet()     Utilizada por csetup() para establecer una métrica interactivamente
  setflags()   Utilizada por init_ctensor()
  readvalue()
  resimp()
  sermet()     Utilizada por csetup() para definir una métrica como se-
rie de Taylor
  txyzsum()
  tmetric()    Métrica del sistema de referencia, utilizada por cmetric()
               cuando cframe_flag:true
  triemann()   Tensor de Riemann en la base del sistema de referencia, se uti-
liza cuando
               cframe_flag:true
  tricci()     Tensor de Ricci en la base del sistema de referencia, se uti-
liza cuando
               cframe_flag:true
  trrc()       Coeficientes de rotación de Ricci, utilizada por christof()
  yesp()
```

27 atensor

27.1 Introducción a atensor

El paquete `atensor` contiene funciones para la manipulación algebraica de tensores. Para hacer uso de `atensor` es necesario cargarlo en memoria haciendo `load("atensor")`, seguido de una llamada a la función `init_atensor`.

La parte más importante de `atensor` es una batería de reglas de simplificación para el producto no conmutativo (`"."`). El paquete `atensor` reconoce algunos tipos de álgebras; las correspondientes reglas de simplificación se activan tan pronto como se hace una llamada a la función `init_atensor`.

Las capacidades de `atensor` se pueden demostrar definiendo el álgebra de cuaterniones como un álgebra de Clifford Cl(0,2) con una base de dos vectores. Las tres unidades imaginarias son los dos vectores de la base junto con su producto:

```
      i = v     j = v     k = v  . v
           1         2         1    2
```

Aunque el paquete `atensor` incluye su propia definición para el álgebra de cuaterniones, no se utiliza en el siguiente ejemplo, en el cual se construye la tabla de multiplicación como una 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             ]
```

El paquete `atensor` reconoce como vectores de la base símbolos indexados, donde el símbolo es el almacenado en `asymbol` y el índice va desde 1 hasta `adim`. Para símbolos indexados, y sólo para ellos, se evalúan las formas bilineales `sf`, `af` y `av`. La evaluación sustituye el valor de `aform[i,j]` en lugar de `fun(v[i],v[j])`, donde `v` representa el valor de `asymbol` y `fun` es `af` o `sf`; o sustituye `v[aform[i,j]]` en lugar de `av(v[i],v[j])`.

Huelga decir que las funciones `sf`, `af` y `av` pueden volver a definirse.

Cuando se carga el paquete `atensor` se hacen las siguientes asignaciones de variables:

```
dotscrules:true;
dotdistrib:true;
dotexptsimp:false;
```

Si se quiere experimentar con una álgebra no asociativa, también se puede igualar la variable `dotassoc` a `false`. En tal caso, sin embargo, `atensimp` no será siempre capaz de realizar las simplificaciones deseadas.

27.2 Funciones y variables para atensor

`init_atensor` (*alg_type*, *opt_dims*) [Función]
`init_atensor` (*alg_type*) [Función]

Inicializa el paquete `atensor` con el tipo de álgebra especificado, *alg_type*, que puede ser una de las siguientes:

`universal`: El álgebra universal no tiene reglas de conmutación.

`grassmann`: El álgebra de Grassman se define mediante la relación de conmutación `u.v+v.u=0`.

`clifford`: El álgebra de Clifford se define mediante la regla de conmutación `u.v+v.u=-2*sf(u,v)` donde `sf` es una función escalar simétrica. Para esta álgebra, *opt_dims* puede contener hasta tres enteros no negativos, que representan el número de dimensiones positivas, degeneradas y negativas, respectivamente, de esta álgebra. Si se suministran los valores de *opt_dims*, `atensor` configurará los valores de `adim` y `aform` de forma apropiada. En otro caso, `adim` tomará por defecto el valor 0 y `aform` no se definirá.

`symmetric`: El álgebra simétrica se define mediante la regla de conmutación `u.v-v.u=0`.

`symplectic`: El álgebra simpléctica se define mediante la regla de conmutación `u.v-v.u=2*af(u,v)`, donde `af` es una función escalar antisimétrica. Para el álgebra simpléctica, *opt_dims* puede contener hasta dos enteros no negativos, que representan las dimensiones no degeneradas y degeneradas, respectivamente. Si se suministran los valores de *opt_dims*, `atensor` configurará los valores de `adim` y `aform` de forma apropiada. En otro caso, `adim` tomará por defecto el valor 0 y `aform` no se definirá.

`lie_envelop`: El álgebra de la envolvente de Lie se define mediante la regla de conmutación `u.v-v.u=2*av(u,v)`, donde `av` es una función antisimétrica.

La función `init_atensor` también reconoce algunos tipos de álgebras predefinidas:

`complex` implementa el álgebra de números complejos como un álgebra de Clifford Cl(0,1). La llamada `init_atensor(complex)` equivale a `init_atensor(clifford,0,0,1)`.

`quaternion` implementa el álgebra de cuaterniones. La llamada `init_atensor(quaternion)` equivale a `init_atensor(clifford,0,0,2)`.

`pauli` implementa el álgebra de Pauli como un álgebra de Clifford Cl(3,0). La llamada `init_atensor(pauli)` equivale a `init_atensor(clifford,3)`.

`dirac` implementa el álgebra de Dirac como un álgebra de Clifford Cl(3,1). La llamada `init_atensor(dirac)` equivale a `init_atensor(clifford,3,0,1)`.

`atensimp` (*expr*) [Función]

Simplifica la expresión algebraica de un tensor *expr* de acuerdo con las reglas configuradas mediante una llamada a `init_atensor`. La simplificación incluye la aplicación recursiva de las reglas de conmutación y llamadas a `sf`, `af` y `av` siempre que sea posible. Se utiliza un algoritmo que asegure que la función termina siempre, incluso en el caso de expresiones complejas.

`alg_type` [Función]

Tipo de álgebra. Valores válidos son `universal`, `grassmann`, `clifford`, `symmetric`, `symplectic` y `lie_envelop`.

`adim` [Variable]

Valor por defecto: 0

La dimensión del álgebra. El paquete `atensor` utiliza el valor de `adim` para determinar si un objeto indexado es un vector válido para la base. Véase `abasep`.

`aform` [Variable]

Valor por defecto: `ident(3)`

Valores por defecto para las formas bilineales `sf`, `af` y `av`. El valor por defecto es la matriz identidad `ident(3)`.

`asymbol` [Variable]

Valor por defecto: `v`

Símbolo para los vectores base.

`sf` (*u*, *v*) [Función]

Una función escalar simétrica que se utiliza en relaciones de conmutación. La implementación por defecto analiza si los dos argumentos son vectores base mediante `abasep` y en tal caso sustituye el valor correspondiente de la matriz `aform`.

`af` (*u*, *v*) [Función]

Una función escalar antisimétrica que se utiliza en relaciones de conmutación. La implementación por defecto analiza si los dos argumentos son vectores base mediante `abasep` y en tal caso sustituye el valor correspondiente de la matriz `aform`.

`av` (*u*, *v*) [Función]

Una función antisimétrica que se utiliza en relaciones de conmutación. La implementación por defecto analiza si los dos argumentos son vectores base mediante `abasep` y en tal caso sustituye el valor correspondiente de la matriz `aform`.

Ejemplo:

```
(%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]);
                                      x
(%o5)                                  3
```

`abasep` (*v*) [Función]

Analiza si su argumento es un vector base en `atensor`. Esto es, si se trata de un símbolo indexado, siendo el símbolo el mismo que el valor de `asymbol` y si el índice tiene un valor numérico entre 1 y `adim`.

28 Sumas productos y series

28.1 Funciones y variables para sumas y productos

`bashindices` (*expr*) [Función]

Transforma la expresión *expr* dándole a cada sumatorio y producto un único índice. Esto le da a `changevar` mayor precisión cuando opera con sumas y productos. La forma del único índice es `j`*number*. La cantidad *number* se determina en función de `gensumnum`, valor que puede cambiar el usuario. Por ejemplo, haciendo `gensumnum:0$`.

`lsum` (*expr*, *x*, *L*) [Función]

Representa la suma de *expr* para cada elemento x en L.

Se retornará la forma nominal `'lsum` si el argumento L no es una lista.

Ejemplos:

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

`intosum` (*expr*) [Función]

Mueve los factores multiplicativos que están fuera de un sumatorio hacia dentro de éste. Si el índice del sumatorio aparece en la expresión exterior, entonces `intosum` busca un índice razonable, lo mismo que hace con `sumcontract`. Se trata de la operación contraria a extraer factores comunes de los sumatorios.

En algunos caos puede ser necesario hacer `scanmap (multthru,` *expr*`)` antes que `intosum`.

Ejemplo:

```
(%i1) sum(2*x^2*n^k, k , 0, inf);
                                 inf
                                 ====
                              2  \      k
(%o1)                        2 x  >    n
                                 /
                                 ====
                                 k = 0
```

```
(%i2) intosum(%);
                              inf
                              ====
                              \        k  2
(%o2)                          >    2 n  x
                              /
                              ====
                              k = 0
```

product (*expr*, *i*, *i_0*, *i_1*) [Función]

Representa el producto de los valores de `expr` según el índice *i* varía de *i_0* hasta *i_1*. La forma nominal `'product` se presenta en forma de letra pi mayúscula.

La función `product` evalúa *expr* y los límites inferior y superior, *i_0* y *i_1*, pero no evalúa el índice *i*.

Si la diferencia entre los límites superior e inferior es un número entero, la expresión *expr* se evalúa para cada valor del índice *i*, siendo el resultado un producto en forma explícita.

En caso contrario, el rango del índice no está definido, aplicándose entonces algunas reglas que permitan simplificar el producto. Cuando la variable global `simpproduct` valga `true`, se aplicarán reglas adicionales. En ciertos casos, la simplificación dará lugar a un resultado que ya no tenga el formato del producto; en caso contrario se devolverá una forma nominal `'product`.

Véanse también `nouns` y `evflag`.

Ejemplos:

```
(%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
```

`simpsum` [Variable opcional]

Valor por defecto: `false`

Si `simpsum` vale `true`, se simplifica el resultado de un sumatorio `sum`. Esta simplificación podrá producir en ocasiones una expresión compacta. Si `simpsum` vale `false` o si se utiliza la forma apostrofada `'sum`, el valor es una forma nominal que representa la notación sigma habitual en matemáticas.

`sum` (*expr*, *i*, *i_0*, *i_1*) [Función]

Representa la suma de los valores de `expr` según el índice *i* varía de *i_0* hasta *i_1*. La forma nominal `'sum` se presenta en forma de letra sigma mayúscula.

La función `sum` evalúa su sumando *expr* y los límites inferior y superior, *i_0* y *i_1*, pero no evalúa el índice *i*.

Si la diferencia entre los límites superior e inferior es un número entero, el sumando *expr* se evalúa para cada valor del índice *i*, siendo el resultado una suma en forma explícita.

En caso contrario, el rango del índice no está definido, aplicándose entonces algunas reglas que permitan simplificar la suma. Cuando la variable global `simpsum` valga `true`, se aplicarán reglas adicionales. En ciertos casos, la simplificación dará lugar a un resultado que ya no tenga el formato del sumatorio; en caso contrario se devolverá una forma nominal `'product`.

Cuando `cauchysum` vale `true`, el producto de sumatorios se expresa como un producto de Cauchy, en cuyo caso el índice del sumatorio interior es función del índice del exterior, en lugar de variar independientemente.

La variable global `genindex` guarda el prefijo alfabético a utilizar cuando sea necesario generar automáticamente el siguiente índice de sumatorio.

La variable global `gensumnum` guarda el sufijo numérico a utilizar cuando sea necesario generar automáticamente el siguiente índice de sumatorio. Si `gensumnum` vale `false`, un índice generado automáticamente constará sólo de `genindex`, sin sufijo numérico.

Véanse también `sumcontract`, `intosum`, `bashindices`, `niceindices`, `nouns` y `evflag`.

Ejemplos:

```
(%i1) sum (i^2, i, 1, 7);
```

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

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

`sumcontract` (*expr*) [Función]

Combina todos los sumatorios de una suma cuyos límites inferiores y superiores difieren por constantes. El resultado es una expresión que contiene un sumatorio por cada conjunto de tales sumatorios, más todos los demás términos adicionales que tuvieron que extraerse para formar la suma. La función `sumcontract` combina todos los sumatorios compatibles y utiliza uno de los índices de uno de los sumatorios si puede, si no formará un índice que sea razonable.

Puede ser necesario hacer `intosum (`*expr*`)` antes que `sumcontract`.

Ejemplo:

```
(%i1) 'sum(1/l,l,1,n)+'sum(k,k,1,n+2);
                          n        n + 2
                         ====     ====
                         \     1  \
(%o1)                     >    - +  >    k
                         /     l  /
                         ====     ====
                         l = 1    k = 1
(%i2) sumcontract(%);
                             n
                            ====
                            \          1
                  2 n +      >    (l + -) + 3
                            /          l
                            ====
                            l = 1
```

`sumexpand` [Variable opcional]

Valor por defecto: `false`

Si `sumexpand` vale `true`, productos de sumatorios y de sumatorios con exponentes se reducen a sumatorios anidados.

Véase también `cauchysum`.

Ejemplos:

```
(%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
```

28.2 Introducción a las series

Maxima dispone de las funciones `taylor` y `powerseries` para calcular las series de las funciones diferenciables. También tiene herramientas como `nusum` capaces de encontrar la expresión compacta de algunas series. Operaciones como la suma y la multiplicación operan de la forma habitual en el contexto de las series. Esta sección presenta las variables globales que controlan la expansión.

28.3 Funciones y variables para las series

`cauchysum` [Variable opcional]

Valor por defecto: `false`

Cuando se multiplican sumatorios infinitos, si `sumexpand` vale `true` y `cauchysum` vale `true`, entonces se utilizará el producto de Cauchy en lugar del usual. En el producto de Cauchy el índice de la suma interna es función del índice de la exterior en lugar de variar de forma independiente. Un ejemplo aclara esta idea:

```
(%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*) [Función]

Para cada función *f_i* de variable *x_i*, `deftaylor` define *expr_i* como una serie de Taylor alrededor de cero. La expresión *expr_i* será un polinomio en *x_i* o una suma; `deftaylor` admite también expresiones más generales.

La llamada `powerseries (f_i(x_i), x_i, 0)` devuelve la serie definida por `deftaylor`.

La función `deftaylor` evalúa sus argumentos y devuelve la lista de las funciones *f_1*, ..., *f_n*.

Ejemplo:

```
(%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 [Variable opcional]

Valor por defecto: `true`

Si `maxtayorder` vale `true`, entonces durante la manipulación algebraica de series truncadas de Taylor, la función `taylor` trata de retener tantos términos correctos como sea posible.

niceindices (*expr*) [Función]

Cambia las etiquetas de los índices de sumas y productos de *expr*. La función `niceindices` trata de cambiar cada índice al valor de `niceindicespref[1]`, a menos que esa etiqueta aparezca ya en el sumando o factor, en cuyo caso `niceindices` realiza intentos con los siguientes elementos de `niceindicespref`, hasta que encuentre una variable que que no esté en uso. Si todas las variables de la lista han

sido ya revisadas, se formarán nuevos ínices añadiendo números enteros al valor de `niceindicespref[1]`, como `i0`, `i1`, `i2`, ...

La función `niceindices` evalúa sus argumentos y devuelve una expresión.

Ejemplo:

```
(%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` [Variable opcional]

Valor por defecto: `[i, j, k, l, m, n]`

La variable `niceindicespref` es la lista de la que la función `niceindices` va tomando nombres de etiquetas para índices de sumatorios y productos.

En `niceindicespref` se guardan normalmente nombres de variables.

Ejemplo:

```
(%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*) [Función]

Calcula la suma hipergeométrica indefinida de *expr* con respecto a la variable *x* utilizando una procedimiento de decisión debido a R.W. Gosper. La expresión *expr* y el resultado deben poder ser escritos como productos de potencias enteras, factoriales, coeficientes binomiales y funciones racionales.

Los términos suma "definida" e "indefinida" se usan de forma análoga a integración "definida" e "indefinida". La suma indefinida significa dar un resultado simbólico.

Las funciones `nusum` y `unsum` disponen de cierta información sobre sumas y diferencias de productos finitos. Véase también `unsum`.

Ejemplos:

```
(%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*) [Función]

Devuelve la lista de todas las funciones racionales que tienen el desarrollo de Taylor dado, en las que la suma de los grados del numerador y denominador es menor o igual que el nivel de truncamiento de la serie de potencias.

La expresión *taylor_series* es una serie de Taylor univariante. Los argumentos *numer_deg_bound* y *denom_deg_bound* son enteros positivos que indican las cotas para numerador y denominador.

La expresión *taylor_series* también puede ser una serie de Laurent, y las cotas de los grados pueden ser `inf`. El grado total se define como *`numer_deg_bound`* + *`denom_deg_bound`*. La longitud de una serie de potencias se define como `"truncation level" + 1 - min(0, "order of series")`.

```
(%i1) taylor (1 + x + x^2 + x^3, x, 0, 3);
                              2    3
(%o1)/T/             1 + x + x  + x  + . . .
(%i2) pade (%, 1, 1);
                                1
(%o2)                       [- -----]
                              x - 1
(%i3) t: taylor(-(83787*x^10 - 45552*x^9 - 187296*x^8
                  + 387072*x^7 + 86016*x^6 - 1507328*x^5
                  + 1966080*x^4 + 4194304*x^3 - 25165824*x^2
                  + 67108864*x - 134217728)
       /134217728, x, 0, 10);
                    2    3       4       5       6        7
             x   3 x    x    15 x    23 x    21 x    189 x
(%o3)/T/ 1 - - + ---- - -- - ----- + ----- - ----- - ------
             2    16    32   1024    2048    32768   65536

                                  8         9          10
                            5853 x    2847 x    83787 x
                          + ------- + ------- - --------- + . . .
                            4194304   8388608   134217728
(%i4) pade (t, 4, 4);
(%o4)                          []
```

No hay ninguna función racional de grado 4 en numerador y denominador con este desarrollo en serie de potencias. Es necesario dar un número de grados al numerador y denominador cuya suma sea al menos el grado del desarrollo de la serie, a fin de disponer de un número suficiente de coeficientes desconocidos para calcular.

```
(%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)]
```

powerseries (*expr*, *x*, a) [Función]

Devuelve la forma general del desarrollo en serie de potencias de *expr* para la variable *x* alrededor del punto *a* (que puede ser inf, de infinito):

```
                inf
                ====
                \                n
                 >     b  (x - a)
                /       n
                ====
                n = 0
```

Si powerseries no es capaz de desarrollar *expr*, la función taylor puede calcular los primeros términos de la serie.

Si verbose vale true, powerseries va mostrando mensajes mientras progresa el cálculo.

```
(%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 [Variable opcional]

Valor por defecto: false

Si psexpand vale true, toda expresi'on racional se muestra completamente expandida. La variable ratexpand tiene el mismo efecto.

Si `psexpand` vale `false`, las expresines multivariantes se presentan tal como lo hace el paquete de funciones racionales.

Si `psexpand` vale `multi`, los términos de igual grado son agrupados.

`revert` (*expr*, *x*) [Función]
`revert2` (*expr*, *x*, *n*) [Función]

Estas funciones devuelven el recíproco de *expr* en forma de desarrollo de Taylor alrededor de cero respecto de la variable *x*. La función `revert` devuelve un polinomio de grado igual a la mayor potencia en *expr*. La función `revert2` devuelve un polinomio de grado *n*, el cual puede ser mayor, igual o menor que el grado de *expr*.

Para utilizar estas funciones es necesario cargarlas en memoria mediante `load ("revert")`.

Ejemplos:

```
(%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*) [Función]
`taylor` (*expr*, [*x_1*, *x_2*, ...], *a*, *n*) [Función]
`taylor` (*expr*, [*x*, *a*, *n*, *'asymp*]) [Función]
`taylor` (*expr*, [*x_1*, *x_2*, ...], [*a_1*, *a_2*, ...], [*n_1*, *n_2*, ...]) [Función]

`taylor` (*expr*, [*x_1*, *a_1*, *n_1*], [*x_2*, *a_2*, *n_2*], ...) [Función]

La llamada `taylor (expr, x, a, n)` expande la expresión *expr* en un desarrollo de Taylor o de Laurent respecto de la variable *x* alrededor del punto *a*, con términos hasta `(x - a)^n`.

Si *expr* es de la forma `f(x)/g(x)` y `g(x)` no tiene términos hasta de grado *n*, entonces `taylor` intenta expandir `g(x)` hasta el grado `2 n`. Si aún así no hay términos no nulos, `taylor` dobla el grado de la expansión de `g(x)` hasta que el grado de la expansión sea menor o igual que `n 2^taylordepth`.

La llamada `taylor (expr, [x_1, x_2, ...], a, n)` devuelve la serie en potencias truncada de grado *n* en todas las variables *x_1*, *x_2*, ... alrededor del punto `(a, a, ...)`.

La llamada `taylor (expr, [x_1, a_1, n_1], [x_2, a_2, n_2], ...)` devuelve la serie en potencias truncada en las variables *x_1*, *x_2*, ... alrededor del punto `(a_1, a_2, ...)`; el truncamiento se realiza, respectivamente, en los grados *n_1*, *n_2*, ...

La llamada `taylor (expr, [x_1, x_2, ...], [a_1, a_2, ...], [n_1, n_2, ...])` devuelve la serie en potencias truncada en las variables *x_1*, *x_2*, ... alrededor del punto `(a_1, a_2, ...)`, el truncamiento se realiza, respectivamente, en los grados *n_1*, *n_2*, ...

La llamada `taylor (expr, [x, a, n, 'asymp])` devuelve el desarrollo de *expr* en potencias negativas de `x - a`. El término de mayor orden es `(x - a)^-n`.

Si `maxtayorder` vale `true`, entonces durante la manipulación algebraica de las series (truncadas) de Taylor, la función `taylor` intenta mantener tantos términos correctos como sea posible.

Si `psexpand` vale `true`, una expresión racional desarrollada se muestra completamente expandida. La variable `ratexpand` tiene el mismo efecto. Si `psexpand` vale `false`, una expresión multivariante se mostrará tal como lo hace el paquete de funciones racionales. Si `psexpand` vale `multi`, los términos del mismo grado son agrupados.

Véase también la variable `taylor_logexpand` para el control del desarrollo.

Ejemplos:

```
(%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` [Variable opcional]

Valor por defecto: 3

Si todavía no hay términos no nulos, la función `taylor` dobla el grado del desarrollo de *g*(*x*) tantas veces como sea necesario para que el grado del desarrollo sea menor o igual que *n* `2^taylordepth`.

`taylorinfo` (*expr*) [Función]

Devuelve información sobre el desarrollo de Taylor *expr*. El valor devuelto por esta función es una lista de listas. Cada lista contiene el nombre de una variable, el punto de expansión y el grado del desarrollo.

La función `taylorinfo` devuelve `false` si *expr* no es un desarrollo de Taylor.

Ejemplo:

```
(%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*) [Función]

Devuelve `true` si *expr* es un desarrollo de Taylor y `false` en caso contrario.

`taylor_logexpand` [Variable opcional]

Valor por defecto: `true`

La variable `taylor_logexpand` controla los desarrollos de logaritmos en la función `taylor`.

Si `taylor_logexpand` vale `true`, todos los logaritmos se expanden completamente de manera que algunos problemas que se plantean debido a ciertas identidades logarítmicas no interfieran con el proceso del cálculo del desarrollo de Taylor. Sin embargo, este proceder no es del todo correcto.

`taylor_order_coefficients` [Variable opcional]

Valor por defecto: `true`

La variable `taylor_order_coefficients` controla la ordenación de los coeficientes en un desarrollo de Taylor.

Si `taylor_order_coefficients` vale `true`, los coeficientes del desarrollo de Taylor se ordenan de la forma canónica.

`taylor_simplifier` (*expr*) [Función]

Simplifica los coeficientes de la serie de potencias *expr*. Esta función es llamada desde la función `taylor`.

`taylor_truncate_polynomials` [Variable opcional]

Valor por defecto: `true`

Si `taylor_truncate_polynomials` vale `true`, los polinomios quedan truncados en base a los niveles de truncamiento de entrada.

En otro caso, aquellos polinomios que se utilicen como entrada a la función `taylor` se consideran que tienen precisión infinita.

`taytorat (`*expr*`)` [Función]

Convierte *expr* del formato de `taylor` al formato CRE (Canonical Rational Expression). El efecto es el mismo que haciendo `rat (ratdisrep (`*expr*`))`, pero más rápido.

`trunc (`*expr*`)` [Función]

Devuelve la representación interna de la expresión *expr* de tal forma como si sus sumas fuesen una serie truncada de Taylor. La expresión *expr* no sufre ninguna otra modificación.

Ejemplo:

```
(%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*`)` [Función]

Devuelve la diferencia `f(`*n*`) - f(`*n* `- 1)`. En cierto sentido `unsum` es la inversa de `sum`.

Véase también `nusum`.

Ejemplos:

```
(%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` [Variable opcional]

Valor por defecto: `false`

Si `verbose` vale `true`, la función `powerseries` va imprimiendo mensajes durante su ejecución.

28.4 Introducción a las series de Fourier

El paquete `fourie` contiene funciones para el cálculo simbólico de series de Fourier. Hay funciones en el paquete `fourie` para calcular los coeficientes y para manipular las expresiones.

28.5 Funciones y variables para series de Fourier

`equalp` (*x*, *y*) [Función]

Devuelve `true` si `equal (x, y)`, en otro caso devuelve `false`. No devuelve el mensaje de error que se obtiene de `equal (x, y)` en un caso como éste.

`remfun` (*f*, *expr*) [Función]
`remfun` (*f*, *expr*, *x*) [Función]

La llamada `remfun (f, expr)` reemplaza todas las subexpresiones `f (arg)` por *arg* en *expr*.

La llamada `remfun (f, expr, x)` reemplaza todas las subexpresiones `f (arg)` por *arg* en *expr* sólo si *arg* contiene a la variable *x*.

`funp` (*f*, *expr*) [Función]
`funp` (*f*, *expr*, *x*) [Función]

La llamada `funp (f, expr)` devuelve `true` si *expr* contiene la función *f*.

La llamada `funp (f, expr, x)` devuelve `true` si *expr* contiene la función *f* y la variable *x* está presente en el argumento de alguna de las presencias de *f*.

`absint` (*f*, *x*, *halfplane*) [Función]
`absint` (*f*, *x*) [Función]
`absint` (*f*, *x*, *a*, *b*) [Función]

La llamada `absint (f, x, halfplane)` devuelve la integral indefinida de *f* con respecto a *x* en el semiplano dado (`pos`, `neg` o `both`). La función *f* puede contener expresiones de la forma `abs (x)`, `abs (sin (x))`, `abs (a) * exp (-abs (b) * abs (x))`.

La llamada `absint (f, x)` equivale a `absint (f, x, pos)`.

La llamada `absint (f, x, a, b)` devuelve la integral definida de *f* con respecto a *x* de *a* a *b*.

`fourier` (*f*, *x*, *p*) [Función]

Devuelve una lista con los coeficientes de Fourier de `f(x)` definida en el intervalo `[-p, p]`.

`foursimp (l)` [Función]
Simplifica `sin (n %pi)` a 0 si `sinnpiflag` vale `true` y `cos (n %pi)` a `(-1)^n` si `cosnpiflag` vale `true`.

`sinnpiflag` [Variable opcional]
Valor por defecto: `true`
Véase `foursimp`.

`cosnpiflag` [Variable opcional]
Valor por defecto: `true`
Véase `foursimp`.

`fourexpand (l, x, p, limit)` [Función]
Calcula y devuelve la serie de Fourier a partir de la lista de los coeficientes de Fourier *l* hasta el término *limit* (*limit* puede ser `inf`). Los argumentos *x* y *p* tienen el mismo significado que en `fourier`.

`fourcos (f, x, p)` [Función]
Devuelve los coeficientes de los cosenos de Fourier de `f(x)` definida en `[0, p]`.

`foursin (f, x, p)` [Función]
Devuelve los coeficientes de los senos de Fourier de `f(x)` definida en `[0, p]`.

`totalfourier (f, x, p)` [Función]
Devuelve `fourexpand (foursimp (fourier (f, x, p)), x, p, 'inf)`.

`fourint (f, x)` [Función]
Calcula y devuelve la lista de los coeficientes integrales de Fourier de `f(x)` definida en `[minf, inf]`.

`fourintcos (f, x)` [Función]
Devuelve los coeficientes integrales de los cosenos `f(x)` en `[0, inf]`.

`fourintsin (f, x)` [Función]
Devuelve los coeficientes integrales de los senos `f(x)` en `[0, inf]`.

28.6 Funciones y variables para series de Poisson

`intopois (a)` [Función]
Convierte *a* en un codificado Poisson.

`outofpois (a)` [Función]
Convierte *a* desde codificado de Poisson a una representación general. Si *a* no está en forma de Poisson, `outofpois` hace la conversión, siendo entonces el valor retornado `outofpois (intopois (a))`. Esta función es un simplificador canónico para sumas de potencias de senos y cosenos.

`poisdiff (a, b)` [Función]
Deriva *a* con respecto a *b*. El argumento *b* debe aparecer sólo en los argumentos trigonométricos o sólo en los coeficientes.

`poisexpt (a, b)` [Función]
Idéntico a `intopois (a^b)`. El argumento *b* debe ser un entero positivo.

`poisint (a, b)` [Función]
Integra en un sentido restringido similar a `poisdiff`.

`poislim` [Variable optativa]
Valor por defecto: 5

La variable `poislim` determina el dominio de los coeficientes en los argumentos de las funciones trigonométricas. El valor por defecto 5 corresponde al intervalo [-2^(5-1)+1,2^(5-1)], o [-15,16], pero puede reasignarse para [-2^(n-1)+1, 2^(n-1)].

`poismap (series, sinfn, cosfn)` [Función]
Aplica las funciones *sinfn* a los términos sinusoidales y las funciones *cosfn* a los cosenoidales de la serie de Poisson dada. Tanto *sinfn* como *cosfn* son funciones de dos argumentos, los cuales son un coeficiente y una parte trigonométrica de un término de la serie.

`poisplus (a, b)` [Función]
Idéntico a `intopois (a + b)`.

`poissimp (a)` [Función]
Convierte *a* en una serie de Poisson para *a* en su representación general.

`poisson` [Símbolo especial]
El símbolo `/P/` sigue a la etiqueta de las líneas que contienen expresiones que son series de Poisson.

`poissubst (a, b, c)` [Función]
Sustituye *b* por *a* en *c*, donde *c* es una serie de Poisson.

(1) Si *b* es una de las variables *u*, *v*, *w*, *x*, *y* o *z*, entonces *a* debe ser una expresión lineal en esas variables (por ejemplo, `6*u + 4*v`).

(2) Si *b* no es ninguna de esas variables, entonces *a* no puede contener tampoco a ninguna de ellas, ni senos, ni cosenos.

`poistimes (a, b)` [Función]
Idéntico a `intopois (a*b)`.

`printpois (a)` [Función]
Presenta una serie de Poisson en un formato legible. Conjuntamente con `outofpois`, si es necesario convertirá *a* primero en una codificación de Poisson.

29 Teoría de Números

29.1 Funciones y variables para teoría de números

`bern` (*n*) [Función]

Devuelve el *n*-ésimo número de Bernoulli del entero *n*. Los números de Bernoulli iguales a cero son suprimidos si `zerobern` vale `false`.

Véase también `burn`.

```
(%i1) zerobern: true$
(%i2) map (bern, [0, 1, 2, 3, 4, 5, 6, 7, 8]);
                      1  1       1      1        1
(%o2)          [1, - -, -, 0, - --, 0, --, 0, - --]
                      2  6       30     42       30
(%i3) zerobern: false$
(%i4) map (bern, [0, 1, 2, 3, 4, 5, 6, 7, 8]);
                      1  1    1   1     1   5     691   7
(%o4)          [1, - -, -, - --, --, - --, --, - ----, -]
                      2  6    30  42    30  66    2730  6
```

`bernpoly` (*x*, *n*) [Función]

Devuelve el *n*-ésimo polinomio de Bernoulli de variable *x*.

`bfzeta` (*s*, *n*) [Función]

Devuelve la función zeta de Riemann para el argumento *s*. El valor que devuelve es del tipo "big float" (bfloat) y *n* es su número de dígitos.

Es necesario cargar en memoria esta función haciendo `load ("bffac")`.

`bfhzeta` (*s*, *h*, *n*) [Función]

Devuelve la función zeta de Hurwitz para los argumentos *s* y *h*. El valor que devuelve es del tipo "big float" (bfloat) y *n* es su número de dígitos.

La función zeta de Hurwitz se define como

$$\zeta(s,h) = \sum_{k=0}^{\infty} \frac{1}{(k+h)^s}$$

Ejecútese `load ("bffac")` antes de hacer uso de esta función.

`burn` (*n*) [Función]

Siendo *n* entero, Devuelve un número racional que aproxima el *n*-ésimo número de Bernoulli. La función `burn` aprovecha el hecho de que los números de Bernoulli racionales se pueden aproximar con notable precisión gracias a

```
                    n - 1  1 - 2 n
             (- 1)      2        zeta(2 n) (2 n)!
B(2 n) = ------------------------------------
                                2 n
                             %pi
```

La función `burn` puede ser más eficiente que `bern` cuando n es un número grande, ya que `bern` calcula todos los números de Bernoulli hasta el n-ésimo. Por el contrario, `burn` hace uso de la aproximación para enteros pares n > 255. En caso de enteros impares y n <= 255, se hace uso de la función `bern`.

Para utilizar esta función hay que cargarla antes en memoria escribiendo `load ("bffac")`. Véase también `bern`.

`chinese` ([*r_1*, ..., *r_n*], [*m_1*, ..., *m_n*]) [Función]

Resulve el sistema de congruencias `x = r_1 mod m_1`, ..., `x = r_n mod m_n`. Los restos *r_n* pueden ser enteros arbitrarios, mientras que los módulos *m_n* deben ser positivos y primos dos a dos.

```
(%i1) mods : [1000, 1001, 1003, 1007];
(%o1)                    [1000, 1001, 1003, 1007]
(%i2) lreduce('gcd, mods);
(%o2)                                1
(%i3) x : random(apply("*", mods));
(%o3)                          685124877004
(%i4) rems : map(lambda([z], mod(x, z)), mods);
(%o4)                        [4, 568, 54, 624]
(%i5) chinese(rems, mods);
(%o5)                          685124877004
(%i6) chinese([1, 2], [3, n]);
(%o6)                     chinese([1, 2], [3, n])
(%i7) %, n = 4;
(%o7)                               10
```

`cf` (*expr*) [Función]

Calcula aproximaciones con fracciones continuas. *expr* es una expresión que contiene fracciones continuas, raíces cuadradas de enteros, y números reales (enteros, racionales, decimales en coma flotante y decimales de precisión arbitraria). `cf` calcula expansiones exactas de números racionales, pero las expansiones de nÃºmeros decimales de coma flotante se truncan de acuerdo con el valor de `ratepsilon`, y la de los de decimales de precisión arbitraria (bigfloats) lo hacen respecto de `10^(-fpprec)`.

En las expresiones se pueden combinar operandos con operadores aritméticos. Maxima no conoce operaciones con fracciones continuas más allá de la función `cf`.

La función `cf` evalúa sus argumentos después de asignar a la variable `listarith` el valor `false`, retornando una fracción continua en forma de lista.

Una fracción continua `a + 1/(b + 1/(c + ...))` se representa como la lista `[a, b, c, ...]`, donde los elementos `a`, `b`, `c`, ... se evalúan como enteros. La expresión *expr* puede contener también `sqrt (n)` donde `n` es un entero; en tal caso, `cf` devolverá tantos términos de la fracción continua como indique el valor de la variable `cflength` multiplicado por el período.

Una fracción continua puede reducirse a un número evaluando la representación aritmética que devuelve `cfdisrep`. Véase también `cfexpand`, que es otra alternativa para evaluar fracciones continuas.

Véanse asimismo `cfdisrep`, `cfexpand` y `cflength`.

Ejemplos:

- La expresión *expr* contiene fracciones continuas y raíces cuadradas de enteros.

```
(%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]
```

- La variable `cflength` controla cuantos períodos de la fracción continua se calculan para números irracionales algebraicos.

```
(%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]
```

- Una fracción continua puede calcularse evaluando la representación aritmética que devuelve `cfdisrep`.

```
(%i1) cflength: 3$
(%i2) cfdisrep (cf (sqrt (3)))$
(%i3) ev (%, numer);
(%o3)                   1.731707317073171
```

- Maxima no sabe sobre operaciones con fracciones continuas más de lo que aporta la función `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` (*lista*) [Función]

Construye y devuelve una expresión aritmética ordinaria de la forma `a + 1/(b + 1/(c + ...))` a partir de la representación en formato lista de la fracción continua `[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*) [Función]

Devuelve la matriz con los numeradores y denominadores de la última (columna 1) y penúltima (columna 2) convergentes de la fracción continua *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` [Variable opcional]

Valor por defecto: 1

La variable `cflength` controla el número de términos de la fracción continua que devuelve la función `cf`, que será `cflength` multiplicado por el período. Así, el valor por defecto será el de un 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*) [Función]

`divsum` (*n*) [Función]

La llamada `divsum (n, k)` devuelve la suma de los divisores de *n* elevados a la *k*-ésima potencia.

La llamada `divsum (n)` devuelve la suma de los 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*) [Función]

Devuelve el *n*-ésimo número de Euler del entero no negativo *n*. Los número de Euler iguales a cero se eliminan si `zerobern` vale `false`.

Para la constante de Euler-Mascheroni, véase `%gamma`.

```
(%i1) zerobern: true$
(%i2) map (euler, [0, 1, 2, 3, 4, 5, 6]);
(%o2)               [1, 0, - 1, 0, 5, 0, - 61]
(%i3) zerobern: false$
(%i4) map (euler, [0, 1, 2, 3, 4, 5, 6]);
(%o4)               [1, - 1, 5, - 61, 1385, - 50521, 2702765]
```

`factors_only` [Variable opcional]

Valor por defecto: `false`

Controla el resultado devuelto por `ifactors`. El valor por defecto `false` hace que `ifactors` no dé información sobre las multiplicidades de los factores primos calculados. Cuando `factors_only` vale `true`, `ifactors` solo devuelve la lista de factores primos.

Para ejemplos, véase `ifactors`.

`fib` (*n*) [Función]

Devuelve el *n*-ésimo número de Fibonacci. La llamada `fib(0)` devuelve 0, `fib(1)` devuelve 1 y `fib (-`*n*`)` es igual a `(-1)^(`*n* `+ 1) * fib(`*n*`)`.

Después de llamar a `fib`, la variable `prevfib` toma el valor `fib (`*n* `- 1)`, que es el número de Fibonacci que precede al último calculado.

```
(%i1) map (fib, [-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8]);
(%o1)           [- 3, 2, - 1, 1, 0, 1, 1, 2, 3, 5, 8, 13, 21]
```

`fibtophi` (*expr*) [Función]

Expresa los números de Fibonacci en *expr* en términos de la razón áurea `%phi`, que es `(1 + sqrt(5))/2`, aproximadamente 1.61803399.

Ejemplos:

```
(%i1) fibtophi (fib (n));
                            n             n
                         %phi  - (1 - %phi)
(%o1)                    ------------------
                             2 %phi - 1
(%i2) fib (n-1) + fib (n) - fib (n+1);
(%o2)          - fib(n + 1) + fib(n) + fib(n - 1)
(%i3) fibtophi (%);
            n + 1             n + 1       n             n
          %phi      - (1 - %phi)        %phi  - (1 - %phi)
(%o3) - -------------------------- + ------------------
                2 %phi - 1                2 %phi - 1
                                          n - 1             n - 1
                                        %phi      - (1 - %phi)
                                      + --------------------------
                                                2 %phi - 1
(%i4) ratsimp (%);
(%o4)                           0
```

`ifactors` (*n*) [Función]

Devuelve la factorización del entero positivo *n*. Si `n=p1^e1..pk^nk` es la descomposición de *n* en números primos, `ifactors` devuelve `[[p1, e1], ... , [pk, ek]]`.

Los métodos de factorización se basan en divisiones tentativas con números primos hasta 9973, en los métodos rho y p-1 de Pollard y en curvas elípticas.

La respuesta que se obtiene de `ifactors` está controlada por la variable opcional `factors_only`. El valor por defecto `false` hace que `ifactors` no dé información sobre las multiplicidades de los factores primos calculados. Cuando `factors_only` vale `true`, `ifactors` solo devuelve la lista de factores primos.

```
(%i1) ifactors(51575319651600);
(%o1)     [[2, 4], [3, 2], [5, 2], [1583, 1], [9050207, 1]]
(%i2) apply("*", map(lambda([u], u[1]^u[2]), %));
(%o2)                          51575319651600
(%i3) ifactors(51575319651600), factors_only : true;
(%o3)                    [2, 3, 5, 1583, 9050207]
```

`igcdex` (*n*, *k*) [Función]

Devuelve la lista `[a, b, u]`, donde *u* es el máximo común divisor de *n* y *k*, siendo *u* igual a `a n + b k`. Los argumentos *n* y *k* deben ser enteros.

`igcdex` implementa el algoritmo de Euclides. Véase también `gcdex`.

La instrucción `load("gcdex")` carga esta función.

Ejemplos:

```
(%i1) load("gcdex")$

(%i2) igcdex(30,18);
(%o2)                         [- 1, 2, 6]
(%i3) igcdex(1526757668, 7835626735736);
(%o3)             [845922341123, - 164826435, 4]
(%i4) igcdex(fib(20), fib(21));
(%o4)                     [4181, - 2584, 1]
```

`inrt` (*x*, *n*) [Función]

Devuelve la raíz entera *n*-ésima del 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*) [Función]

Calcula el inverso de *n* módulo *m*. La llamada `inv_mod (n,m)` devuelve `false` si *n* es un divisor nulo módulo *m*.

```
(%i1) inv_mod(3, 41);
(%o1)                            14
(%i2) ratsimp(3^-1), modulus = 41;
(%o2)                            14
(%i3) inv_mod(3, 42);
(%o3)                           false
```

`isqrt (x)` [Función]
Devuelve la "raíz cuadrada entera" del valor absoluto de *x*, el cual debe ser un entero.

`jacobi (p, q)` [Función]
Devuelve el símbolo de Jacobi para *p* y *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)` [Función]
Devuelve el mínimo común múltiplo de sus argumentos. Los argumentos pueden ser tanto expresiones en general como enteros.

Es necesario cargar en memoria esta función haciendo `load ("functs")`.

`lucas (n)` [Función]
Devuelve el *n*-ésimo número de Lucas. `lucas(0)` es igual a 2, `lucas(1)` es igual a 1 y `lucas(-n)` es igual a `(-1)^(-n) * lucas(n)`.

```
(%i1) map (lucas, [-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8]);
(%o1)             [7, - 4, 3, - 1, 2, 1, 3, 4, 7, 11, 18, 29, 47]
```

Después de llamar a `lucas`, la variable global `next_lucas` es igual a `lucas (n + 1)`, el número de Lucas que sigue al último que se ha devuelto. El ejemplo muestra como los números de Fibonacci se pueden calcular mediante `lucas` y `next_lucas`.

```
(%i1) fib_via_lucas(n) :=
         block([lucas : lucas(n)],
         signum(n) * (2*next_lucas - lucas)/5 )$
(%i2) map (fib_via_lucas, [-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8]);
(%o2)             [- 3, 2, - 1, 1, 0, 1, 1, 2, 3, 5, 8, 13, 21]
```

`mod (x, y)` [Función]
Si *x* e *y* son números reales e *y* es distinto de cero, devuelve `x - y * floor(x / y)`. Para todos los reales *x*, se tiene `mod (x, 0) = x`. Para información sobre la definición de `mod (x, 0) = x`, véase la sección 3.4 de "Concrete Mathematics", by Graham, Knuth, and Patashnik. La función `mod (x, 1)` es de diente de sierra con periodo unidad y con `mod (1, 1) = 0` y `mod (0, 1) = 0`.

Para encontrar el argumento principal (un número del intervalo `(-%pi, %pi]`) de un número complejo, hágase uso de la función `x |-> %pi - mod (%pi - x, 2*%pi)`, donde *x* es un argumento.

Si *x* e *y* son expresiones constantes (por ejemplo, `10 * %pi`), `mod` utiliza el mismo esquema de evaluación basado en números grandes en coma flotante (big floats) que `floor` y `ceiling`. También es posible, pero improbable, que `mod` pueda retornar un valor erróneo en tales casos.

Para argumentos no numéricos *x* o *y*, `mod` aplica algunas reglas de simplificación:

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

```
(%i3) mod (0, x);
(%o3)                          0
```

next_prime (*n*) [Función]

Devuelve el menor de los primos mayores que *n*.

```
(%i1) next_prime(27);
(%o1)                         29
```

partfrac (*expr*, *var*) [Función]

Expande la expresión *expr* en fracciones parciales respecto de la variable principal *var*. La función `partfrac` hace una descomposición completa en fracciones parciales. El algoritmo que se utiliza se basa en el hecho de que los denominadores de la expansión en fracciones parciales (los factores del denominador original) son primos relativos. Los numeradores se pueden escribir como combinaciones lineales de los denominadores.

```
(%i1) 1/(1+x)^2 - 2/(1+x) + 2/(2+x);
                        2       2        1
(%o1)                -------- - ----- + -------
                     x + 2   x + 1          2
                                     (x + 1)
(%i2) ratsimp (%);
                                  x
(%o2)                  - ------------------
                            3      2
                           x  + 4 x  + 5 x + 2
(%i3) partfrac (%, x);
                        2       2        1
(%o3)                -------- - ----- + -------
                     x + 2   x + 1          2
                                     (x + 1)
```

power_mod (a, *n*, *m*) [Función]

Utiliza un algoritmo modular para calcular `a^n mod m`, siendo *a* y *n* enteros cualesquiera y *m* un entero positivo. Si *n* es negativo, se utilizará `inv_mod` para encontrar el inverso modular.

```
(%i1) power_mod(3, 15, 5);
(%o1)                          2
(%i2) mod(3^15,5);
(%o2)                          2
(%i3) power_mod(2, -1, 5);
(%o3)                          3
(%i4) inv_mod(2,5);
(%o4)                          3
```

primep (*n*) [Función]

Comprueba si el número entero *n* es o no primo, devolviendo `true` o `false` según el caso.

Cuando el resultado de `primep (n)` es `false`, *n* es un número compuesto, y si es `true`, *n* es primo con alta probabilidad.

Si n es menor que 3317044064679887385961981, se utiliza una versión determinística de la prueba de Miller-Rabin. En tal caso, si `primep (n)` devuelve `true`, entonces n es un número primo.

Para n mayor que 3317044064679887385961981 `primep` realiza `primep_number_of_tests` pruebas de seudo-primalidad de Miller-Rabin y una prueba de seudo-primalidad de Lucas. La probabilidad de que un número compuesto n pase una prueba de Miller-Rabin es menor que 1/4. Con el valor por defecto de `primep_number_of_tests`, que es 25, la probabilidad de que n sea compuesto es menor que 10^-15.

`primep_number_of_tests` [Variable opcional]

Valor por defecto: 25

Número de pruebas de Miller-Rabin a realizar por `primep`.

`prev_prime (n)` [Función]

Devuelve el mayor de los primos menores que n.

```
(%i1) prev_prime(27);
(%o1)                        23
```

`qunit (n)` [Función]

Devuelve la unidad principal de `sqrt (n)`, siendo n un entero; consiste en la resolución de la ecuación de Pell `a^2 - n b^2 = 1`.

```
(%i1) qunit (17);
(%o1)                      sqrt(17) + 4
(%i2) expand (% * (sqrt(17) - 4));
(%o2)                            1
```

`totient (n)` [Función]

Devuelve el número de enteros menores o iguales a n que son primos relativos con n.

`zerobern` [Variable opcional]

Valor por defecto: `true`

Si `zerobern` vale `false`, `bern` excluye los números de Bernoulli y `euler` excluye los números de Euler que sean iguales a cero. Véase `bern` y `euler`.

`zeta (n)` [Función]

Devuelve la función zeta de Riemann. Si n es entero negativo, 0 o número par positivo, la función zeta de Riemann devuelve un valor exacto; en el caso de número par positivo, la variable opcional `zeta%pi`, además, tiene que tener el valor `true` (véase `zeta%pi`). Cuando el argumento es un número decimal o bigfloat, entonces la función zeta de Riemann se calcula numéricamente. Maxima devuelve una forma nominal `zeta (n)` para cualesquiera otros argumentos, incluidos los racionales no enteros, los números complejos y los enteros pares si `zeta%pi` vale `false`.

`zeta(1)` no está definida, pero Maxima conce el límite de `limit(zeta(x), x, 1)` por ambos lados.

La función zeta de Riemann se distribuye sobre las listas, matrices y ecuaciones.

Véanse también `bfzeta` y `zeta%pi`.

Ejemplos:

```
(%i1) zeta([-2,-1,0,0.5,2,3,1+%i]);
                                                    2
             1     1                           %pi
(%o1)  [0, - --, - -, - 1.460354508809587, ----, zeta(3), zeta(%i + 1)]
             12    2                            6

(%i2) limit(zeta(x),x,1,plus);
(%o2)                                  inf
(%i3) limit(zeta(x),x,1,minus);
(%o3)                                 minf
```

`zeta%pi` [Variable opcional]

Valor por defecto: `true`

Si `zeta%pi` vale `true`, `zeta` devuelve una expresión proporcional a `%pi^n` si `n` es un número par positivo. En caso contrario, `zeta` no se evalúa y devuelve la forma nominal `zeta (n)`.

Ejemplos:

```
(%i1) zeta%pi: true$
(%i2) zeta (4);
                                    4
                                 %pi
(%o2)                            ----
                                  90
(%i3) zeta%pi: false$
(%i4) zeta (4);
(%o4)                          zeta(4)
```

`zn_add_table` (*n*) [Función]

Muestra la tabla de la suma de todos los elementos de (Z/*n*Z).

Véanse también `zn_mult_table` y `zn_power_table`.

`zn_determinant` (*matrix*, *p*) [Función]

Utiliza el procedimiento de la descomposición LU para calcular el determinante de *matrix* sobre (Z/*p*Z). El argumento *p* debe ser un número primo.

Si el determinante es igual a cero, el procedimiento puede fallar, en cuyo caso `zn_determinant` calcula el determinante no modular y luego reduce.

Véase también `zn_invert_by_lu`.

Ejemplo:

```
(%i1) m : matrix([1,3],[2,4]);
                                  [ 1  3 ]
(%o1)                             [      ]
                                  [ 2  4 ]
(%i2) zn_determinant(m, 5);
(%o2)                                3
(%i3) m : matrix([2,4,1],[3,1,4],[4,3,2]);
```

```
                                    [ 2  4  1 ]
                                    [         ]
(%o3)                               [ 3  1  4 ]
                                    [         ]
                                    [ 4  3  2 ]
(%i4) zn_determinant(m, 5);
(%o4)                                  0
```

zn_invert_by_lu (*matrix*, *p*) [Función]

Utiliza el procedimiento de la descomposición LU para calcular la inversa modular de *matrix* sobre (Z/*p*Z). El argumento *p* debe ser un número primo y *matrix* invertible. La función `zn_invert_by_lu` devuelve `false` si *matrix* no es invertible.

Véase `zn_determinant`.

Ejemplo:

```
(%i1) m : matrix([1,3],[2,4]);
                                    [ 1  3 ]
(%o1)                               [      ]
                                    [ 2  4 ]
(%i2) zn_determinant(m, 5);
(%o2)                                  3
(%i3) mi : zn_invert_by_lu(m, 5);
                                    [ 3  4 ]
(%o3)                               [      ]
                                    [ 1  2 ]
(%i4) matrixmap(lambda([a], mod(a, 5)), m . mi);
                                    [ 1  0 ]
(%o4)                               [      ]
                                    [ 0  1 ]
```

zn_log (a, *g*, *n*) [Función]

zn_log (a, *g*, *n*, [[*p1*, *e1*], . . . , [*pk*, *ek*]]) [Función]

Calcula el logaritmo discreto. Sea (Z/*n*Z)* un grupo cíclico, *g* una raíz primitiva módulo *n* y *a* un miembro de este grupo, entonces `zn_log (a, g, n)` calcula la congruencia `g^x = a mod n`.

El algoritmo que se aplica necesita una factorización prima de `totient(n)`. Esta factorización puede requerir mucho tiempo de cálculo, por lo que en ciertos casos puede ser aconsejable factorizar primero y luego pasar la lista de factores a `zn_log` como cuarto argumento. La lista debe ser de la misma forma que las lista devuelta por `ifactors(totient(n))` utilizando la opción por defecto `factors_only : false`.

El algoritmo utiliza la reducción de Pohlig-Hellman y el método Rho de Pollard para los logaritmos discretos. El tiempo de ejecución de `zn_log` depende en primer lugar del número de bits del mayor factor primo del totient.

Véanse también `zn_primroot`, `zn_order`, `ifactors` y `totient`.

Ejemplos:

`zn_log (a, g, n)` resuelve la congruencia `g^x = a mod n`.

```
(%i1) n : 22$
```

```
(%i2) g : zn_primroot(n);
(%o2)                                7
(%i3) ord_7 : zn_order(7, n);
(%o3)                               10
(%i4) powers_7 : makelist(power_mod(g, x, n), x, 0, ord_7 - 1);
(%o4)             [1, 7, 5, 13, 3, 21, 15, 17, 9, 19]
(%i5) zn_log(21, g, n);
(%o5)                                5
(%i6) map(lambda([x], zn_log(x, g, n)), powers_7);
(%o6)               [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

El cuarto argumento opcional debe ser de la misma forma que la lista devuelta por `ifactors(totient(n))`.

```
(%i1) (p : 2^127-1, primep(p));
(%o1)                               true
(%i2) ifs : ifactors(p - 1)$
(%i3) g : zn_primroot(p, ifs);
(%o3)                                43
(%i4) a : power_mod(g, 1234567890, p)$
(%i5) zn_log(a, g, p, ifs);
(%o5)                           1234567890
(%i6) time(%o5);
(%o6)                             [1.204]
(%i7) f_max : last(ifs);
(%o7)                        [77158673929, 1]
(%i8) slength( printf(false, "~b", f_max[1]) );
(%o8)                                37
```

`zn_mult_table` (*n*) [Función]

`zn_mult_table` (*n*, *all*) [Función]

Sin el argumento opcional *all*, `zn_mult_table(n)` muestra la tabla de multiplicación de los elementos de $(\mathrm{Z}/n\mathrm{Z})^*$, que son todos elementos invertibles módulo *n*.

El argumento opcional *all* hace que la tabla se genere para todos los elementos no nulos.

Véanse también `zn_add_table` y `zn_power_table`.

Ejemplo:

```
(%i1) zn_mult_table(4);
                                [ 1  3 ]
(%o1)                           [      ]
                                [ 3  1 ]
(%i2) zn_mult_table(4, all);
                               [ 1  2  3 ]
                               [         ]
(%o2)                          [ 2  0  2 ]
                               [         ]
                               [ 3  2  1 ]
```

`zn_order` (*x*, *n*) [Función]
`zn_order` (*x*, *n*, [[*p1*, *e1*], . . . , [*pk*, *ek*]]) [Función]

Devuelve el orden de *x* si es una unidad del grupo finito (Z/*n*Z)*, o devuelve `false`. *x* una unidad módulo *n* si es coprimo con *n*.

El algoritmo que se aplica necesita una factorización prima de `totient(n)`. Esta factorización puede requerir mucho tiempo de cálculo, por lo que en ciertos casos puede ser aconsejable factorizar primero y luego pasar la lista de factores a `zn_log` como tercer argumento. La lista debe ser de la misma forma que las lista devuelta por `ifactors(totient(n))` utilizando la opción por defecto `factors_only : false`.

Véanse también `zn_primroot`, `ifactors` y `totient`.

Ejemplos:

`zn_order` calcula el orden de la unidad *x* en (Z/*n*Z)*.

```
(%i1) n : 22$
(%i2) g : zn_primroot(n);
(%o2)                                7
(%i3) units_22 : sublist(makelist(i,i,1,21), lambda([x], gcd(x, n) = 1));
(%o3)             [1, 3, 5, 7, 9, 13, 15, 17, 19, 21]
(%i4) (ord_7 : zn_order(7, n)) = totient(n);
(%o4)                             10 = 10
(%i5) powers_7 : makelist(power_mod(g,i,n), i,0,ord_7 - 1);
(%o5)             [1, 7, 5, 13, 3, 21, 15, 17, 9, 19]
(%i6) map(lambda([x], zn_order(x, n)), powers_7);
(%o6)             [1, 10, 5, 10, 5, 2, 5, 10, 5, 10]
(%i7) map(lambda([x], ord_7/gcd(x, ord_7)), makelist(i, i,0,ord_7 - 1));
(%o7)             [1, 10, 5, 10, 5, 2, 5, 10, 5, 10]
(%i8) totient(totient(n));
(%o8)                                4
```

El tercer argumento opcional debe ser de la misma forma que la lista devuelta por `ifactors(totient(n))`.

```
(%i1) (p : 2^142 + 217, primep(p));
(%o1)                             true
(%i2) ifs : ifactors( totient(p) )$
(%i3) g : zn_primroot(p, ifs);
(%o3)                                3
(%i4) is( (ord_3 : zn_order(g, p, ifs)) = totient(p) );
(%o4)                             true
(%i5) map(lambda([x], ord_3/zn_order(x, p, ifs)), makelist(i,i,2,15));
(%o5)        [22, 1, 44, 10, 5, 2, 22, 2, 8, 2, 1, 1, 20, 1]
```

`zn_power_table` (*n*) [Función]
`zn_power_table` (*n*, *all*) [Función]

Sin el argumento opcional *all*, `zn_power_table(n)` muestra la tabla de potencias de los elementos de (Z/*n*Z)*, que son todos elementos invertibles módulo *n*. El exponente se obtiene con un bucle desde `1` hasta `totient(n)` y la tabla termina con una columna de unos al lado derecho.

El argumento opcional *all* hace que la tabla se genere para todos los elementos no nulos. En este caso, el exponente se calcula con un bucle desde `1` hasta `totient(n) + 1` y la última columna es por lo tanto igual a la primera.

Véanse también `zn_add_table` y `zn_mult_table`.

Ejemplo:

```
(%i1) zn_power_table(6);
                                  [ 1  1 ]
(%o1)                             [      ]
                                  [ 5  1 ]
(%i2) zn_power_table(6, all);
                                [ 1  1  1 ]
                                [         ]
                                [ 2  4  2 ]
                                [         ]
(%o2)                           [ 3  3  3 ]
                                [         ]
                                [ 4  4  4 ]
                                [         ]
                                [ 5  1  5 ]
```

`zn_primroot` (*n*) [Función]

`zn_primroot` (*n*, [[*p1*, *e1*], ..., [*pk*, *ek*]]) [Función]

Si el grupo multiplicativo es cíclico, `zn_primroot` calcula la menor raíz primitiva de módulo *n*. (Z/*n*Z)* es cíclico si *n* es igual a `2`, `4`, `p^k` o `2*p^k`, siendo `p` primo y mayor que `2` y `k` un número natural. Si a la variable opcional `zn_primroot_pretest`, cuyo valor por defecto es `false`, se le da el valor `true`, entonces `zn_primroot` realiza una prueba previa. En cualquier caso, el cálculo está limitado por la cota superior `zn_primroot_limit`.

Si (Z/*n*Z)* no es cíclico o si no tiene raíces primitivas menores que `zn_primroot_limit`, `zn_primroot` devuelve `false`.

El algoritmo que se aplica necesita una factorización prima de `totient(n)`. Esta factorización puede requerir mucho tiempo de cálculo, por lo que en ciertos casos puede ser aconsejable factorizar primero y luego pasar la lista de factores a `zn_log` como argumento adicional. La lista debe ser de la misma forma que las lista devuelta por `ifactors(totient(n))` utilizando la opción por defecto `factors_only : false`.

Véanse también `zn_primroot_p`, `zn_order`, `ifactors` y `totient`.

Ejemplos:

`zn_primroot` calcula la menor raíz primitiva de módulo *n* o devuelve `false`.

```
(%i1) n : 14$
(%i2) g : zn_primroot(n);
(%o2)                               3
(%i3) zn_order(g, n) = totient(n);
(%o3)                             6 = 6
(%i4) n : 15$
(%i5) zn_primroot(n);
```

```
(%o5)                                false
```

El argumento opcional debe ser de la misma forma que la lista devuelta por `ifactors(totient(n))`.

```
(%i1) (p : 2^142 + 217, primep(p));
(%o1)                                true
(%i2) ifs : ifactors( totient(p) )$
(%i3) g : zn_primroot(p, ifs);
(%o3)                                  3
(%i4) [time(%o2), time(%o3)];
(%o4)                     [[15.556972], [0.004]]
(%i5) is(zn_order(g, p, ifs) = p - 1);
(%o5)                                true
(%i6) n : 2^142 + 216$
(%i7) ifs : ifactors(totient(n))$
(%i8) zn_primroot(n, ifs),
      zn_primroot_limit : 200, zn_primroot_verbose : true;
'zn_primroot' stopped at zn_primroot_limit = 200
(%o8)                                false
```

`zn_primroot_limit` [Option variable]

Valor por defecto: `1000`

Si `zn_primroot` no puede encontrar una raíz primitiva, entonces se para en esta cota superior. Si a la variable opcional `zn_primroot_verbose` se le da el valor `true`, se imprimirá un mensaje cuando `zn_primroot_limit` sea alcanzado.

`zn_primroot_p` (*x*, *n*) [Función]
`zn_primroot_p` (*x*, *n*, [[*p1*, *e1*], . . . , [*pk*, *ek*]]) [Función]

Comprueba si x es una raíz primitiva en el grupo multiplizativo $(\mathrm{Z}/n\mathrm{Z})^*$.

El algoritmo que se aplica necesita una factorización prima de `totient(n)`. Esta factorización puede requerir mucho tiempo de cálculo, por lo que en ciertos casos puede ser aconsejable factorizar primero y luego pasar la lista de factores a `zn_log` como tercer argumento. La lista debe ser de la misma forma que las lista devuelta por `ifactors(totient(n))` utilizando la opción por defecto `factors_only : false`.

Véanse también `zn_primroot`, `zn_order`, `ifactors` y `totient`.

Ejemplos:

`zn_primroot_p` como función de predicado.

```
(%i1) n : 14$
(%i2) units_14 : sublist(makelist(i,i,1,13), lambda([i], gcd(i, n) = 1));
(%o2)                      [1, 3, 5, 9, 11, 13]
(%i3) zn_primroot_p(13, n);
(%o3)                               false
(%i4) sublist(units_14, lambda([x], zn_primroot_p(x, n)));
(%o4)                               [3, 5]
(%i5) map(lambda([x], zn_order(x, n)), units_14);
(%o5)                       [1, 6, 6, 3, 3, 2]
```

El tercer argumento opcional debe ser de la misma forma que la lista devuelta por `ifactors(totient(n))`.

```
(%i1) (p : 2^142 + 217, primep(p));
(%o1)                                true
(%i2) ifs : ifactors( totient(p) )$
(%i3) sublist(makelist(i,i,1,50), lambda([x], zn_primroot_p(x, p, ifs)));
(%o3)      [3, 12, 13, 15, 21, 24, 26, 27, 29, 33, 38, 42, 48]
(%i4) [time(%o2), time(%o3)];
(%o4)                    [[7.748484], [0.036002]]
```

`zn_primroot_pretest` [Option variable]

Valor por defecto: `false`

El grupo multiplicativo $(\mathrm{Z}/n\mathrm{Z})^*$ es cíclico si if n es igual a `2`, `4`, `p^k` o `2*p^k`, siendo `p` un número primo mayor que `2` y `k` es un número natural.

La variable `zn_primroot_pretest` controla si `zn_primroot` debe comprobar si sucede alguna de estas situaciones antes de calcular la menor raíz primitiva. Solo se realizará esta comprobación si `zn_primroot_pretest` toma el valor `true`.

`zn_primroot_verbose` [Option variable]

Valor por defecto: `false`

Controla si `zn_primroot` imprime un mensaje cuando alcanza `zn_primroot_limit`.

30 Simetrías

Paquete escrito para Macsyma-Symbolics por Annick Valibouze[1]. Los algoritmos están descritos en los siguientes artículos:

1. Fonctions symétriques et changements de bases[2]. Annick Valibouze. EUROCAL'87 (Leipzig, 1987), 323–332, Lecture Notes in Comput. Sci 378. Springer, Berlin, 1989.
2. Résolvantes et fonctions symétriques[3]. Annick Valibouze. Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation, ISSAC'89 (Portland, Oregon). ACM Press, 390-399, 1989.
3. Symbolic computation with symmetric polynomials, an extension to Macsyma[4]. Annick Valibouze. Computers and Mathematics (MIT, USA, June 13-17, 1989), Springer-Verlag, New York Berlin, 308-320, 1989.
4. Théorie de Galois Constructive. Annick Valibouze. Mémoire d'habilitation à diriger les recherches (HDR), Université P. et M. Curie (Paris VI), 1994

30.1 Funciones y variables para simetrías

`comp2pui` (*n*, *l*) [Función]

Realiza el paso de las funciones simétricas completas de la lista *l* a las funciones simétricas elementales de 0 a *n*. En caso de que la lista *l* contenga menos de `n+1` elementos, se completará con valores formales. El primer elemento de la lista *l* almacena el cardinal del alfabeto, en caso de que exista; en caso contrario se le da el valor *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*) [Función]

Convierte el polinomio particionado asociado a la forma contraída *pc*, cuyas variables se encuentran en *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]]
```

Otras funciones para efectuar cambios de representación son: `contract`, `explose`, `part2cont`, `partpol`, `tcontract` y `tpartpol`.

`contract` (*psym*, *lvar*) [Función]

Convierte una forma contraída (como un monomio por órbita sobre la acción del grupo simétrico) del polinomio *psym* cuyas variables se encuentran en la lista *lvar*.

---

[1] https://web.archive.org/web/20061125035035/http://www-calfor.lip6.fr/~avb/

[2] www.stix.polytechnique.fr/publications/1984-1994.html

[3] https://web.archive.org/web/20061125035035/http://www-calfor.lip6.fr/~avb/DonneesTelechargeables/MesArticles/issac89ACMValibouze.pdf

[4] www.stix.polytechnique.fr/publications/1984-1994.html

La función `explose` realiza la operación inversa. A mayopes, la función `tcontract` comprueba la simetría del polinomio.

```
(%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
```

Otras funciones para efectuar cambios de representación son:

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

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

Calcula la imagen directa (véase M. Giusti, D. Lazard et A. Valibouze, ISSAC 1988, Roma) asociada a la función *f*, en las listas de variables *lvar_1*, ..., *lvar_n*, y en los polinomios *p_1*, ..., *p_n* de una variable *y*. Si la expresión de *f* no depende de variable alguna, no sólo es inútil aportar esa variable, sino que también disminuyen considerablemente los cálculos cuando la variable no se declara.

```
(%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
```

Búsqueda del polinomio cuyas raíces son la suma $a+u$ o a es la raíz de $z^2-e1*z+e2$ y u es la 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
```

La función `direct` acepta dos indicadores: `elementaires` (elementales) y `puissances` (potenciales, que es el valor por defecto) que permiten hacer la descomposición de los polinomios simétricos que aparezcan en los cálculos en funciones simétricas elementales o en funciones potenciales, respectivamente.

Funciones de `sym` utilizadas en esta función:

`multi_orbit`(por tanto `orbit`),`pui_direct`, `multi_elem` (por tanto `elem`), `multi_pui` (por tanto `pui`), `pui2ele`, `ele2pui` (si al indicador `direct` se le asignó `puissances`).

`ele2comp` (*m*, *l*) [Función]

Pasa las funciones simétricas elementales a funciones completas, de forma similar a `comp2ele` y `comp2pui`.

Otras funciones para cambio de bases son:

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

ele2polynome (*l*, *z*) [Función]

Devuelve el polinomio en *z* en el que las funciones simétricas elementales de las raíces son las de la lista *l*. `l = [n, e_1, ..., e_n]`, donde *n* es el grado del polinomio y *e_i* la *i*-ésima función simétrica elemental.

```
(%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
```

La función recíproca es `polynome2ele (P, z)`

Véanse también `polynome2ele` y `pui2polynome`.

ele2pui (*m*, *l*) [Función]

Pasa las funciones simétricas elementales a funciones completas, de forma similar a `comp2ele` y `comp2comp`.

Otras funciones para cambio de bases son:

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

elem (*ele*, *sym*, *lvar*) [Función]

Descompone el polinomio simétrico *sym* con las variables continuas de la lista *lvar* en las funciones simétricas elementales contenidas en la lista *ele*. El primer elemento de la lista *ele* almacena el cardinal del alfabeto, en caso de que exista; en caso contrario se le da como valor el grado del polinomio *sym*. Si faltan valores en la lista *ele*, ésta se completará con valores formales del tipo "ei". El polinomio *sym* puede especificarse de tres formas diferentes: contraído (en cuyo caso `elem` debe valer 1, que es el valor por defecto), particionado (`elem` valdrá 3) o extendido (por ejemplo, el polinomio completo) (en este caso, `elem` valdrá 2). La utilización de la función `pui` se hace siguiendo este mismo modelo.

Con un alfabeto de cardinal 3 con *e1*, la primera función simétrica elemental valiendo 7, el polinomio simétrico de tres variables cuya forma contraída (aquí dependiendo solamente de dos de sus variables) es $^4 - 2 * x * y$, se descompone en funciones simétricas elementales:

```
(%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
```

Otras funciones para cambio de bases son: `comp2ele`, `comp2pui`, `ele2comp`, `ele2pui`, `mon2schur`, `multi_elem`, `multi_pui`, `pui`, `pui2comp`, `pui2ele`, `puireduc` y `schur2comp`.

`explose` (*pc*, *lvar*) [Función]

Devuelve el polinomio simétrico asociado a la forma contraída *pc*. La lista *lvar* contiene las variables.

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

Otras funciones para efectuar cambios de representación son: `contract`, `cont2part`, `part2cont`, `partpol`, `tcontract` y `tpartpol`.

`kostka` (*part_1*, *part_2*) [Función]

Función escrita por P. Espert, calcula el número de Kostka asociado a las particiones *part_1* y *part_2*.

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

`lgtreillis` (*n*, *m*) [Función]

Devuelve la lista de particiones de peso *n* y longitud *m*.

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

Véanse también `ltreillis`, `treillis` y `treinat`.

`ltreillis` (*n*, *m*) [Función]

Devuelve la lista de particiones de peso *n* y longitud menor o igual que *m*.

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

Véanse tambiént `lgtreillis`, `treillis` y `treinat`.

`mon2schur` (*l*) [Función]

La lista *l* representa la función de Schur S_*l*: Se tiene *l* = [*i_1*, *i_2*, ..., *i_q*] con *i_1* <= *i_2* <= ... <= *i_q*. La función de Schur es S_[*i_1*, *i_2*, ..., *i_q*], el menor de la matriz infinita (h_{i-j}) *i* >= 1, *j* >= 1 compuesto de las *q* primeras filas y columnas *i_1* + 1, *i_2* + 2, ..., *i_q* + *q*.

Se ha escrito esta función de Schur en función de las formas monomiales utilizando las funciones `treinat` y `kostka`. La forma devuelta es un polinomio simétrico en una de sus representaciones contraídas con las variables *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
```

Para 3 variables se tendrá:

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

Otras funciones para cambio de bases son:

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

`multi_elem` (*l_elem*, *multi_pc*, *l_var*) [Función]

Descompone un polinomio multisimétrico sobre una forma multicontraída *multi_pc* en los grupos de variables contenidas en la lista de listas *l_var* sobre los grupos de funciones simétricas elementales contenidas en *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
```

Otras funciones para cambio de bases son:

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

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

P es un polinomio en el conjunto de variables contenidas en las listas *lvar_1*, *lvar_2*, ..., *lvar_p*. Esta función restablece la órbita del polinomio *P* sobre la acción del producto de los grupos simétricos de los conjuntos de variables representadas por esas *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]
```

Véase también `orbit` para la acción de un solo grupo simérico.

`multi_pui` [Función]

Es a la función `pui` lo que la función `multi_elem` es a la función `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*) [Función]

El argumento *r* es el peso de la partición *part*. Esta función calcula el coeficiente multinomial asociado: si las partes de las particiones *part* son *i_1*, *i_2*, ..., *i_k*, el resultado de `multinomial` es *r*`!/(`*i_1*`!` *i_2*`!` `...` *i_k*`!)`.

`multsym` (*ppart_1*, *ppart_2*, *n*) [Función]

Calcula el producto de dos polinomios simétricos de *n* variables operando solamente con el módulo de la acción del grupo simétrico de orden *n*. Los polinomios están en su representación particionada.

Sean los dos polinomios simétricos en `x` e `y`: `3*(x + y) + 2*x*y` y `5*(x^2 + y^2)` cuyas formas particionadas son `[[3, 1], [2, 1, 1]]` y `[[5, 2]]`, respectivamente; el producto de ambos será:

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

o sea, `10*(x^3*y + y^3*x) + 15*(x^2*y + y^2*x) + 15*(x^3 + y^3)`.

Funciones de cambio de representación de un polinomio simétrico:

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

`orbit` (*P*, *lvar*) [Función]

Calcula la órbita de un polinomio *P* en las variables de la lista *lvar* bajo la acción del grupo simétrico del conjunto de variables contenidas en la 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]
```

Véase también `multi_orbit` para la acción de un producto de grupos simétricos sobre un polinomio.

`part2cont` (*ppart*, *lvar*) [Función]

Transforma un polinomio simétrico de su forma particionada a su forma contraída. La forma contraída se devuelve con las variables contenidas en *lvar*.

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

Otras funciones para efectuar cambios de representación son:

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

`partpol` (*psym*, *lvar*) [Función]

Restablece la representación particionada del polinomio simétrico *psym* de variables en *lvar*.

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

Otras funciones para efectuar cambios de representación son:

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

`permut` (*l*) [Función]

Devuelve la lista de permutaciones de la lista *l*.

`polynome2ele` (*P*, *x*) [Función]

Devuelve la lista `l = [n, e_1, ..., e_n]`, en la que *n* es el grado del polinomio *P* de variable *x* y *e_i* es la *i*-ésima función simétrica elemental de las raíces 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
```

La función recíproca es `ele2polynome (l, x)`.

`prodrac (l, k)` [Función]

Siendo *l* una lista que contiene las funciones simétricas elementales sobre un conjunto *A*, la función `prodrac` calcula el polinomio cuyas raíces son los productos *k* a *k* de los elementos de *A*.

`pui (l, sym, lvar)` [Función]

Descompone el polinomio simétrico *sym*, cuyas variables son las contenidas en *lvar*, en las funciones potenciales contenidas en la lista *l*. El primer elemento de la lista *l* almacena el cardinal del alfabeto, en caso de que exista; en caso contrario se le da el grado del polinomio *sym*. Si faltan los valores de la lista *l*, en su lugar serán colocados valores formales del tipo "pi". El polinomio *sym* puede especificarse de tres formas diferentes: contraído (en cuyo caso `pui` debe valer 1, que es el valor por defecto), particionado (`pui` valdrá 3) o extendido (por ejemplo, el polinomio completo) (en este caso, `pui` valdrá 2). La utilización de la función `elem` se hace siguiendo este mismo modelo.

```
(%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
```

Otras funciones para cambio de bases son: `comp2ele`, `comp2pui`, `ele2comp`, `ele2pui`, `elem`, `mon2schur`, `multi_elem`, `multi_pui`, `pui2comp`, `pui2ele`, `puireduc` y `schur2comp`.

`pui2comp (n, lpui)` [Función]

Devuelve la lista de las *n* primeras funciones completas (con el cardinal en primer lugar) en función de las funciones potenciales dadas en la lista *lpui*. Si la lista *lpui* estuviese vacía, el cardinal sería *N*; si no estuviese vacía, se tomaría como cardinal su primer elemento, de forma similar a como se procede en `comp2ele` y en `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
```

Otras funciones para cambio de bases son: `comp2ele`, `comp2pui`, `ele2comp`, `ele2pui`, `elem`, `mon2schur`, `multi_elem`, `multi_pui`, `pui`, `pui2ele`, `puireduc` y `schur2comp`.

`pui2ele` (*n*, *lpui*) [Función]

Transforma las funciones potenciales a funciones simétricas elementales. Si la variable global `pui2ele` vale `girard`, se recupera la lista de funciones simétricas elementales de 1 *n*, y si es igual a `close`, se recupera la *n*-ésima función simétrica elemental.

Otras funciones para cambio de bases son: `comp2ele`, `comp2pui`, `ele2comp`, `ele2pui`, `elem`, `mon2schur`, `multi_elem`, `multi_pui`, `pui`, `pui2comp`, `puireduc` y `schur2comp`.

`pui2polynome` (*x*, *lpui*) [Función]

Calcula el polinomio en *x* cuyas raíces tienen como funciones potenciales las dadas en la 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
```

Véanse también `polynome2ele` y `ele2polynome`.

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

Sea *f* un polinomio en *n* bloques de variables *lvar_1*, ..., *lvar_n*. Sea *c_i* el número de variables en *lvar_i* y *SC* el producto de los *n* grupos simétricos de grados *c_1*, ..., *c_n*, que actúan sobre *f*. La lista *orbite* es la órbita, representada por `SC(f)`, de la función *f* sobre la acción de *SC*, la cual puede ser obtenida por medio de la función `multi_orbit`. Los valores `d_i` son enteros tales que *c_1* <= *d_1*, *c_2* <= *d_2*, ..., *c_n* <= *d_n*. Por último, sea *SD* el producto de los grupos simétricos *S_d1* x *S_d2* x ... x *S_dn*.

La función `pui_direct` devuelve las n primeras funciones potenciales de `SD(f)` deducidas de las funciones potenciales de `SC(f)`, siendo n el cardinal de `SD(f)`.

El resultado se devuelve en la forma multicontraída respecto 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, 3]);
                              2             2
(%o4) [3 x + 2 a, 6 x y + 3 x  + 4 a x + 4 a ,

                2                   3       2       2       3
              9 x  y + 12 a x y + 3 x  + 6 a x  + 12 a  x + 8 a ]
```

`puireduc` (*n*, *lpui*) [Función]

Siendo *lpui* una lista en la que el primer elemento es un entero m, `puireduc` devuelve las n primeras funciones potenciales en función de las m primeras.

```
(%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*]) [Función]

Calcula la resolvente del polinomio P de variable x y grado $n >= d$ por la función f de variables *x_1*, ..., *x_d*. Para mejorar los cálculos, es importante no incluir en la lista `[x_1, ..., x_d]` las variables que no intervienen en la función de transformación *f*.

Con el fin de hacer más eficaces los cálculos, se puede asignar a `resolvante` un indicador que permita seleccionar el algoritmo más apropiado:

- `unitaire`,
- `lineaire`,
- `alternee`,
- `somme`,
- `produit`,
- `cayley`,
- `generale`.

```
(%i1) resolvante: unitaire$
(%i2) resolvante (x^7 - 14*x^5 + 56*x^3 - 56*x + 22, x, x^3 - 1,
      [x]);

" resolvante unitaire " [7, 0, 28, 0, 168, 0, 1120, - 154, 7840,
                         - 2772, 56448, - 33880,

413952, - 352352, 3076668, - 3363360, 23114112, - 30494464,

175230832, - 267412992, 1338886528, - 2292126760]
  3       6      3       9      6      3
[x  - 1, x  - 2 x  + 1, x  - 3 x  + 3 x  - 1,

 12      9      6      3       15      12       9       6      3
x   - 4 x  + 6 x  - 4 x  + 1, x   - 5 x   + 10 x  - 10 x  + 5 x

       18      15       12       9       6      3
 - 1, x   - 6 x   + 15 x   - 20 x  + 15 x  - 6 x  + 1,

 21      18       15       12       9       6      3
x   - 7 x   + 21 x   - 35 x   + 35 x  - 21 x  + 7 x  - 1]
[- 7, 1127, - 6139, 431767, - 5472047, 201692519, - 3603982011]
       7      6        5         4          3           2
(%o2) y  + 7 y  - 539 y  - 1841 y  + 51443 y  + 315133 y

                                                + 376999 y + 125253
(%i3) resolvante: lineaire$
(%i4) resolvante (x^4 - 1, x, x1 + 2*x2 + 3*x3, [x1, x2, x3]);

" resolvante lineaire "
       24       20         16            12             8
(%o4) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                    4
                                          + 344489984 y  + 655360000
(%i5) resolvante: general$
```

```
(%i6) resolvante (x^4 - 1, x, x1 + 2*x2 + 3*x3, [x1, x2, x3]);

" resolvante generale "
       24       20         16            12             8
(%o6) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                      4
                                           + 344489984 y  + 655360000
(%i7) resolvante (x^4 - 1, x, x1 + 2*x2 + 3*x3, [x1, x2, x3, x4]);

" resolvante generale "
       24       20         16            12             8
(%o7) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                      4
                                           + 344489984 y  + 655360000
(%i8) direct ([x^4 - 1], x, x1 + 2*x2 + 3*x3, [[x1, x2, x3]]);
       24       20         16            12             8
(%o8) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                      4
                                           + 344489984 y  + 655360000
(%i9) resolvante :lineaire$
(%i10) resolvante (x^4 - 1, x, x1 + x2 + x3, [x1, x2, x3]);

" resolvante lineaire "
                              4
(%o10)                       y  - 1
(%i11) resolvante: symetrique$
(%i12) resolvante (x^4 - 1, x, x1 + x2 + x3, [x1, x2, x3]);

" resolvante symetrique "
                              4
(%o12)                       y  - 1
(%i13) resolvante (x^4 + x + 1, x, x1 - x2, [x1, x2]);

" resolvante symetrique "
                           6      2
(%o13)                    y  - 4 y  - 1
(%i14) resolvante: alternee$
(%i15) resolvante (x^4 + x + 1, x, x1 - x2, [x1, x2]);

" resolvante alternee "
           12      8       6        4        2
(%o15)     y   + 8 y  + 26 y  - 112 y  + 216 y  + 229
(%i16) resolvante: produit$
```

```
(%i17) resolvante (x^7 - 7*x + 3, x, x1*x2*x3, [x1, x2, x3]);

" resolvante produit "
       35      33         29        28         27        26
(%o17) y   - 7 y   - 1029 y   + 135 y   + 7203 y   - 756 y

         24           23          22            21           20
 + 1323 y   + 352947 y   - 46305 y   - 2463339 y   + 324135 y

          19           18             17              15
 - 30618 y   - 453789 y   - 40246444 y   + 282225202 y

             14              12             11            10
 - 44274492 y   + 155098503 y   + 12252303 y   + 2893401 y

              9            8            7             6
 - 171532242 y  + 6751269 y  + 2657205 y  - 94517766 y

            5             3
 - 3720087 y  + 26040609 y  + 14348907
(%i18) resolvante: symetrique$
(%i19) resolvante (x^7 - 7*x + 3, x, x1*x2*x3, [x1, x2, x3]);

" resolvante symetrique "
       35      33         29        28         27        26
(%o19) y   - 7 y   - 1029 y   + 135 y   + 7203 y   - 756 y

         24           23          22            21           20
 + 1323 y   + 352947 y   - 46305 y   - 2463339 y   + 324135 y

          19           18             17              15
 - 30618 y   - 453789 y   - 40246444 y   + 282225202 y

             14              12             11            10
 - 44274492 y   + 155098503 y   + 12252303 y   + 2893401 y

              9            8            7             6
 - 171532242 y  + 6751269 y  + 2657205 y  - 94517766 y

            5             3
 - 3720087 y  + 26040609 y  + 14348907
(%i20) resolvante: cayley$
```

```
(%i21) resolvante (x^5 - 4*x^2 + x + 1, x, a, []);

" resolvante de Cayley "
       6       5         4          3            2
(%o21) x  - 40 x  + 4080 x  - 92928 x  + 3772160 x  + 37880832 x

                                                          + 93392896
```

Para la resolvente de Cayley, los dos últimos argumentos son neutros y el polinomio dado en el argumento debe ser necesariamente de grado 5.

Véanse también:

`resolvante_bipartite`, `resolvante_produit_sym`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`.

`resolvante_alternee1` (*P*, *x*) [Función]

Calcula la transformación de `P(x)` de grado *n* por la función $\prod_{1\leq i<j\leq n-1} (x_i-x_j)$.

Véanse también:

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

`resolvante_bipartite` (*P*, *x*) [Función]

Calcula la transformación de `P(x)` de grado *n* (*n* par) por la función $x_1x_2\ldots x_{n/2}+x_{n/2+1}\ldotsx_n$

```
(%i1) resolvante_bipartite (x^6 + 108, x);
                10        8           6             4
(%o1)          y   - 972 y  + 314928 y  - 34012224 y
```

Véanse también:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante`, `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`, `resolvante_alternee1`.

`resolvante_diedrale` (*P*, *x*) [Función]

Calcula la transformación de `P(x)` por la función *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
```

Véanse también:

`resolvante_produit_sym`, `resolvante_unitaire`,

resolvante_alternee1, resolvante_klein, resolvante_klein3, resolvante_vierer, resolvante.

resolvante_klein (*P*, *x*) [Función]

Calcula la transformación de `P(x)` por la función `x_1 x_2 x_4 + x_4`.

Véanse también:

resolvante_produit_sym, resolvante_unitaire, resolvante_alternee1, resolvante, resolvante_klein3, resolvante_vierer, resolvante_diedrale.

resolvante_klein3 (*P*, *x*) [Función]

Calcula la transformación de `P(x)` por la función `x_1 x_2 x_4 + x_4`.

Véanse también:

resolvante_produit_sym, resolvante_unitaire, resolvante_alternee1, resolvante_klein, resolvante, resolvante_vierer, resolvante_diedrale.

resolvante_produit_sym (*P*, *x*) [Función]

Calcula la lista de todas las resolventes producto del polinomio `P(x)`.

```
(%i1) resolvante_produit_sym (x^5 + 3*x^4 + 2*x - 1, x);
       5      4             10      8       7       6       5
(%o1) [y  + 3 y  + 2 y - 1, y   - 2 y  - 21 y  - 31 y  - 14 y

    4       3      2       10      8       7    6       5       4
 - y  + 14 y  + 3 y  + 1, y   + 3 y  + 14 y  - y  - 14 y  - 31 y

       3      2       5      4
 - 21 y  - 2 y  + 1, y  - 2 y  - 3 y - 1, y - 1]
(%i2) resolvante: produit$
(%i3) resolvante (x^5 + 3*x^4 + 2*x - 1, x, a*b*c, [a, b, c]);

" resolvante produit "
       10      8       7    6        5       4       3     2
(%o3) y   + 3 y  + 14 y  - y  - 14 y  - 31 y  - 21 y  - 2 y  + 1
```

Véanse también:

resolvante, resolvante_unitaire, resolvante_alternee1, resolvante_klein, resolvante_klein3, resolvante_vierer, resolvante_diedrale.

resolvante_unitaire (*P*, *Q*, *x*) [Función]

Calcula la resolvente del polinomio `P(x)` por el polinomio `Q(x)`.

Véanse también:

resolvante_produit_sym, resolvante, resolvante_alternee1, resolvante_klein, resolvante_klein3, resolvante_vierer, resolvante_diedrale.

resolvante_vierer (*P*, *x*) [Función]

Calcula la transformación de `P(x)` por la función `x_1 x_2 - x_3 x_4`.

Véanse también:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante_klein`, `resolvante_klein3`, `resolvante`, `resolvante_diedrale`.

schur2comp (*P*, *l_var*) [Función]

P es un polinomio de variables contenidas en la lista *l_var*. Cada una de las variables de *l_var* representa una función simétrica completa. La *i*-ésima función simétrica completa de *l_var* se representa como la concatenación de la letra `h` con el entero *i*: `hi`. La función `schur2comp` devuelve la expresión de *P* en función de las funciones 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*) [Función]

Si la lista *l* contiene las funciones simétricas elementales de un polinomio *P*, la función `somrac` calcula el polinomio cuyas raíces son las sumas *k* a *k* de las raíces de *P*.

Véase también `prodrac`.

tcontract (*pol*, *lvar*) [Función]

Comprueba si el polinomio *pol* es simétrico en las variable contenidas en la lista *lvar*. En caso afirmativo, devuelve una forma contraída tal como lo hace la función `contract`.

Otras funciones para efectuar cambios de representación son: `contract`, `cont2part`, `explose`, `part2cont`, `partpol` y `tpartpol`.

tpartpol (*pol*, *lvar*) [Función]

Comprueba si el polinomio *pol* es simétrico en las variable contenidas en la lista *lvar*. En caso afirmativo, devuelve una forma particionada tal como lo hace la función `partpol`.

Otras funciones para efectuar cambios de representación son: `contract`, `cont2part`, `explose`, `part2cont`, `partpol` y `tcontract`.

treillis (*n*) [Función]

Devuelve todas las particiones de pesos *n*.

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

Véanse también `lgtreillis`, `ltreillis` y `treinat`.

`treinat` (*part*) [Función]

Devuelve la lista de las particiones inferiores de la partición *part* en su orden 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]]
```

Véanse también `lgtreillis`, `ltreillis` y `treillis`.

31 Grupos

31.1 Funciones y variables para grupos

`todd_coxeter` (*relaciones*, *subgrupo*) [Función]
`todd_coxeter` (*relaciones*) [Función]

Busca el orden de G/H donde G es el módulo del Grupo Libre de *relations*, y H es el subgrupo de G generado por `subgrupo`. `subgrupo` es un argumento opcional, cuyo valor por defecto es [].

En este proceso se obtiene una tabla de multiplicación para la acción correcta de G sobre G/H, donde los co-cojuntos son enumerados [H,Hg2,Hg3,...]. Esto puede ser observado internamente en el `todd_coxeter_state`.

Ejemplo:

```
(%i1) symet(n):=create_list(
        if (j - i) = 1 then (p(i,j))^^3 else
            if (not i = j) then (p(i,j))^^2 else
                p(i,i) , j, 1, n-1, i, 1, j);
                                                              <3>
(%o1) symet(n) := create_list(if j - i = 1 then p(i, j)

                                    <2>
 else (if not i = j then p(i, j)    else p(i, i)), j, 1, n - 1,

i, 1, j)
(%i2) p(i,j) := concat(x,i).concat(x,j);
(%o2)         p(i, j) := concat(x, i) . concat(x, j)
(%i3) symet(5);
          <2>           <3>    <2>           <2>           <3>
(%o3) [x1   , (x1 . x2)   , x2   , (x1 . x3)   , (x2 . x3)   ,

             <2>           <2>           <2>           <3>    <2>
           x3   , (x1 . x4)   , (x2 . x4)   , (x3 . x4)   , x4   ]
(%i4) todd_coxeter(%o3);

Rows tried 426
(%o4)                          120
(%i5) todd_coxeter(%o3,[x1]);

Rows tried 213
(%o5)                          60
(%i6) todd_coxeter(%o3,[x1,x2]);

Rows tried 71
(%o6)                          20
```

32 Entorno de Ejecución

32.1 Introducción al entorno de ejecución

El fichero `maxima-init.mac` se carga automáticamente cada vez que se empieza a ejecutar Maxima. Se puede utilizar `maxima-init.mac` para personalizar el entorno de Maxima. Si existe, `maxima-init.mac` se almacena normalmente en el directorio indicado por `maxima_userdir`, aunque puede estar alojado en cualquier otro directorio que esté al alcance de la función `file_search`.

He aquí un ejemplo de fichero `maxima-init.mac`:

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

En este ejemplo, `setup_autoload` le dice a Maxima que cargue en memoria el fichero `specfun.mac` si cualquiera de las funciones `ultraspherical` o `assoc_legendre_p` es invocada pero todavía no está definida. De esta manera, no es necesario recordar cargar el fichero antes de llamar a las funciones.

La sentencia `showtime: all` le dice a Maxima que haga una asignación a la variable `showtime`. El fichero `maxima-init.mac` puede contener cualesquiera otras asignaciones o sentencias de Maxima.

32.2 Interrupciones

El usuario puede detener un cómputo que esté consumiendo recursos excesivos con el carácter `^C` (control-C). La acción que se sigue por defecto es la detención del cómputo y la impresión de otro prompt. En este caso, no será posible reiniciar la tarea interrumpida.

Si a la variable Lisp `*debugger-hook*` se le asigna `nil` haciendo

```
:lisp (setq *debugger-hook* nil)
```

entonces tras recibir `^C`, Maxima entra en el depurador de Lisp y el usuario podrá utilizar el depurador para inspeccionar el entorno Lisp. La tarea que haya sido interrumpida podrá reiniciarse escribiendo `continue` en el depurado de Lisp. La forma de volver a Maxima desde el depurador de Lisp, que no sea la de permitir la computación hasta la terminación de la tarea, dependerá de la versión de Lisp.

En sistemas Unix el carácter `^Z` (control-Z) hace que Maxima deje de ejecutarse devolviendo el control al terminal del sistema. El comando `fg` hace que la ejecución de Maxima se reanude en el punto que lo dejó.

32.3 Funciones y variables para el entorno de ejecución

`maxima_tempdir` [Variable del sistema]

La variable `maxima_tempdir` almacena la ruta del directorio en el que Maxima crea ciertos ficheros temporales. En particular, los ficheros temporales para la realización de gráficos se guardan en `maxima_tempdir`.

El valor que inicialmente toma esta variable es el directorio de inicio del usuario, si Maxima es capaz de localizarlo; en caso contrario, Maxima intenta encontrar un directorio que sea aceptable.

A la variable `maxima_tempdir` se le puede asignar una cadena de caracteres con la ruta del directorio.

`maxima_userdir` [Variable del sistema]

La variable `maxima_userdir` almacena la ruta del directorio en el que Maxima buscará ficheros Lisp y de Maxima. Maxima también busca en otros directorios, guardando las variables `file_search_maxima` y `file_search_lisp` la lista completa de búsqueda.

El valor que inicialmente toma esta variable es el de un subdirectorio del directorio de inicio del usuario, si Maxima es capaz de localizarlo; en caso contrario, Maxima intenta encontrar un directorio que sea aceptable.

A la variable `maxima_userdir` se le puede asignar una cadena de caracteres con la ruta del directorio. Sin embargo, cambiando el valor de la variable `maxima_userdir` no se alteran `file_search_maxima` ni `file_search_lisp`, cuyos contenidos se modifican mediante otro sistema.

`room` () [Función]
`room` (*true*) [Función]
`room` (*false*) [Función]

Presenta una descrpción del estado de almacenamiento y gestión de la pila en Maxima. La llamada `room` invoca a la función Lisp homónima.

- `room ()` prints out a moderate description.
- `room (true)` prints out a verbose description.
- `room (false)` prints out a terse description.

`sstatus` (*keyword*, *item*) [Función]

Si *keyword* es el símbolo `feature`, *item* será colocado en la lista de propiedades del sistema. Una vez ejecutado `sstatus (keyword, item)`, `status (feature, item)` devuelve `true`. Si *keyword* es el símbolo `nofeature`, *item* se borrará de la lista de propiedades del sistema. Esto puede ser de utilidad para los autores de paquetes, permitiendo mantener el control sobre las propiedades que se han ido estableciendo.

Véase también `status`.

`status` (`feature`) [Función]
`status` (`feature`, *item*) [Función]

Devuelve información sobre la presencia o ausencia de ciertas propiedades dependientes del sistema.

- `status (feature)` devuelve una lista con características del sistema. Éstas incluyen la versión de Lisp, tipo de sistema operativo, etc. La lista puede variar de un Lisp a otro.
- `status (feature, item)` devuelve `true` si *item* está en la lista de elementos retornados por `status (feature)` y `false` en otro caso. La función `status` no evalúa el argumento *item*. El operador de doble comilla simple, `''`, permite la evaluación. Una propiedad cuyo nombre contenga un carácter especial debe ser suministrada como un argumento del tipo cadena. Por ejemplo, `status (feature, "ansi-cl")`.

Véase también `sstatus`.

La variable `features` contiene una lista de propiedades que se aplican a expresiones matemáticas. Véanse `features` y `featurep` para más información.

`system` (*command*) [Función]

Ejecuta la instrucción *command* como un proceso independiente de Maxima. La instrucción se le pasa a la consola del sistema para su ejecución. La función `system` no está soportada por todos los sistemas operativos, pero suele estarlo en todos los entornos Unix y similares.

Suponiendo que `_hist.out` es una lista de frecuencias que se quieren representar en un diagrama de barras utilizando el programa `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"));
```

A fin de hacer el diagrama y eliminar el archivo temporal posteriormente, hágase:

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

`time` (*%o1, %o2, %o3, ...*) [Función]

Devuelve una lista de los tiempos, en segundos, que fueron necesarios para calcular los resultados de las salidas `%o1`, `%o2`, `%o3`, ... Los tiempos devueltos son estimaciones hechas por Maxima del tiempo interno de computación. La función `time` sólo puede utilizarse para variables correspondientes a líneas de salida; para cualquier otro tipo de variables, `time` devuelve `unknown`.

Hágase `showtime: true` para que Maxima devuelva el tiempo de ejecución de cada línea de salida.

`timedate` () [Función]

`timedate` (*T*) [Función]

Sin argumento, `timedate` devuelve una cadena que representa la hora y fecha actuales. La cadena tiene el formato `YYYY-MM-DD HH:MM:SS[+|-]ZZ:ZZ`, donde los campos indicados son: año, mes, día, horas, minutos, segundos y número de horas de diferencia con respecto a la hora GMT.

Con argumento, `timedate(`*T*`)` devuelve la hora *T* como una cadena con formato `YYYY-MM-DD HH:MM:SS[+|-]ZZ:ZZ`. *T* se interpreta como el número de segundos transcurridos desde la medianoche del uno de enero de 1900, tal como lo devuelve `absolute_real_time`.

Ejemplos:

`timedate` sin argumento devuelve una cadena con la hora y fecha actuales.

```
(%i1) d : timedate ();
(%o1)                       2010-06-08 04:08:09+01:00
(%i2) print ("timedate reports current time", d) $
timedate reports current time 2010-06-08 04:08:09+01:00
```

`timedate` con argumento devuelve una cadena que representa al propio argumento.

```
(%i1) timedate (0);
```

```
(%o1)                      1900-01-01 01:00:00+01:00
(%i2) timedate (absolute_real_time () - 7*24*3600);
(%o2)                      2010-06-01 04:19:51+01:00
```

`absolute_real_time ()` [Función]

Devuelve el número de segundos transcurridos desde la medianoche del 1 de enero de 1900 UTC. Este valor es un número entero positivo.

Véanse también `elapsed_real_time` y `elapsed_run_time`.

Ejemplo:

```
(%i1) absolute_real_time ();
(%o1)                      3385045277
(%i2) 1900 + absolute_real_time () / (365.25 * 24 * 3600);
(%o2)                   2007.265612087104
```

`elapsed_real_time ()` [Función]

Devuelve los segundos (incluyendo fracciones de segundo) transcurridos desde que Maxima se inició (o reinició) la sesión de Maxima. Este valor es un decimal en coma flotante.

Véanse también `absolute_real_time` y `elapsed_run_time`.

Ejemplo:

```
(%i1) elapsed_real_time ();
(%o1)                       2.559324
(%i2) expand ((a + b)^500)$
(%i3) elapsed_real_time ();
(%o3)                       7.552087
```

`elapsed_run_time ()` [Función]

Devuelve una estimación en segundos (incluyendo fracciones de segundo) durante los cuales Maxima ha estado realizando cálculos desde que se inició (o reinició) la sesión actual. Este valor es un decimal en coma flotante.

Véanse también `absolute_real_time` y `elapsed_real_time`.

Ejemplo:

```
(%i1) elapsed_run_time ();
(%o1)                         0.04
(%i2) expand ((a + b)^500)$
(%i3) elapsed_run_time ();
(%o3)                         1.26
```

33 Miscelánea de opciones

33.1 Introducción a la miscelánea de opciones

En esta sección se comentan varias opciones que tienen un efecto global sobre le comportamiento de Maxima. También se comentan varias listas, como la de las funciones definidas por el usuario.

33.2 Share

El directorio "share" de Maxima contiene programas y ficheros de interés para los usuarios de Maxima, pero no forman parte del núcleo de Maxima. Estos programas se cargan en memoria con llamadas a las funciones `load` o `setup_autoload`.

El código `:lisp *maxima-sharedir*` muestra la localización del directorio "share" dentro del sistema de ficheros del usuario.

El código `printfile ("share.usg")` muestra una lista actualizada de paquetes en "share". Los usuarios pueden encontrar más información accediendo directamente a los contenidos del directorio "share".

33.3 Funciones y variables para la miscelánea de opciones

`askexp` [Variable del sistema]

Cuando se invoca a `asksign`, la expresión que se va a analizar es precisamente `askexp`.

`genindex` [Variable optativa]

Valor por defecto: `i`

La variable `genindex` es el prefijo alfabético utilizado para generar la siguiente variable de sumación en caso de necesidad.

`gensumnum` [Variable optativa]

Valor por defecto: 0

La variable `gensumnum` es el sufijo numérico utilizado para generar la siguiente variable de sumación. Si vale `false` entonces el índice consistirá solamente de `genindex`, sin sufijo numérico.

`gensym ()` [Función]
`gensym (x)` [Función]

`gensym()` crea y devuelve una nueva símbolo o variable sin valor asignado.

El nombre del nuevo símbolo está formado por la concatenación de un prefijo, cuyo valor por defecto es "g", y de un sufijo, el cual es la representación decimal de un número que coincide, por defecto, con el valor de un contador interno de Lisp.

En caso de suministrar el argumento *x*, siendo este una cadena, se utilizará como prefijo en lugar de "g", lo cual tendrá efecto sólo para esta llamada a `gensym`Â·

En caso de suministrar el argumento *x*, siendo este un número entero, se utilizará como sufijo en lugar del contador interno de Lisp, lo cual tendrá efecto sólo para esta llamada a `gensym`Â·

Si no se suministra el sufijo en forma explícita, y sólo en este caso, el contador interno sufrirá un incremento después de haber sido utilizado.

Ejemplos:

```
(%i1) gensym();
(%o1)                          g887
(%i2) gensym("new");
(%o2)                         new888
(%i3) gensym(123);
(%o3)                          g123
```

`packagefile` [Variable opcional]

Valor por defecto: `false`

Los autores de paquetes que utilizan `save` o `translate` para crear librerías para otros usuarios pueden hacer la asignación `packagefile: true` para prevenir que se añada información a las listas con información del sistema de Maxima, como `values` o `functions`.

`remvalue` (*nombre_1*, ..., *nombre_n*) [Función]

`remvalue` (*all*) [Función]

Elimina del sistema los valores de las variable de usuario *nombre_1*, ..., *nombre_n* (incluso las que tienen subíndices).

La llamada `remvalue (all)` elimina los valores de todas las variables en `values`, la lista de todas las variables a las que el usuario a dado algún nombre, pero no de aquéllas a las que Maxima asigna automáticamente un valor.

Véase también `values`.

`rncombine` (*expr*) [Función]

Transforma *expr* combinando todos los términos de *expr* que tengan denominadores idénticos o que difieran unos de otros por factores numéricos. Su comportamiento es diferente al de la función `combine`, que combina términos con iguales denominadores.

Haciendo `pfeformat: true` y utilizando `combine` se consiguen resultados similares a aquéllos que se pueden obtener con `rncombine`, pero `rncombine` realiza el paso adicional de multiplicar denominadores numéricos. Esto da como resultado expresiones en las que se pueden reconocer algunas cancelaciones.

Antes de utilizar esta función ejecútese `load("rncomb")`.

`setup_autoload` (*nombre_fichero*, *función_1*, ..., *función_n*) [Función]

Especifica que si alguna de las funciones *function_1*, ..., *function_n* es referenciada pero todavía no ha sido definida, se cargará *nombre_fichero* mediante una llamada a `load`. El *nombre_fichero* normalmente contendrá las definiciones de las funciones especificadas, aunque esto no es imperativo.

La función `setup_autoload` no opera con arreglos de funciones.

La función `setup_autoload` no evalúa sus argumentos.

Ejemplo:

```
(%i1) legendre_p (1, %pi);
(%o1)                  legendre_p(1, %pi)
```

```
(%i2) setup_autoload ("specfun.mac", legendre_p, ultraspherical);
(%o2)                          done
(%i3) ultraspherical (2, 1/2, %pi);
Warning - you are redefining the Macsyma function ultraspherical
Warning - you are redefining the Macsyma function legendre_p
                                2
                  3 (%pi - 1)
(%o3)             ------------ + 3 (%pi - 1) + 1
                       2
(%i4) legendre_p (1, %pi);
(%o4)                           %pi
(%i5) legendre_q (1, %pi);
                                %pi + 1
                           %pi log(-------)
                                1 - %pi
(%o5)                      ---------------- - 1
                                  2
```

`tcl_output` (*list*, *i0*, *skip*) [Función]

`tcl_output` (*list*, *i0*) [Función]

`tcl_output` ([*list_1*, ..., *list_n*], *i*) [Función]

Imprime los elementos de una lista encerrándolos con llaves `{ }`, de forma apropiada para ser utilizado en un programa en el lenguaje Tcl/Tk.

`tcl_output (`*list*`, `*i0*`, `*skip*`)` imprime *list*, empezando por el elemento *i0* siguiendo luego con los elementos *i0* + *skip*, *i0* + 2 *skip*, etc.

`tcl_output (`*list*`, `*i0*`)` equivale a `tcl_output (`*list*`, `*i0*`, 2)`.

`tcl_output ([`*list_1*`, ..., `*list_n*`] , `*i*`)` imprime los *i*-ésimos elementos de *list_1*, ..., *list_n*.

Ejemplos:

```
(%i1) tcl_output ([1, 2, 3, 4, 5, 6], 1, 3)$

 {1.000000000     4.000000000
 }
(%i2) tcl_output ([1, 2, 3, 4, 5, 6], 2, 3)$

 {2.000000000     5.000000000
 }
(%i3) tcl_output ([3/7, 5/9, 11/13, 13/17], 1)$

 {((RAT SIMP) 3 7) ((RAT SIMP) 11 13)
 }
(%i4) tcl_output ([x1, y1, x2, y2, x3, y3], 2)$

 {$Y1 $Y2 $Y3
 }
(%i5) tcl_output ([[1, 2, 3], [11, 22, 33]], 1)$
```

```
{SIMP 1.000000000     11.00000000
}
```

34 Reglas y patrones

34.1 Introducción a reglas y patrones

Esta sección describe las reglas de simplificación y los patrones de comparación definidos por el usuario. Hay dos grupos de funciones que implementan diferentes esquemas de comparación de patrones. En un grupo están `tellsimp`, `tellsimpafter`, `defmatch`, `defrule`, `apply1`, `applyb1` y `apply2`. En el otro, se encuentran `let` y `letsimp`. Ambos esquemas definen patrones en términos de variables de patrones declaradas mediante `matchdeclare`.

Las reglas de comparación de patrones definidas por `tellsimp` y `tellsimpafter` se aplican automáticamente por el simplificador de Maxima. Las reglas definidas por `defmatch`, `defrule` y `let` se aplican previa llamada a una función.

Hay otros mecanismos para las reglas; las relativas a polinomios se controlan mediante `tellrat` y las del álgebra conmutativa y no conmutativa se definen en el paquete `affine`.

34.2 Funciones y variables sobre reglas y patrones

`apply1` (*expr*, *regla_1*, ..., *regla_n*) [Función]

Aplica de forma repetida la *regla_1* a *expr* hasta que falla, a continuación aplica repetidamente la misma regla a todas las subexpresiones de *expr*, de izquierda a derecha, hasta que la *regla_1* haya fallado en todas las subexpresiones. Llámese *expr_2* al resultado de transformar *expr* de esta forma. Entonces la *regla_2* se aplica de la misma manera comenzando en el nivel superior de *expr_2*. Cuando la *regla_n* falla en la última expresión, se devuelve el resultado.

`maxapplydepth` es el nivel de las subexpresiones más internas procesadas por `apply1` y `apply2`.

Véase también `applyb1`, `apply2` y `let`.

`apply2` (*expr*, *regla_1*, ..., *regla_n*) [Función]

Si la *regla_1* falla en una subexpresión dada, entonces se aplica la *regla_2* repetidamente, etc. Sólo si todas las reglas fallan en una subexpresión serán aplicadas todas las reglas de forma repetida a la siguiente subexpresión. Si alguna de las reglas tiene éxito entonces la misma subexpresión es reprocesada, comenzando por la primera regla.

`maxapplydepth` es el nivel de las subexpresiones más internas procesadas por `apply1` y `apply2`.

Véase también `applyb1` y `let`.

`applyb1` (*expr*, *regla_1*, ..., *regla_n*) [Función]

Aplica la *regla_1* reiteradamente hasta la subexpresión más interna de *expr* hasta que falle, a continuación pasa a aplicar la misma regla en un nivel superior (esto es, en subexpresiones más grandes), hasta que la *regla_1* falle en la expresión de nivel más alto. Después se aplica la *regla_2* de la misma manera al resultado obtenido de *regla_1*. Tras la aplicación de la *regla_n* a la expresión de mayor nivel, se devuelve el resultado.

La función `applyb1` es similar a `apply1` pero opera de abajo-arriba, en lugar de arriba-abajo.

`maxapplyheight` es la máxima altura a la que llega `applyb1` antes de terminar su cometido.

Véase también `apply1`, `apply2` y `let`.

`current_let_rule_package` [Variable opcional]

Valor por defecto: `default_let_rule_package`

La variable `current_let_rule_package` es el nombre del paquete de reglas que están utilizando las funciones del paquete `let` (`letsimp`, etc.), a menos que se especifique otro paquete de reglas. A esta variable se le puede asignar el nombre de cualquier paquete de reglas definido por medio de la instrucción `let`.

Si se hace la llamada `letsimp (expr, rule_pkg_name)`, el paquete de reglas `rule_pkg_name` será utilizado únicamente para esa llamada y el valor de `current_let_rule_package` no cambia.

`default_let_rule_package` [Variable opcional]

Valor por defecto: `default_let_rule_package`

La variable `default_let_rule_package` es el nombre del paquete de reglas utilizado cuando el usuario no especifica otro explícitamente con `let` o cambiando el valor de `current_let_rule_package`.

`defmatch` (*nombre_prog*, *patrón*, *x_1*, ..., *x_n*) [Función]
`defmatch` (*progname*, *pattern*) [Función]

Define una función *nombre_prog*(*expr*, *x_1*, ..., *x_n*) que analiza si *expr* coincide con el *patrón*.

El argumento *patrón* es una expresión que contiene los argumentos de patrón *x_1*, ..., *x_n* y algunas variables de patrón. Los argumentos de patrón se dan de forma explícita como argumentos a `defmatch`, mientras que las variables de patrón se declaran mediante la función `matchdeclare`. Cualquier variable no declarada bien como variable patrón en `matchdeclare`, bien como argumento patrón en `defmatch` se hace coincidir con ella misma.

El primer argumento de la función definida *nombre_prog* es una expresión a ser comparada con el patrón y los demás argumentos son los argumentos que se corresponden con las variables ficticias *x_1*, ..., *x_n* del patrón.

Si el resultado de la comparación es positivo, *nombre_prog* devuelve una lista de ecuaciones cuyos miembros izquierdos son los argumentos y variables de patrón, y cuyos miembros derechos son las subexpresiones en las que se han producido las coincidencias con patrones. A las variables de patrón, no a los argumentos, se les asignan las subexpresiones con las que coinciden. Si la comparación falla, *nombre_prog* devuelve `false`.

Un patrón literal, es decir, que no contiene ni argumentos ni variables de patrón, devuelve `true` en caso de coincidencia.

A literal pattern (that is, a pattern which contains neither pattern arguments nor pattern variables) returns `true` if the match succeeds.

Véase también `matchdeclare`, `defrule`, `tellsimp` y `tellsimpafter`.

Ejemplos:

Define una función `linearp(expr, x)` que comprueba si `expr` es de la forma `a*x + b`, donde ni `a` ni `b` contienen a `x` y `a` es no nulo. La función definida reconoce expresiones lineales respecto de cualquier variable, pues el argumento de patrón `x` es pasado a `defmatch`.

```
(%i1) matchdeclare (a, lambda ([e], e#0 and freeof(x, e)),
                    b, freeof(x));
(%o1)                          done
(%i2) defmatch (linearp, a*x + b, x);
(%o2)                         linearp
(%i3) linearp (3*z + (y + 1)*z + y^2, z);
                          2
(%o3)               [b = y , a = y + 4, x = z]
(%i4) a;
(%o4)                          y + 4
(%i5) b;
                                 2
(%o5)                           y
(%i6) x;
(%o6)                            x
```

Define una función `linearp(expr)` que comprueba si `expr` es de la forma `a*x + b`, donde ni `a` ni `b` contienen a `x` y `a` es no nulo. La función definida sólo reconoce expresiones lineales únicamente respecto de `x`, pues no se le pasa a `defmatch` nigún argumento de patrón

```
(%i1) matchdeclare (a, lambda ([e], e#0 and freeof(x, e)),
                    b, freeof(x));
(%o1)                          done
(%i2) defmatch (linearp, a*x + b);
(%o2)                         linearp
(%i3) linearp (3*z + (y + 1)*z + y^2);
(%o3)                          false
(%i4) linearp (3*x + (y + 1)*x + y^2);
                               2
(%o4)                  [b = y , a = y + 4]
```

Define una función `checklimits(expr)` que comprueba si `expr` es una integral definida.

```
(%i1) matchdeclare ([a, f], true);
(%o1)                          done
(%i2) constinterval (l, h) := constantp (h - l);
(%o2)        constinterval(l, h) := constantp(h - l)
(%i3) matchdeclare (b, constinterval (a));
(%o3)                          done
(%i4) matchdeclare (x, atom);
(%o4)                          done
```

```
(%i5) simp : false;
(%o5)                          false
(%i6) defmatch (checklimits, 'integrate (f, x, a, b));
(%o6)                       checklimits
(%i7) simp : true;
(%o7)                          true
(%i8) 'integrate (sin(t), t, %pi + x, 2*%pi + x);
                        x + 2 %pi
                       /
                       [
(%o8)                  I          sin(t) dt
                       ]
                       /
                        x + %pi
(%i9) checklimits (%);
(%o9)    [b = x + 2 %pi, a = x + %pi, x = t, f = sin(t)]
```

`defrule` (*nombre_regla*, *patrón*, *reemplazamiento*) [Función]

Define y da nombre a una regla de reemplazamiento para el patrón dado. Si la regla *nombre_regla* es aplicada a una expresión (por `apply1`, `applyb1` o `apply2`), cada subexpresión que coincida con el patrón será reemplazada por el contenido de *reemplazamiento*.

Las propias reglas pueden ser tratadas como funciones que transforman una expresión mediante una operación consistente en la búsqueda de una coincidencia y posterior aplicación de un reemplazamiento. Si la comparación falla, la función que implementa la regla devuelve `false`.

`disprule` (*nombre_regla_1*, ..., *nombre_regla_n*) [Función]

`disprule` (*all*) [Función]

Muestra las reglas de *nombre_regla_1*, ..., *nombre_regla_n*, tal como son devueltas por `defrule`, `tellsimp` o `tellsimpafter`, o un patrón definido por `defmatch`. Cada regla se muestra con una etiqueta de expresión intermedia (`%t`).

La llamada `disprule (all)` muestra todas las reglas.

La función `disprule` no evalúa sus argumentos y devuelve la lista de etiquetas de expresiones intermedias correspondientes a las reglas mostradas.

Véase también `letrules`, que muestra las reglas definidas por `let`.

Ejemplos:

```
(%i1) tellsimpafter (foo (x, y), bar (x) + baz (y));
(%o1)                    [foorule1, false]
(%i2) tellsimpafter (x + y, special_add (x, y));
(%o2)                    [+rule1, simplus]
(%i3) defmatch (quux, mumble (x));
(%o3)                          quux
(%i4) disprule (foorule1, "+rule1", quux);
(%t4)        foorule1 : foo(x, y) -> baz(y) + bar(x)
```

```
(%t5)          +rule1 : y + x -> special_add(x, y)

(%t6)                quux : mumble(x) -> []

(%o6)                     [%t4, %t5, %t6]
(%i6) ''%;
(%o6) [foorule1 : foo(x, y) -> baz(y) + bar(x),
     +rule1 : y + x -> special_add(x, y), quux : mumble(x) -> []]
```

`let` (*prod*, *repl*, *predname*, *arg_1*, ..., *arg_n*) [Función]
`let` ([*prod*, *repl*, *predname*, *arg_1*, ..., *arg_n*], *nombre_paquete*) [Función]

Define una regla de sustitución para `letsimp` tal que *prod* es sustituido por *repl*, donde *prod* es un producto de potencias positivas o negativas de los términos siguientes:

- Átomos que `letsimp` buscará a menos que antes de llamar a `letsimp` se utilice la función `matchdeclare` para asociar un predicado con el átomo. En este caso `letsimp` hará coincidir el átomo con cualquier término del producto que satisfaga el predicado.
- Expresiones básicas como `sin(x)`, `n!`, `f(x,y)`, etc. Como en el caso anterior, `letsimp` buscará coincidencias exactas, a menos que se utilice `matchdeclare` para asociar un predicado con el argumento de la expresión básica (`sin(x)`, `n!`, `f(x,y)`, ...).

Si se incluye un predicado en la función `let` seguido de una lista de argumentos, una coincidencia aceptable (es decir, una que fuese aceptada si se hubiese omitido el predicado) se aceptará sólo si `predname (arg_1', ..., arg_n')` vale `true`, donde *arg_i*' es el valor coincidente con *arg_i*. El argumento *arg_i* puede ser el nombre de cualquier átomo o el argumento de cualquier expresión básica que aparezca en *prod*. *repl* puede ser cualquier expresión racional. Si cualquiera de los átomos o argumentos de *prod* aparece en *repl* se llevan a cabo las sustituciones correspondientes.

La variable global `letrat` controla la simplificación de los cocientes por `letsimp`. Cuando `letrat` vale `false`, `letsimp` simplifica separadamente el numerador y denominador de *expr* y no simplifica el cociente. Sustituciones como que `n!/n` se reduzca a `(n-1)!` ya no se realizarán. Cuando `letrat` vale `true`, entonces se simplifican el numerador, el denominador y el cociente, en este orden.

Estas funciones de sustitución permiten al usuario trabajar con varios paquetes de reglas al mismo tiempo. Cada paquete de reglas puede contener cierto número de reglas `let` que son referenciadas por un nombre dado por el usuario. `let ([`*prod*`, `*repl*`, `*predname*`, `*arg_1*`, ..., `*arg_n*`], `*nombre_paquete*`)` añade la regla *predname* al paquete de reglas *nombre_paquete*. `letsimp (`*expr*`, `*package_name*`)` aplica las reglas de *nombre_paquete*. La llamada `letsimp (`*expr*`, `*nombre_paquete1*`, `*nombre_paquete2*`, ...)` es equivalente a `letsimp (`*expr*`, `*nombre_paquete1*`)` seguida de `letsimp (%, `*nombre_paquete2*`)`, ...

`current_let_rule_package` es el nombre del paquete de reglas que se está utilizando. A esta variable se le puede asignar el nombre de cualquier paquete de reglas definido mediante el comando `let`. Siempre que una de las funciones incluidas en el paquete `let` sean invocadas sin nombre de paquete, se utilizará el paquete cuyo nombre se guarde en `current_let_rule_package`. Si se hace una llamada tal como `letsimp`

(*expr*, *rule_pkg_name*), el paquete de reglas *rule_pkg_name* es utilizado solamente para ese comando `letsimp`, sin efectuarse cambios en `current_let_rule_package`. A menos que se indique otra cosa, `current_let_rule_package` toma por defecto el valor de `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` [Variable opcional]

Valor por defecto: `false`

Cuando `letrat` vale `false`, `letsimp` simplifica separadamente el numerador y denominador de una fracción sin simplificar luego el cociente.

Cuando `letrat` vale `true`, se simplifican el numerador, denominador y cociente, por este orden.

```
(%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 ()` [Función]

`letrules (`*nombre_paquete*`)` [Función]

Muestra las reglas de un paquete de reglas. La llamada `letrules ()` muestra las reglas del paquete de reglas actual. La llamada `letrules (`*nombre_paquete*`)` muestra las reglas de *nombre_paquete*.

El paquete de reglas actual tiene su nombre almacenado en by `current_let_rule_package`. A menos que se indique de otra manera, `current_let_rule_package` toma por defecto el valor de `default_let_rule_package`.

Véase también `disprule`, que muestra las reglas definidas por `tellsimp` y `tellsimpafter`.

`letsimp (`*`expr`*`)` [Función]
`letsimp (`*`expr`*`, `*`nombre_paquete`*`)` [Función]
`letsimp (`*`expr`*`, `*`nombre_paquete_1`*`, ..., `*`nombre_paquete_n`*`)` [Función]

Aplica repetidamente las reglas definidas por `let` hasta que no se puedan hacer más cambios en *expr*.

La llamada `letsimp (`*`expr`*`)` utiliza las reglas de `current_let_rule_package`.

La llamada `letsimp (`*`expr`*`, `*`nombre_paquete`*`)` utiliza las reglas de *nombre_paquete* sin efectuar cambios en `current_let_rule_package`.

La llamada `letsimp (`*`expr`*`, `*`nombre_paquete_1`*`, ..., `*`nombre_paquete_n`*`)` es equivalente a `letsimp (`*`expr`*`, `*`nombre_paquete_1`*`, seguida de `letsimp (%, `*`nombre_paquete_2`*`)` y así sucesivamente.

`let_rule_packages` [Variable opcional]

Valor por defecto: `[default_let_rule_package]`

La variable `let_rule_packages` guarda una lista con todos los paquetes de reglas definidos por el usuario, junto con el paquete por defecto `default_let_rule_package`.

`matchdeclare (`*`a_1`*`, `*`pred_1`*`, ..., `*`a_n`*`, `*`pred_n`*`)` [Función]

Asocia un predicado *pred_k* con una variable o lista de variables *a_k*, de forma que *a_k* se comparará con expresiones para las cuales el predicado devuelva algo que no sea `false`.

Un predicado puede ser el nombre de una función, una expresión lambda, una llamada a función, una llamada a una expresión lambda sin el último argumento, `true` o `all`. Cualquier expresión se hace coincidir con `true` o `all`.

Si el predicado se especifica como una llamada a función o a una expresión lambda, la expresión a ser analizada es añadida a la lista de argumentos, siendo los argumentos evaluados en el momento de ser evaluada la comparación. En cambio, si el predicado se especifica como un nombre de función o como una expresión lambda, la expresión a ser analizada será su único argumento. No es necesario definir una función de predicado cuando se hace una llamada a `matchdeclare`; el predicado no se evalúa hasta que se ensaya una comparación.

Un predicado puede devolver tanto una expresión booleana, como `true` o `false`. Las expresiones booleanas se evalúan con `is` dentro de la regla, por lo que no es necesario llamar a `is` desde dentro del predicado.

Si una expresión satisface un predicado, se asigna a la variable de comparación la expresión, excepto cuando las variables de comparación son operandos de sumas `+` o multiplicaciones `*`. Solamente las sumas y multiplicaciones son tratadas de forma especial; los demás operadores n-arios (tanto los del sistema como los definidos por el usuario) son tratados como funciones ordinarias.

En el caso de sumas y multiplicaciones, a la variable de comparación se le puede asignar una expresión simple que satisfaga el predicado de comparación, o una suma o producto, respectivamente, de tales expresiones. Los predicados son evaluados en el orden en el que sus variables asociadas aparecen en el patrón de comparación, y un término que satisfaga más de un predicado es tomado por el primer predicado que satisfaga. Cada predicado se compara con todos los operandos de la suma o producto antes de ser evaluado el siguiente predicado. Además, si 0 o 1, respectivamente, satisface un predicado de comparación, y no hay otros términos que lo satisfagan, se asignará el 0 o 1 a la variable de comparación asociada al predicado.

El algoritmo para procesar patrones de suma y multiplicación hace que los resultados de algunas comparaciones dependan del orden de los términos en el patrón de comparación y en la expresión a ser comparada. Sin embargo, si todos los predicados de comparación son mutuamente excluyentes, el resultado de la comparación no depende para nada de la ordenación, puesto que un predicado de comparación no puede aceptar términos aceptados por otros predicados.

Invocando `matchdeclare` con una variable *a* como argumento cambia la propiedad de `matchdeclare` para *a*, si ya había una declarada; solamente el `matchdeclare` más reciente está activo cuando se define una regla. Cambios posteriores en la propiedad de `matchdeclare` (via `matchdeclare` o `remove`) no afectan a las reglas existentes.

`propvars (matchdeclare)` devuelve la lista de todas las variables para las cuales hay una propiedad de `matchdeclare`. La llamada `printprops (a, matchdeclare)` devuelve el predicado para la variable `a`. La llamada `printprops (all, matchdeclare)` devuelve la lista de predicados de todas las variables de `matchdeclare`. La llamada `remove (a, matchdeclare)` borra la propiedad `matchdeclare` de *a*.

Las funciones `defmatch`, `defrule`, `tellsimp`, `tellsimpafter` y `let` construyen reglas que analizan expresiones mediante patrones.

`matchdeclare` no evalúa sus argumentos y siempre devuelve `done`.

Ejemplos:

Un predicado puede ser el nombre de una función, una expresión lambda, una llamada a función, una llamada a una expresión lambda sin el último argumento, `true` o `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
```

Si una expresión satisface un predicado, se asigna a la variable de comparación la expresión.

```
(%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]
```

En el caso de sumas y multiplicaciones, a la variable de comparación se le puede asignar una expresión simple que satisfaga el predicado de comparación, o una suma o producto, respectivamente, de tales expresiones.

```
(%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)]
```

`maxapplydepth` [Variable opcional]

Valor por defecto: 10000

La variable `maxapplydepth` es la máxima profundidad a la que van a introducirse `apply1` y `apply2`.

`maxapplyheight` [Variable opcional]

Valor por defecto: 10000

La variable `maxapplyheight` es la m2'axima altura a la que escalará `applyb1` antes de detenerse.

`remlet` (*prod*, *nombre*) [Función]
`remlet` () [Función]
`remlet` (*all*) [Función]
`remlet` (*all*, *nombre*) [Función]

Elimina la última regla de sustitución *prod* –> repl que haya sido definida por la función `let`. Si se suministar el nombre la regla será borrada del paquete con ese mismo nombre.

Las llamadas `remlet()` y `remlet(all)` eliminan todas las reglas de sustitución del paquete de reglas actual. Si se suministra el nombre de un paquete de reglas, como en `remlet (all, nombre)`, el paquete de reglas con ese *nombre* es también eliminado.

Si es necesario cambiar una sustitución haciendo uso de la misma producción, no es necesario llamar a `remlet`, simplemente redefínase la sustitución utilizando la misma

producción con la función `let` junto con el nuevo reemplazamiento y/o nombre de predicado. De ser llamado nuevamente `remlet (`*`prod`*`)` la sustitución original sería recuperada.

Véase también `remrule`, que elimina una regla definida por `tellsimp` o `tellsimpafter`.

`remrule` (*op*, *nombre_regla*) [Función]
`remrule` (*op*, *all*) [Función]

Elimina las reglas previamente definidas por `tellsimp` o `tellsimpafter`.

La llamada `remrule (`*`op`*`, `*`nombre_regla`*`)` elimina la regla de nombre *nombre_regla* del operador *op*.

Independientemente de que *op* sea un operador propio de Maxima o haya sido definido por el usario (como los establecidos por `infix`, `prefix`, etc.), tanto *op* como *rulename* deben ir encerrados entre comillas dobles.

La llamada `remrule (`*`function`*`, all)` borra todas las reglas para el operador *op*.

Véase también `remlet`, que elimina una regla definida mediante `let`.

Ejemplos:

```
(%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` (*patrón*, *reemplazamiento*) [Función]

La función `tellsimp` es similar a `tellsimpafter` pero coloca nueva información antes que la antigua, de manera que se aplica antes que las reglas de simplificación de Maxima.

La función `tellsimp` se utiliza cuando es importante utilizar la expresión antes de que el simplificador opere sobre ella; por ejemplo, cuando el simplificador ya "sabe" algo sobre una expresión, pero lo que devuelve no es lo que quiere el usuario. En cambio, cuando el simplificador ya "sabe" algo sobre una expresión pero lo que devuelve no es lo suficiente para el usuario, entonces éste podrá estar interesado en utilizar `tellsimpafter`.

El patrón no puede ser una suma, ni un producto, ni una variable ni un número.

`rules` es la lista de reglas definidas por `defrule`, `defmatch`, `tellsimp` y `tellsimpafter`.

Ejemplos:

```
(%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` (*patrón*, *reemplazamiento*) [Función]

Define una regla de simplificación que el simplificador aplicará después de las reglas de simplificación propias de de Maxima. El *patrón* es una expresión que contiene variables de patrón (declaradas por `matchdeclare`) junto con otros átomos y operadores. El contenido de *reemplazamiento* sustituye una expresión que coincida con el patrón; a las variables de patrón en *reemplazamiento* se les asignan los valores coincidentes en la expresión.

El *patrón* puede ser una expresión no atómica en la que el operador principal no sea una variable de patrón; la regla de simplificación se asocia con el operador principal. Los nombres de las funciones (con una excepción que se indica más abajo), listas y arrays pueden aparecer en el *patrón* como operador principal sólo como literales (no variables de patrones); esto excluye expresiones como `aa(x)` y `bb[y]`, si tanto `aa` como `bb` son patrones de variables. Nombres de funciones, listas y arrays que sean variables de patrón pueden aparecer como operadores que no sean el operador principal de *patrón*.

Hay una excepción a la regla indicada más arriba concerniente a los nombres de funciones. El nombre de una función subindicada en una expresión tal como `aa[x](y)` puede ser una variable de patrón porque el operador principal no es `aa` sino el átomo de Lisp `mqapply`. Esta es una consecuencia de la representación de expresiones que contienen funciones subindicadas.

Las reglas de simplificación se aplican tras las evaluaciones (a menos que se supriman con el apóstrofo o la variable `noeval`). Las reglas establecidas por `tellsimpafter` se aplican en el orden en que han sido definidas y después de las reglas propias de Maxima. Las reglas se aplican de abajo arriba, esto es, se aplican primero a las subexpresiones antes que a toda la expresión. Puede ser necesario simplificar repetidamente un resultado (por ejemplo, mediante el operador de doble comilla simple `''` o la variable `infeval`) para asegurar que se aplican todas las reglas.

Las variables de patrón se tratan como variables locales en las reglas de simplificación. Una vez definida una regla, el valor de una variable de patrón no afecta a la regla, ni se ve influenciada poe ésta. Una asignación a una variable de patrón que resulta de la aplicación exitosa de una regla no afecta a la asignación actual de la variable de patrón. Sin embargo, como cualquier otro átomo de Maxima, las propiedades de las variables de patrón (tal como se definen con `put` y sus funciones relacionadas) son globales.

La regla construida por `tellsimpafter` es nombrada detrás del operador principal de *patrón*. Reglas para operadores de Maxima y operadores definidos por el usuario con `infix`, `prefix`, `postfix`, `matchfix` y `nofix`, tienen nombres que son cadenas alfanuméricas de Maxima. Reglas para otras funciones tienen nombres que son identificadores ordinarios de Maxima.

El tratamiento de formas nominales y verbales es hasta cierto punto confuso. Si se define una regla para una forma nominal (o verbal) y ya existe una regla para la correspondiente forma verbal (o nominal), la regla recién definida se aplica a ambas formas (nominal y verbal). Si no existe regla para una forma verbal (o nominal) la regla recién definida se aplica únicamente a la forma nominal (o verbal).

La regla construida por `tellsimpafter` es una típica función de Lisp. Si el nombre de la regla es `$foorule1`, la sentencia `:lisp (trace $foorule1)` hace una traza de la función y `:lisp (symbol-function '$foorule1)` muestra su definición.

La función `tellsimpafter` no evalúa sus argumentos y devuelve la lista de reglas para el operador principal de *patrón*, incluida la regla recién establecida.

Véanse también `matchdeclare`, `defmatch`, `defrule`, `tellsimp`, `let`, `kill`, `remrule` y `clear_rules`.

Ejemplos:

pattern puede ser cualquier expresión no atómica en la que el operador principal no sea una variable de patrón.

```
(%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))
```

Las reglas se aplican en el orden en que se definen. Si dos reglas coinciden con una expresión, se aplica aquélla que haya sido definida en primer lugar.

```
(%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)
```

Las variables de patrón se tratan como variables locales en las reglas de simplificación. (Compárese con `defmatch`, que trata las variables de patrón como globales.)

```
(%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 cualquier otro átomo, las propiedades de las variables de patrón son globales, incluso cuando sus valores sean locales. En este ejemplo se declara una propiedad de asignación a treavés de `define_variable`. Esta es una propiedad del átomo `bb` en todo 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);
```

Las reglas se nombran después de los operadores principales. Los nombres de reglas tanto para las funciones de Maxima como para las definidas por el usuario son cadenas alfanuméricas, mientras que los nombres de las otras funciones son identificadores típicos.

```
(%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)                            ^
```

Un ejemplo de producto anticonmutativo.

```
(%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` () [Función]

Ejecuta `kill (rules)` y después inicializa el siguiente número de regla a 1 para la adición `+`, multiplicación `*` y exponenciación `^`.

35 Conjuntos

35.1 Introducción a los conjuntos

Maxima dispone de funciones para realizar operaciones con conjuntos, como la intersección o la unión. Los conjuntos deben ser finitos y definidos por enumeración. Maxima trata a los conjuntos y a las listas como objectos de distinta naturaleza, lo que permite trabajar con conjuntos cuyos elementos puedan ser también conjuntos o listas.

Además de funciones para operar con conjuntos finitos, Maxima dispone también de algunas funciones sobre combinatoria, como los números de Stirling de primera y segunda especie, números de Bell, coeficientes multinomiales, particiones de enteros no negativos y algunos otros. Maxima también define la función delta de Kronecker.

35.1.1 Utilización

Para construir un conjunto cuyos elementos sean `a_1, ..., a_n`, se utiliza la instrucción `set(a_1, ..., a_n)` o `{a_1, ..., a_n}`; para formar un conjunto vacío, basta con hacer `set()` o `{}`. Para introducir conjuntos en Maxima, `set (...)` y `{ ... }` son equivalentes. Los conjuntos se muestran siempre con llave.

Si un elemento se indica más de una vez, el proceso de simplificación elimina los elementos redundantes.

```
(%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]}
```

Dos elementos candidatos a formar parte de un conjunto, x e y, son redundantes, esto es, se consideran el mismo elemento a efectos de consruir el conjunto, si y sólo si `is (x = y)` devuelve el valor `true`. Nótese que `is (equal (x, y))` puede devolver `true` y `is (x = y)` retornar `false`; en cuyo caso los elementos x e y se considerarían 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 formar un conjunto a partir de los miembros de una lista úsese `setify`.

```
(%i1) setify([b, a]);
(%o1)                           {a, b}
```

Los elementos `x` e `y` de un conjunto se consideran iguales si `is(x = y)` devuelve el valor `true`. Así, `rat(x)` y `x` se consideran el mismo elemento de un conjunto; consecuentemente,

```
(%i1) {x, rat(x)};
(%o1)                             {x}
```

Además, puesto que `is((x-1)*(x+1) = x^2 - 1)` devuelve `false`, `(x-1)*(x+1)` y `x^2-1` se consideran elementos diferentes; así

```
(%i1) {(x - 1)*(x + 1), x^2 - 1};
                                          2
(%o1)                {(x - 1) (x + 1), x  - 1}
```

Para reducir este conjunto a otro unitario, aplicar `rat` a cada elemento del 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 eliminar redundancias con otros conjuntos, será necesario utilizar otras funciones de simplificación. He aquí un ejemplo que utiliza `trigsimp`:

```
(%i1) {1, cos(x)^2 + sin(x)^2};
```

```
                              2         2
(%o1)                {1, sin (x) + cos (x)}
(%i2) map (trigsimp, %);
(%o2)                            {1}
```

Se entiende que un conjunto está simplificado cuando entre sus elementos no hay redundancias y se hayan ordenado. La versión actual de las funciones para conjuntos utiliza la función `orderlessp` de Maxima para ordenar sus elementos; sin embargo, *futuras versiones de las funciones para operar con conjuntos podrán utilizar otras funciones de ordenación.*

Algunas operaciones con conjuntos, tales como la sustitución, fuerzan automáticamente una re-simplificación; por ejemplo,

```
(%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 considera a las listas y conjuntos como objetos diferentes; funciones tales como `union` y `intersection` emitirán un error si alguno de sus argumentos no es un conjunto. Si se necesita aplicar una función de conjunto a una lista, se deberá utilizar la función `setify` para convertirla previamente en conjunto. Así,

```
(%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 extraer todos los elementos de un conjunto `s` que satisfagan un predicado `f`, úsese `subset(s,f)`. (Un *predicado* es una función booleana.) Por ejemplo, para encontrar las ecuaciones en un conjunto dado que no dependan de la variable `z`, se hará

```
(%i1) subset ({x + y + z, x - y + 4, x + y - 5},
               lambda ([e], freeof (z, e)));
(%o1)                {- y + x + 4, y + x - 5}
```

La sección `Funciones y variables para los conjuntos` incluye una lista completa de funciones para operar con conjuntos en Maxima.

35.1.2 Iteraciones con elementos

Hay dos formas para operar iterativamente sobre los elementos de un conjunto. Una es utilizar `map`; por ejemplo:

```
(%i1) map (f, {a, b, c});
(%o1)                    {f(a), f(b), f(c)}
```

La otra forma consiste en hacer uso de la construcción `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
```

Las funciones de Maxima `first` y `rest` funcionan también con conjuntos. En este caso, `first` devuelve el primer elemento que se muestra del conjunto, el cual puede depender de la implementación del sistema. Si `s` es un conjunto, entonces `rest(s)` equivale a `disjoin (first(s), s)`. Hay otras funciones que trabajan correctamente con conjuntos. En próximas versiones de las funciones para operar con conjuntos es posible que `first` y `rest` trabajen de modo diferente o que ya no lo hagan en absoluto.

35.1.3 Fallos

Las funciones para operar con conjuntos utilizan la función `orderlessp` de Maxima para ordenar los elementos de los conjuntos, así como la función `like` de Lisp para decidir sobre la igualdad de dichos elementos. Ambas funciones tienen fallos que son conocidos y que pueden aflorar si se trabaja con conjuntos que tengan elementos en formato de listas o matrices y que contengan expresiones racionales canónicas (CRE). Un ejemplo es

```
(%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.
```

Esta expresión provoca una parada de Maxima junto con la emisión de un mensaje de error, el cual dependerá de la versión de Lisp que utilice Maxima. Otro ejemplo es

```
(%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.
```

Estos fallos son causados por fallos en `orderlessp` y `like`, no por fallos cuyo origen se encuentre en las funciones para conjuntos. Para ilustrarlo, se pueden ejecutar las siguientes expresiones

```
(%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
```

Hasta que estos errores no se corrijan, no es aconsejable construir conjuntos que tengan por elementos listas o matrices que contengan expresiones en forma CRE; sin embargo, un conjunto con elementos de la forma CRE no deberían dar problemas:

```
(%i1) {x, rat (x)};
(%o1)                          {x}
```

La función `orderlessp` de Maxima tiene otro fallo que puede causar problemas con las funciones para conjuntos, en concreto, que el predicado de ordenación `orderlessp` no es transitivo. El ejemplo más simple que ilustra este punto es

```
(%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
```

El fallo puede causar problemas con todas las funciones para conjuntos, así como también con otras funciones de Maxima. Es probable, pero no seguro, que este fallo se puede evitar si todos los elementos del conjunto están en la forma de expresión racional canónica (CRE) o han sido simplificados con `ratsimp`.

Los mecanismos `orderless` y `ordergreat` de Maxima son incompatibles con las funciones para conjuntos. Si se necesitan utilizar `orderless` o `ordergreat`, hágase antes de construir los conjuntos y no se utilice la instrucción `unorder`.

Se ruega a todo usuario que crea haber encontrado un fallo en las funciones para conjuntos que lo comunique en la base de datos de Maxima. Véase `bug_report`.

35.1.4 Autores

Stavros Macrakis de Cambridge, Massachusetts y Barton Willis de la University of Nebraska at Kearney (UNK).

35.2 Funciones y variables para los conjuntos

`adjoin` (*x*, *a*) [Función]

Calcula la unión del conjunto *a* y `{`*x*`}`.

La función `adjoin` emite un mensaje de error si *a* no es un conjunto literal.

Las sentencias `adjoin(`*x*`, a)` y `union(set(`*x*`), a)` son equivalentes, aunque `adjoin` puede ser algo más rápida que `union`.

Véase también `disjoin`.

Ejemplos:

```
(%i1) adjoin (c, {a, b});
(%o1)                        {a, b, c}
(%i2) adjoin (a, {a, b});
(%o2)                         {a, b}
```

belln (*n*) [Función]

Representa el *n*-ésimo número de Bell, de modo que `belln(n)` es el número de particiones de un conjunto de *n* elementos.

El argumento *n* debe ser un entero no negativo.

La función `belln` se distribuye sobre ecuaciones, listas, matrices y conjuntos.

Ejemplos:

`belln` se aplica a enteros no 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
```

Si *n* no es un entero no negativo, la función `belln(n)` no hace cálculo alguno.

```
(%i1) [belln (x), belln (sqrt(3)), belln (-9)];
(%o1)        [belln(x), belln(sqrt(3)), belln(- 9)]
```

cardinality (*a*) [Función]

Devuelve el número de elementos del conjunto *a*.

La función `cardinality` ignora los elementos redundantes, incluso cuando la simplificación está desabilitada.

Ejemplos:

```
(%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*) [Función]

Devuelve un conjunto formado por listas de la forma `[x_1, ..., x_n]`, siendo *x_1*, ..., *x_n* elementos de los conjuntos *b_1*, ... , *b_n*, respectivamente.

La función `cartesian_product` emite un mensaje de error si alguno de sus argumentos no es un conjunto literal.

Ejemplos:

```
(%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*) [Función]

Devuelve el conjunto *a* sin el elemento *x*. Si *x* no es elemento de *a*, entonces el resultado es el propio *a*.

La función `disjoin` emite un mensaje de error si *a* no es un conjunto literal.

Las sentencias `disjoin(x, a)`, `delete(x, a)` y `setdifference(a, set(x))` son todas ellas equivalentes; pero en general, `disjoin` será más rápida que las otras.

Ejemplos:

```
(%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*) [Función]

Devuelve `true` si y sólo si los conjuntos *a* y *b* son disjuntos.

La función `disjointp` emite un mensaje de error si *a* o *b* no son conjuntos literales.

Ejemplos:

```
(%i1) disjointp ({a, b, c}, {1, 2, 3});
(%o1)                           true
(%i2) disjointp ({a, b, 3}, {1, 2, 3});
(%o2)                           false
```

`divisors` (*n*) [Función]

Calcula el conjunto de divisores de *n*.

La sentencia `divisors(n)` devuelve un conjunto de enteros si *n* es un entero no nulo. El conjunto de divisores incluye los elementos 1 y *n*. Los divisores de un entero negativo son los divisores de su valor absoluto.

La función `divisors` se distribuye sobre las ecuaciones, listas, matrices y conjuntos.

Ejemplos:

Se puede comprobar que 28 es un número perfecto: la suma de sus divisores (excepto él mismo) es 28.

```
(%i1) s: divisors(28);
(%o1)                 {1, 2, 4, 7, 14, 28}
(%i2) lreduce ("+", args(s)) - 28;
(%o2)                           28
```

La función `divisors` es simplificadora. Haciendo la sustitución de `a` por 8 en `divisors(a)` devuelve los divisores sin tener que reevaluar `divisors(8)`,

```
(%i1) divisors (a);
(%o1)                       divisors(a)
(%i2) subst (8, a, %);
(%o2)                      {1, 2, 4, 8}
```

La función `divisors` se distribuye sobre ecuaciones, listas, matrices y 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*) [Función]

Devuelve `true` si y sólo si *x* es miembro del conjunto *a*.

La función `elementp` emite un mensaje de error si *a* no es un conjunto literal.

Ejemplos:

```
(%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*) [Función]

Devuelve `true` si y sólo si *a* es el conjunto vacío o la lista vacía.

Ejemplos:

```
(%i1) map (emptyp, [{}, []]);
(%o1)                      [true, true]
(%i2) map (emptyp, [a + b, {{}}, %pi]);
(%o2)                 [false, false, false]
```

`equiv_classes` (*s*, *F*) [Función]

Devuelve el conjunto de las clases de equivalencia del conjunto *s* respecto de la relación de equivalencia *F*.

El argumento *F* es una función de dos variables definida sobre el producto cartesiano *s* por *s*. El valor devuelto por *F* debe ser `true` o `false`, o bien una expresión *expr* tal que `is(`*expr*`)` tome el valor `true` o `false`.

Si *F* no es una relación de equivalencia, `equiv_classes` la acepta sin emitir ningún mensaje de error, pero el resultado será incorrecto en general.

Ejemplos:

La relación de equivalencia es una expresión lambda que devuelve `true` o `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}}
```

La relación de equivalencia es el nombre de una función relacional en la que `is` evalúa a `true` o `false`,

```
(%i1) equiv_classes ({1, 1.0, 2, 2.0, 3, 3.0}, equal);
(%o1)            {{1, 1.0}, {2, 2.0}, {3, 3.0}}
```

Las clases de equivalencia son números que difieren en un múltiplo de 3.

```
(%i1) equiv_classes ({1, 2, 3, 4, 5, 6, 7},
                     lambda ([x, y], remainder (x - y, 3) = 0));
(%o1)                {{1, 4, 7}, {2, 5}, {3, 6}}
```

`every` (*f*, *s*) [Función]
`every` (*f*, *L_1*, ..., *L_n*) [Función]

Devuelve `true` si el predicado *f* vale `true` para todos los argumentos dados.

Dado un conjunto como segundo argumento, `every(f, s)` devuelve `true` si `is(f(a_i))` devuelve `true` para todos los *a_i* pertenecientes *s*. La función `every` puede evaluar o no *f* para todos los *a_i* pertenecientes *s*. Puesto que los conjuntos no están ordenados, `every` puede evaluar `f(a_i)` en cualquier orden.

Dada una o más listas como argumentos, `every(f, L_1, ..., L_n)` devuelve `true` si `is(f(x_1, ..., x_n))` devuelve `true` para todo *x_1*, ..., *x_n* en *L_1*, ..., *L_n*, respectivamente. La función `every` puede evaluar o no *f* para cualquier combinación de *x_1*, ..., *x_n*; además, `every` evalúa las listas en el orden creciente del índice.

Dado un conjunto vacío `{}` o lista vacía `[]` como argumentos, `every` devuelve `false`.

Si la variable global `maperror` vale `true`, todas las listas *L_1*, ..., *L_n* deben ser de igual longitud. Si `maperror` vale `false`, los argumentos en forma de listas se truncan para igualar sus longitudes a la de la lista más corta.

Los valores que devuelve el predicado *f* cuando toman (mediante `is`) un valor diferente a `true` y `false` se controlan con la variable global `prederror`. Si `prederror` vale `true`, tales valores se consideran como `false` y la respuesta de `every` es `false`. Si `prederror` vale `false`, tales valores se consideran como desconocidos (`unknown`) y la respuesta de `every` es `unknown`.

Ejemplos:

Se aplica `every` a un único conjunto. El predicado es una función de un 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
```

Se aplica `every` a dos listas. El predicado es una función de dos argumentos.

```
(%i1) every ("=", [a, b, c], [a, b, c]);
(%o1)                        true
(%i2) every ("#", [a, b, c], [a, b, c]);
(%o2)                        false
```

Las respuestas del predicado *f* que se evalúan a cualquier cosa diferente de `true` y `false` están controlados por la variable 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*) [Función]
`extremal_subset` (*s*, *f*, *min*) [Función]

Calcula el subconjunto de *s* para el cual la función *f* toma sus valores mayor y menor.

La sentencia `extremal_subset(s, f, max)` devuelve el subconjunto del conjunto o lista *s* para el cual la función real *f* toma su valor máximo.

La sentencia `extremal_subset(s, f, min)` devuelve el subconjunto del conjunto o lista *s* para el cual la función real *f* toma su valor mínimo.

Ejemplos

```
(%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*) [Función]

Recoge los argumentos de subexpresiones con el mismo operador que *expr* y construye con ellas otra expresión a partir de estos argumentos.

Aquellas subexpresiones en las que el operador es diferente del operador principal de `expr` se copian sin modificarse, incluso cuando ellas mismas contengan subexpresiones en las que el operador sea el mismo que el de `expr`.

Es posible que `flatten` construya expresiones en las que el número de argumentos difiera del número admitido por el operador, lo cual hará que se emita un mensaje de error. La función `flatten` no intentará detectar estas situaciones.

Las expresiones que tengan representaciones especiales, por ejemplo las racionales canónicas (CRE), no admiten que se aplique sobre ellas la función `flatten`; en tales casos se devuelve el argumento sin modificación.

Ejemplos:

Aplicada a una lista, `flatten` reune todos los elementos que son a su vez 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 un conjunto, `flatten` reune todos los elementos que son a su vez conjuntos.

```
(%i1) flatten ({a, {b}, {{c}}});
(%o1)                         {a, b, c}
(%i2) flatten ({a, {[a], {a}}});
(%o2)                         {a, [a]}
```

La función `flatten` es similar a la declaración del operador principal como n-ario. Sin embargo, `flatten` no tiene efecto alguno sobre subexpresiones que tengan un operador diferente del principal, mientras que sí lo tiene una declaración n-aria.

```
(%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)))
```

La función `flatten` trata las funciones subindicadas como a cualquier otro operador.

```
(%i1) flatten (f[5] (f[5] (x, y), z));
(%o1)                       f (x, y, z)
                             5
```

Es posible que `flatten` construya expresiones en las que el número de argumentos difiera del número admitido por el 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)` [Función]

Sustituye los operadores de conjunto presentes en *a* por operadores de listas, devolviendo el resultado. La función `full_listify` sustituye operadores de conjuntos en subexpresiones anidadas, incluso cuando el operador principal no es `set`.

La función `listify` sustituye únicamente el operador principal.

Ejemplos:

```
(%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)` [Función]

Si *a* es una lista, sustituye el operador de lista por el de conjunto, aplicando posteriormente `fullsetify` a todos los elementos que son a su vez conjuntos. Si *a* no es una lista, se devuelve sin cambio alguno.

La función `setify` sustituye solamente el operador principal.

Ejemplos:

En la salida `(%o2)` el argumento de `f` no se convierte en conjunto porque el operador principal de `f([b])` no es una lista.

```
(%i1) fullsetify ([a, [a]]);
(%o1)                         {a, {a}}
(%i2) fullsetify ([a, f([b])]);
(%o2)                        {a, f([b])}
```

`identity (x)` [Función]

La función `identity` devuelve su argumento cualquiera que sea éste.

Ejemplos:

La función `identity` puede utilizarse como predicado cuando los argumentos ya son valores booleanos.

```
(%i1) every (identity, [true, true]);
(%o1)                         true
```

`integer_partitions` (*n*) [Función]
`integer_partitions` (*n*, *len*) [Función]

Devuelve particiones enteras de *n*, esto es, listas de enteros cuyas sumas son *n*.

La sentencia `integer_partitions(n)` devuelve el conjunto de todas las particiones del entero *n*. Cada partición es una lista ordenada de mayor a menor.

La sentencia `integer_partitions(n, len)` devuelve todas las particiones de longitud *len* o menor; en este caso, se añaden ceros a cada partición con menos de *len* términos para que todas ellas sean de longitud *len*. Las particiones son listas ordenadas de mayor a menor.

Una lista $[a_1, ..., a_m]$ es una partición de un entero no negativo n si (1) cada a_i es entero no nulo y (2) $a_1 + ... + a_m = n$. Así, 0 no tiene particiones.

Ejemplos:

```
(%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 las particiones que satisfagan cierta condición, utilícese la función `subset`; he aquí un ejemplo que encuentra todas las particiones de 10 formadas por 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*) [Función]

Es una forma abreviada de la función `intersection`.

`intersection` (*a_1*, ..., *a_n*) [Función]

Devuelve el conjunto de todos los elementos que son comunes a los conjuntos *a_1* a *a_n*.

Emite un mensaje de error en caso de que cualquiera de los *a_i* no sea un conjunto.

Ejemplos:

```
(%i1) S_1 : {a, b, c, d};
(%o1)                     {a, b, c, d}
(%i2) S_2 : {d, e, f, g};
(%o2)                     {d, e, f, g}
(%i3) S_3 : {c, d, e, f};
(%o3)                     {c, d, e, f}
(%i4) S_4 : {u, v, w};
(%o4)                       {u, v, w}
(%i5) intersection (S_1, S_2);
(%o5)                           {d}
(%i6) intersection (S_2, S_3);
(%o6)                       {d, e, f}
(%i7) intersection (S_1, S_2, S_3);
(%o7)                           {d}
(%i8) intersection (S_1, S_2, S_3, S_4);
(%o8)                           {}
```

`kron_delta` (*x1*, *y1*, ..., *xp*, *yp*) [Función]

Es la función delta de Kronecker.

La función `kron_delta` devuelve 1 cuando *xi* y *yi* son iguales para todos los pares, devolviendo 0 si existe un par en el que *xi* y *yi* no sean iguales. La igualdad se determina utilizando `is(equal(xi,xj))` y la desigualdad con `is(notequal(xi,xj))`. En caso de un solo argumento, `kron_delta` devuelve un mensaje de error.

Ejemplos:

```
(%i1) kron_delta(a,a);
(%o1)                                1
(%i2) kron_delta(a,b,a,b);
(%o2)                        kron_delta(a, b)
(%i3) kron_delta(a,a,b,a+1);
(%o3)                                0
(%i4) assume(equal(x,y));
(%o4)                          [equal(x, y)]
(%i5) kron_delta(x,y);
(%o5)                                1
```

`listify` (`a`) [Función]

Si *a* es un conjunto, devuelve una lista con los elementos de *a*; si *a* no es un conjunto, devuelve *a*.

La función `full_listify` sustituye todos los operadores de conjunto en *a* por operadores de lista.

Ejemplos:

```
(%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*) [Función]
`lreduce` (*f*, *s*, *init*) [Función]

Amplía la función binaria F a n-aria mediante composición, siendo s una lista.

La sentencia `lreduce(F, s)` devuelve `F(... F(F(s_1, s_2), s_3), ... s_n)`. Si se incluye el argumento opcional *s_0*, el resultado equivale a `lreduce(F, cons(s_0, s))`.

La función F se aplica primero a los elementos del extremo izquierdo de la lista, de ahí el nombre `lreduce`, (*left reduce*).

Véanse también `rreduce`, `xreduce` y `tree_reduce`.

Ejemplos:

La función `lreduce` sin el 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)
```

La función `lreduce` con el argumento opcional,

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

La función `lreduce` aplicada a operadores binarios de Maxima. El símbolo `/` es el operador división.

```
(%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*) [Función]

Genera un conjunto cuyos miembros se generan a partir de la expresión *expr*, siendo x una lista de variables de *expr* y s un conjunto o lista de listas. Para generar los elementos del conjunto, se evalúa *expr* asignando a las variables de x los elementos de s en paralelo.

Los elementos de s deben tener la misma longitud que x. La lista de variables x debe ser una lista de símbolos sin subíndices. Cuando se trate de un único símbolo, x debe expresarse como una lista de un elemento y cada elemento de s debe ser una lista de un sólo elemento.

Véase también `makelist`.

Ejemplos:

```
(%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)` [Función]

Representa la función de Moebius.

Si n es el producto de k números primos diferentes, `moebius(n)` devuelve $(-1)^k$, retornando 1 si $n = 1$ y 0 para cualesquiera otros enteros positivos.

La función de Moebius se distribuye respecto de ecuaciones, listas, matrices y conjuntos.

Ejemplos:

```
(%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)` [Función]

`multinomial_coeff ()` [Función]

Calcula el coeficiente multinomial.

Si todos los a_k son enteros no negativos, el coeficiente multinomial es el número de formas de colocar `a_1 + ... + a_n` objetos diferentes en n cajas con a_k elementos

en la k-ésima caja. En general, `multinomial_coeff (a_1, ..., a_n)` calcula `(a_1 + ... + a_n)!/(a_1! ... a_n!)`.

Si no se dan argumentos, `multinomial_coeff()` devuelve 1.

Se puede usar `minfactorial` para simplificar el valor devuelto por `multinomial_coeff`.

Ejemplos:

```
(%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)` [Función]
`num_distinct_partitions (n, list)` [Función]

Si n es un entero no negativo, devuelve el número de particiones enteras distintas de n, en caso contrario `num_distinct_partitions` devuelve una forma nominal.

La sentencia `num_distinct_partitions(n, list)` devuelve una lista con el número de particiones distintas de 1, 2, 3, ..., n.

Una partición distinta de n es una lista de números enteros positivos distintos k_1, ..., k_m tales que $n = k_1 + ... + k_m$.

Ejemplos:

```
(%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)` [Función]
`num_partitions (n, list)` [Función]

Si n es un entero no negativo, devuelve el número de particiones enteras de n, en caso contrario `num_partitions` devuelve una expresión nominal.

La sentencia `num_partitions(n, list)` devuelve una lista con los números de particiones enteras de 1, 2, 3, ..., n.

Siendo n un entero no negativo, `num_partitions(n)` es igual a `cardinality(integer_partitions(n))`; sin embargo, `num_partitions` no construye el conjunto de particiones, por lo que es más rápido.

Ejemplos:

```
(%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)` [Función]

Particiona el conjunto *a* respecto del predicado *f*.

La función `partition_set` devuelve una lista con dos conjuntos; el primer conjunto es el subconjunto de *a* para el cual el predicado *f* devuelve `false` y el segundo contiene al resto de elementos de *a*.

La función `partition_set` no aplica `is` al valor devuelto por *f*.

La función `partition_set` emite un mensaje de error si *a* no es un conjunto literal.

Véase también `subset`.

Ejemplos:

```
(%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)` [Función]

Devuelve un conjunto con todas las permutaciones distintas de los miembros de la lista o conjunto *a*. Cada permutación es una lista, no un conjunto.

Si *a* es una lista, sus miembros duplicados no son eliminados antes de buscar sus permutaciones.

Si *a* no es una lista o conjunto, `permutations` emite un mensaje de error.

Véase también `random_permutation`.

Ejemplos:

```
(%i1) permutations ([a, a]);
(%o1)                         {[a, a]}
(%i2) permutations ([a, a, b]);
(%o2)           {[a, a, b], [a, b, a], [b, a, a]}
```

`powerset (a)` [Función]
`powerset (a, n)` [Función]

Devuelve el conjunto de todos los subconjuntos del conjunto *a* o un sunconjunto de ellos.

La sentencia `powerset(a)` devuelve el conjunto de todos los subconjuntos de *a*, que contendrá `2^cardinality(a)` elementos.

La sentencia `powerset(a, n)` devuelve el conjunto de todos los subconjuntos de *a* de cardinalidad *n*.

La función `powerset` emite un mensaje de error si *a* no es un conjunto literal o si *n* no es un entero no negativo.

Ejemplos:

```
(%i1) powerset ({a, b, c});
(%o1) {{}, {a}, {a, b}, {a, b, c}, {a, c}, {b}, {b, c}, {c}}
(%i2) powerset ({w, x, y, z}, 4);
(%o2)                    {{w, x, y, z}}
(%i3) powerset ({w, x, y, z}, 3);
(%o3)     {{w, x, y}, {w, x, z}, {w, y, z}, {x, y, z}}
(%i4) powerset ({w, x, y, z}, 2);
(%o4)   {{w, x}, {w, y}, {w, z}, {x, y}, {x, z}, {y, z}}
(%i5) powerset ({w, x, y, z}, 1);
(%o5)                 {{w}, {x}, {y}, {z}}
(%i6) powerset ({w, x, y, z}, 0);
(%o6)                         {{}}
```

`random_permutation (a)` [Función]

Devuelve una permutación aleatoria del conjunto o lista *a*, siguiendo el algoritmo de Knuth.

El valor devuelto es una lista nueva distinta del argumento, incluso cuando todos los elementos son iguales. Sin embargo, los elementos del argumento no se copian.

Ejemplos:

```
(%i1) random_permutation ([a, b, c, 1, 2, 3]);
(%o1)                   [c, 1, 2, 3, a, b]
(%i2) random_permutation ([a, b, c, 1, 2, 3]);
(%o2)                   [b, 3, 1, c, a, 2]
(%i3) random_permutation ({x + 1, y + 2, z + 3});
(%o3)                  [y + 2, z + 3, x + 1]
(%i4) random_permutation ({x + 1, y + 2, z + 3});
(%o4)                  [x + 1, y + 2, z + 3]
```

`rreduce (`*f*`, `*s*`)` [Función]

`rreduce (`*f*`, `*s*`, `*init*`)` [Función]

Amplía la función binaria *F* a n-aria mediante composición, siendo *s* una lista.

La sentencia `rreduce(`*F*`, `*s*`)` devuelve `F(s_1, ... F(s_{n - 2}, F(s_{n - 1}, s_n)))`. Si se incluye el argumento opcional *s_{n + 1}*, el resultado equivale a `rreduce(`*F*`, endcons(`*s_{n + 1}*`, `*s*`))`.

La función *F* se aplica primero a los elementos del extremo derecho de la lista, de ahí el nombre `rreduce`, (*right reduce*).

Véanse también `lreduce`, `xreduce` y `tree_reduce`.

Ejemplos:

La función `rreduce` sin el 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)))
```

La función `rreduce` con el argumento opcional,

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

La función `rreduce` aplicada a operadores binarios de Maxima. El símbolo `/` es el operador división.

```
(%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*) [Función]

Devuelve el conjunto con los elementos del conjunto *a* que no pertenecen al conjunto *b*.

La función `setdifference` emite un mensaje de error si *a* o *b* no son conjuntos.

Ejemplos:

```
(%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*) [Función]

Devuelve `true` si los conjuntos *a* y *b* tienen el mismo número de elementos y `is (`*x* `=` *y*`)` vale `true` para `x` perteneciente a *a* e `y` perteneciente a *b*, considerados en el orden que determina la función `listify`. En caso contrario, `setequalp` devuelve `false`.

Ejemplos:

```
(%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)` [Función]

Construye un conjunto con los miembros de la lista *a*. Los elementos duplicados de la lista *a* son borrados y ordenados de acuerdo con el predicado `orderlessp`.

La función `setify` emite un mensaje de error si *a* no es un conjunto literal.

Ejemplos:

```
(%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)` [Función]

Devuelve `true` si y sólo si *a* es un conjunto de Maxima.

La función `setp` devuelve `true` tanto cuando el conjunto tiene como cuando no tiene elementos repetidos.

La función `setp` is equivalent to the Maxima function `setp(a) := not atom(a) and op(a) = 'set`.

Ejemplos:

```
(%i1) simp : false;
(%o1)                            false
(%i2) {a, a, a};
(%o2)                         {a, a, a}
(%i3) setp (%);
(%o3)                            true
```

`set_partitions (a)` [Función]
`set_partitions (a, n)` [Función]

Devuelve el conjunto de todas las particiones de *a* o un subconjunto de ellas.

La sentencia `set_partitions(a, n)` devuelve un conjunto con todas las descomposiciones de *a* en *n* conjuntos no vacíos disjuntos.

La sentencia `set_partitions(a)` devuelve el conjunto de todas las particiones.

La función `stirling2` devuelve la cardinalidad del conjunto de las particiones de un conjunto.

Se dice que un conjunto P es una partición del conjunto S si verifica

1. cada elemento de P es un conjunto no vacío,
2. los elementos de P son disjuntos,
3. la unión de los elementos de P es igual a S.

Ejemplos:

El conjunto vacío forma una partición de sí mismo,

```
(%i1) set_partitions ({});
```

```
(%o1)                          {{}}
```

La cardinalidad del conjunto de particiones de un conjunto puede calcularse con `stirling2`,

```
(%i1) s: {0, 1, 2, 3, 4, 5}$
(%i2) p: set_partitions (s, 3)$
(%i3) cardinality(p) = stirling2 (6, 3);
(%o3)                        90 = 90
```

Cada elemento de `p` debería tener $n = 3$ miembros,

```
(%i1) s: {0, 1, 2, 3, 4, 5}$
(%i2) p: set_partitions (s, 3)$
(%i3) map (cardinality, p);
(%o3)                          {3}
```

Por último, para cada miembro de `p`, la unión de sus elementos debe ser igual a `s`,

```
(%i1) s: {0, 1, 2, 3, 4, 5}$
(%i2) p: set_partitions (s, 3)$
(%i3) map (lambda ([x], apply (union, listify (x))), p);
(%o3)                {{0, 1, 2, 3, 4, 5}}
```

`some` (*f*, *a*) [Función]
`some` (*f*, *L_1*, ..., *L_n*) [Función]

Devuelve `true` si el predicado *f* devuelve `true` para al menos uno de sus argumentos. Si el segundo argumento es un conjunto, `some (f, a)` devuelve `true` si `f(a_i)` devuelve también `true` para alguno de los *a_i* en *a*; puede ser que `some` no evalúe *f* para todos los *a_i* de *s*. Puesto que los conjuntos no están ordenados, `some` puede evaluar `f(a_i)` en cualquier orden.

Dada una o más listas como argumentos, `some (f, L_1, ..., L_n)` devuelve `true` si `f(x_1, ..., x_n)` devuelve también `true` para al menos un *x_1*, ..., *x_n* de *L_1*, ..., *L_n*, respectivamente; puede ser que `some` no evalúe *f* para todos las combinaciones *x_1*, ..., *x_n*. La función `some` evalúa las listas en el orden creciente de su índice

Dado un conjunto vacío `{}` o una lista vacía como argumentos, `some` devuelve `false`.

Si la variable global `maperror` vale `true`, todas las listas *L_1*, ..., *L_n* deben tener igual número de elementos. Si `maperror` vale `false`, los argumentos se truncan para tener todos el número de elementos de la lista más corta.

Los valores que devuelve el predicado *f* cuando toman (mediante `is`) un valor diferente a `true` y `false` se controlan con la variable global `prederror`. Si `prederror` vale `true`, tales valores se consideran como `false`. Si `prederror` vale `false`, tales valores se consideran como desconocidos (`unknown`).

Ejemplos:

La función `some` aplicada a un único conjunto. El predicado es una función de un 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
```

La función `some` aplicada a dos listas. El predicado es una función de dos argumentos,

```
(%i1) some ("=", [a, b, c], [a, b, c]);
(%o1)                          true
(%i2) some ("#", [a, b, c], [a, b, c]);
(%o2)                          false
```

Las respuestas del predicado f que se evalúan a cualquier cosa diferente de `true` y `false` están controlados por la variable 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*`)` [Función]

Es el número de Stirling de primera especie.

Si tanto n como m son enteros no negativos, el valor que toma `stirling1 (`*n*`, `*m*`)` es el número de permutaciones de un conjunto de n elementos con m ciclos. Para más detalles, véase Graham, Knuth and Patashnik *Concrete Mathematics*. Maxima utiliza una relación recursiva para definir `stirling1 (`*n*`, `*m*`)` para m menor que 0; no está definida para n menor que 0 ni para argumentos no enteros.

La función `stirling1` es simplificadora. Maxima reconoce las siguientes 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])

Estas identidades se aplican cuando los argumentos son enteros literales o símbolos declarados como enteros y el primer argumento es no negativo. La función `stirling1` no simplifica para argumentos no enteros.

Referencias:

[1] Donald Knuth, *The Art of Computer Programming,* Tercera Edición, Volumen 1, Sección 1.2.6, Ecuaciones 48, 49 y 50.

Ejemplos:

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling1 (n, n);
(%o3)                          1
```

La función `stirling1` no simplifica en caso de argumentos no enteros,

```
(%i1) stirling1 (sqrt(2), sqrt(2));
(%o1)              stirling1(sqrt(2), sqrt(2))
```

Maxima aplicas algunas 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*`)` [Función]

Es el número de Stirling de segunda especie.

Si n y m son enteros no negativos, `stirling2 (`*n*`, `*m*`)` es el número de formas en las que se puede particionar un conjunto de cardinal n en m subconjuntos disjuntos. Maxima utiliza una relación recursiva para definir `stirling2 (`*n*`, `*m*`)` con m menor que 0; la función no está definida para n menor que 0 ni para argumentos no enteros.

La función `stirling2` es simplificadora. Maxima reconoce las siguientes 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. $stirling2(n, m) = sum((-1)^{(}m - k) binomial(mk) k^n, i, 1, m)/m!$ si m y n son enteros y n no negativo. (Ref. [3])

Estas identidades se aplican cuando los argumentos son enteros literales o símbolos declarados como enteros y el primer argumento es no negativo. La función `stirling2` no simplifica para argumentos no enteros.

Referencias:

[1] Donald Knuth. *The Art of Computer Programming*, Tercera Edición, Volumen 1, Sección 1.2.6, Ecuaciones 48, 49 y 50.

[2] Graham, Knuth y Patashnik. *Concrete Mathematics*, Tabla 264.

[3] Abramowitz y Stegun. *Handbook of Mathematical Functions*, Sección 24.1.4.

Ejemplos:

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling2 (n, n);
(%o3)                            1
```

La función `stirling2` no simplifica en caso de argumentos no enteros,

```
(%i1) stirling2 (%pi, %pi);
(%o1)                  stirling2(%pi, %pi)
```

Maxima aplicas algunas 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*) [Función]

Devuelve el subconjunto del conjunto *a* que satisface el predicado *f*.

La función `subset` devuelve el conjunto que contiene a los elementos de *a* para los cuales *f* devuelve un resultado diferente de `false`. La función `subset` no aplica `is` al valor retornado por *f*.

La función `subset` emite un mensaje de error si *a* no es un conjunto literal.

Véase también `partition_set`.

Ejemplos:

```
(%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*) [Función]

Devuelve `true` si y sólo si el conjunto *a* es un subconjunto de *b*.

La función `subsetp` emite un mensaje de error si cualesquiera *a* o *b* no es un conjunto literal.

Ejemplos:

```
(%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*) [Función]

Devuelve la diferencia simétrica de los conjuntos *a_1*, *...*, *a_n*.

Dados dos argumentos, `symmdifference (a, b)` equivale a `union (setdifference (a, b), setdifference (b, a))`.

La función `symmdifference` emite un mensaje de error si alguno de su argumentos no es un conjunto literal.

Ejemplos:

```
(%i1) S_1 : {a, b, c};
(%o1)                         {a, b, c}
(%i2) S_2 : {1, b, c};
(%o2)                         {1, b, c}
(%i3) S_3 : {a, b, z};
(%o3)                         {a, b, z}
(%i4) symmdifference ();
(%o4)                            {}
(%i5) symmdifference (S_1);
(%o5)                         {a, b, c}
(%i6) symmdifference (S_1, S_2);
(%o6)                          {1, a}
(%i7) symmdifference (S_1, S_2, S_3);
(%o7)                         {1, b, z}
(%i8) symmdifference ({}, S_1, S_2, S_3);
(%o8)                         {1,b, z}
```

`tree_reduce (F, s)` [Función]
`tree_reduce (F, s, s_0)` [Función]

Amplía la función binaria F a n-aria, siendo s una lista.

La función `tree_reduce` equivale a lo suguiente: Aplicar F a pares sucesivos de elementos para formar una nueva lista `[F(s_1, s_2), F(s_3, s_4), ...]`, llevando el elemento final sin cambiar si el número de elementos es impar; después repetir hasta que la lista se reduzca a un único elemento, que es el valor de retorno.

Cuando está presente el argumento opcional *s_0*, el resultado equivale a `tree_reduce(F, cons(s_0, s)`.

Para la suma de números decimales en coma flotante, `tree_reduce` puede devolver una suma que tenga un error de redondeo menor que el conseguido por `rreduce` o `lreduce`.

Los elementos de s y los resultados parciales pueden colocarse en un árbol binario de mínima profundidad, de ahí el nombre de *tree_reduce*.

Ejemplos:

La función `tree_reduce` aplicada a una lista con un número par de elementos,

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

La función `tree_reduce` aplicada a una lista con un número impar 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*) [Function]

Devuelve la unión de los conjuntos *a_1* hasta *a_n*.

La sentencia `union()` (sin argumentos) devuelve el conjunto vacío.

La función `union` emite un mensaje de error si alguno de sus argumentos no es un conjunto literal.

Ejemplos:

```
(%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*) [Función]

`xreduce` (*F*, *s*, *s_0*) [Función]

Amplía la función F a n-aria mediante composición; si F ya es n-aria, aplica F a s. Si F no es n-aria, `xreduce` equivale a `lreduce`. El argumento s debe ser una lista.

Funciones n-arias reconocidas por Maxima son la suma `+`, la multiplicación `*`, `and`, `or`, `max`, `min` y `append`. Las funciones también se pueden declarar n-arias mediante `declare(`*F*`, nary)`; para estas funciones, `xreduce` será más rápida que `rreduce` o `lreduce`.

Cuando está presente el argumento opcional *s_0*, el resultado equivale a `xreduce(`*s*`, cons(`*s_0*`, `*s*`))`.

La suma de números decimales en coma flotante no es exactamente asociativa; aún así, `xreduce` aplica la suma n-aria cuando s contiene números en coma flotante.

Ejemplos:

La función `xreduce` aplicada a una función n-aria; `F` es invocada una sóla vez, con todos sus 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]
```

La función `xreduce` aplicada a una función que se desconoce si es n-aria; `G` es invocada varias veces, con dos 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]
```

36 Definición de Funciones

36.1 Introducción a la definición de funciones

36.2 Funciones

36.2.1 Funciones ordinarias

Para definir una función en Maxima es necesario utilizar el operador ':='.

Por ejemplo,

```
f(x) := sin(x)
```

define una función `f`. También se pueden definir funciones anónimas utilizando `lambda`; por ejemplo,

```
lambda ([i, j], ...)
```

puede utilizarse en lugar de `f` donde

```
f(i,j) := block ([], ...);
map (lambda ([i], i+1), l)
```

devolvería una lista con todos sus elementos aumentados en una unidad.

También se puede definir una función con un número variable de argumentos, sin más que añadir un argumento final al que se le asigna una lista con todos los argumentos adicionales.:

```
(%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]]
```

El miembro derecho de una función debe ser una expresión. Así, si se quiere una secuencia de expresiones, se debe hacer

```
f(x) := (expr1, expr2, ...., exprn);
```

siendo el valor que alcance exprn el devuelto por la función.

Si se quiere hacer un `return` desde alguna de las expresiones de la función, se debe utilizar la estructura `block` junto con `return`. Por ejemplo,

```
block ([], expr1, ..., if (a > 10) then return(a), ..., exprn)
```

es una expresión de pleno derecho, por lo que puede ocupar el lado derecho de la definición de una función. Aquí puede ocurrir que el retorno se produzca antes que se alcance la última expresión.

Los primeros corchetes del bloque (`[]`) pueden contener una lista de variables junto con posibles asignaciones, tal como `[a: 3, b, c: []]`, lo que provocará que las tres variables `a`,`b` y `c` se consideren locales y sean independientes de otras globales con el mismo nombre; las variables locales sólo estarán activas mientras se ejecute el código que está dentro de la

estructura `block`, o dentro de funciones que son llamadas desde dentro de `block`. A esto se le llama asignación dinámica, pues las variables sobreviven desde el inicio del bloque hasta que éste deje de estar operativo. Una vez se salga del bloque los valores originales de las variables, si es que los había, quedan restaurados. Es recomendable proteger las variables de esta forma. Se tendrá en cuenta que las asignaciones a las variables del bloque se hacen en paralelo, lo que significa que si como en el ejemplo anterior se hace `c: a` en el momento de entrar en el bloque, el valor de `c` será el que tenía `a` antes de entrar en el bloque, es decir, antes de la asignación `a: 3`. Así, haciendo lo siguiente

```
block ([a: a], expr1, ... a: a+3, ..., exprn)
```

se prevendría de que el valor externo de `a` fuese alterado, pero permitiría acceder a él desde dentro del bloque. La parte derecha de las asignaciones se evalúa dentro de su contexto antes de hacer efectiva la asignación. Utilizando únicamente `block([x],..` haría que `x` se tuviese a sí misma como valor, justo como si se acabase de iniciar una nueva sesión de Maxima.

Los valores de los argumentos de una funcón se tratan exactamente de la misma forma que las variables de un bloque. Así, con

```
f(x) := (expr1, ..., exprn);
```

y

```
f(1);
```

se estaría en un contexto similar para la evaluación de las expresiones como si se hubiera hecho

```
block ([x: 1], expr1, ..., exprn)
```

Dentro de las funciones, cuando el lado derecho de la definición deba ser evaluado será útil hacer uso de `define` y posiblemente de `buildq`.

36.2.2 Funciones array

Una función array almacena el valor de la función la primera vez que es invocada con un argumento dado, devolviendo el valor almacenado sin recalcularlo cuando es llamada con ese mismo argumento. Estas funciones reciben también el nombre de *funciones memorizadoras*.

Los nombres de las funciones array son añadidos a la lista global `arrays`, no a la lista global `functions`. La función `arrayinfo` devuelve la lista de argumentos para los que hay valores almacenados y `listarray` devuelve precisamente estos valores almacenados. Las funciones `dispfun` y `fundef` devuelven la definición de la función array.

La función `arraymake` construye una llamada a una función array, de forma similar a como lo hace `funmake` para las funciones ordinarias. Por otro lado, `arrayapply` aplica una función array a sus argumentos, tal como lo hace `apply` con las funciones ordinarias. No existe para las funciones array nada similar a `map`, aunque `map(lambda([x], a[x]), L)` o `makelist(a[x], x, L)`, siendo L una lista, podrían suplantar esta carencia.

La función `remarray` borra la definición de una función array, así como cualesquiera valores almacenados que tenga asociados, tal como `remfunction` lo hace con las funciones ordinarias.

La llamada `kill(a[x])` borra el valor de la función array a almacenado para el argumento x; la próxima vez que se llame a a con el argumento x, se recalculará el valor correspondiente. Sin embargo, no hay forma de borrar todos los valores almacenados de una sola vez, excepto mediante `kill(a)` o `remarray(a)`, con lo que se borra también la definición de la propia función.

36.3 Macros

`buildq` (*L*, *expr*) [Función]

Sustituye en paralelo las variables nombradas en la lista *L* en la expresión *expr*, sin evaluar ésta. La expresión resultante se simplifica pero no se evalúa hasta que `buildq` termine de hacer las sustituciones.

Los elementos de *L* son símbolos o expresiones de asignación del tipo *`symbol`*`: `*`value`*, evaluadas en paralelo. Esto es, el valor de una variable en la parte derecha de una asignación es el valor que toma dicha variable en el contexto desde el que se invoca a `buildq`. En caso de que a una variable de *L* no se le haga una signación explícita, su valor en `buildq` es el mismo que tiene en el contexto desde el que se llama a `buildq`.

Las variables referenciadas en *L* se sustituyen en *expr* en paralelo. Esto es, la sustitución para cada variable se determina antes de que se hagan las sustituciones, de forma que la sustitución de una variable no tiene efecto alguno sobre las otras.

Si alguna variable *x* aparece como `splice (x)` en *expr*, entonces a *x* se le debe asignar una lista, la cual será interpolada en *expr* en lugar de hacer una simple sustitución; ver ejemplo más abajo.

Cualesquiera otras variables de *expr* que no aparezcan en *L* se traspasan al resultado tal cual, incluso cuando tienen asignados valores en el contexto desde el que se llama a `buildq`.

Ejemplos:

`a` queda asociada explícitamente a `x`, mientras que `b` tiene la misma asociación (29) que en el contexto de llamada y `c` es traspasado al resultado sin ser sustituido. La expresión resultante no se evalúa hasta que no se le obligue a ello mediante la evaluación explícita `''%`.

```
(%i1) (a: 17, b: 29, c: 1729)$
(%i2) buildq ([a: x, b], a + b + c);
(%o2)                       x + c + 29
(%i3) ''%;
(%o3)                        x + 1758
```

En este ejemplo, `e` se asocia a una lista, la cual aparece como tal en los argumentos de `foo` e interpolada en los 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)
```

Como se ve a continuación, el resultado se simplifica tras las sustituciones. Si la simplificación se realizase antes que las sustituciones, ambos resultados serían iguales.

```
(%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
```

Las variables de *L* se asocian en paralelo; si se hiciese secuencialmente, el primer resultado sería `foo (b, b)`. Las sustituciones se llevan a cabo en paralelo. Compárese

el segundo resultado con el resultado de `subst`, que hace las sustituciones de forma secuencial.

```
(%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)
```

Se construye a continuación un sistema de ecuaciones con algunas variables o expresiones en el lado izquierdo y sus valores en el derecho; `macroexpand` muestra la expresión devuelta 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 - b - a = 1683]
(%i4) macroexpand (show_values (a, b, c - a - b));
(%o4)    map(=, '([a, b, c - b - a]), [a, b, c - b - a])
```

Dada una función con varios argumentos, se crea otra función en la cual algunos argumentos son fijos.

```
(%i1) curry (f, [a]) :=
        buildq ([f, a], lambda ([[x]], apply (f, append (a, x))))$
(%i2) by3 : curry ("*", 3);
(%o2)        lambda([[x]], apply(*, append([3], x)))
(%i3) by3 (a + b);
(%o3)                        3 (b + a)
```

`macroexpand` (*expr*) [Función]

Devuelve la macroexpansión de *expr*, sin evaluarla, cuando `expr` es una llamada a una función macro; en caso contrario, `macroexpand` devuelve *expr*.

Si la expansión de *expr* devuelve otra llamada a una función macro, esta llamada también se expande.

La función `macroexpand` no evalúa su argumento. Sin embargo, si la expansión de una llamada a función macro tiene efectos laterales, éstos se ejecutan.

Véanse también `::=`, `macros` y `macroexpand1`.

Ejemplos:

```
(%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*) [Función]

Devuelve la macroexpansión de *expr*, sin evaluarla, cuando `expr` es una llamada a una función macro; en caso contrario, `macroexpand1` devuelve *expr*.

La función `macroexpand1` no evalúa su argumento. Sin embargo, si la expansión de una llamada a función macro tiene efectos laterales, éstos se ejecutan.

Si la expansión de *expr* devuelve otra llamada a una función macro, esta llamada no se expande.

Véanse también `::=`, `macros` y `macroexpand`.

Ejemplos:

```
(%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` [Variable global]

Valor por defecto: `[]`

La variable `macros` es la lista de las funciones macro definidas por el usuario. El operador de definición de funciones macro `::=` coloca la nueva función macro en esta lista, mientras que `kill`, `remove` y `remfunction` eliminan las funciones macro de la lista.

Véase también `infolists`.

`splice` (`a`) [Función]

Interpola la lista nombrada por el átomo *a* dentro de una expresión, pero sólo si `splice` aparece dentro de `buildq`; en otro caso, `splice` se considera una función no definida. Si *a* aparece dentro de `buildq` sin `splice`, entonces queda sustituida por una

lista dentro del resultado. El argumento de `splice` debe ser un átomo, no pudiendo ser una lista literal ni una expresión que devuelva una lista.

Normalmente `splice` suministra los argumentos para una función u operador. Para una función `f`, la expresión `f (splice (a))` dentro de `buildq` se convierte en `f (a[1], a[2], a[3], ...)`. Dado un operador `o`, la expresión `"o" (splice (a)` dentro de `buildq` se convierte en `"o" (a[1], a[2], a[3], ...)`, donde `o` puede ser cualquier tipo de operador, normalmente uno que admita varios argumentos. Nótese que el operador debe ir encerrado entre comillas dobles `"`.

Ejemplos:

```
(%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<>
```

36.4 Funciones y variables para la definición de funciones

`apply` (*F*, [*x_1*, ..., *x_n*]) [Función]

Construye y evalúa la expresión *F*(*arg_1*, ..., *arg_n*).

La función `apply` no hace distinciones entre funciones array y funciones ordinarias; cuando *F* es el nombre de una función array, `apply` evalúa *F*(...), esto es, hace una llamada con paréntesis en lugar de corchetes. La función `arrayapply` evalúa una llamada a función con corchetes para estos casos.

Ejemplos:

La función `apply` evalúa sus argumentos. En este ejemplo, `min` se aplica al valor de `L`.

```
(%i1) L : [1, 5, -10.2, 4, 3];
(%o1)                  [1, 5, - 10.2, 4, 3]
(%i2) apply (min, L);
(%o2)                          - 10.2
```

La función `apply` evalúa sus argumentos, incluso cuando la función *F* no lo hace.

```
(%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]
```

La función `apply` evalúa el nombre de función *F*. La comilla simple `'` evita la evaluación. El nombre `demoivre` corresponde a una variable global y también a una función.

```
(%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*) [Función]
`block` (*expr_1*, ..., *expr_n*) [Función]

La función `block` evalúa *expr_1*, ..., *expr_n* secuencialmente y devuelve el valor de la última expresión evaluada. La secuencia puede alterarse con las funciones `go`, `throw` y `return`. La última expresión es *expr_n* a menos que `return` o una expresión que contenga un `throw` sea evaluada. Las variables *v_1*, ..., *v_m* son locales en el bloque; éstas se distiguen de las globales que tengan el mismo nombre. Si no se declaran variables locales entonces se puede omitir la lista. Dentro del bloque, cualquier otra variable distinta de *v_1*, ..., *v_m* se considera global.

La función `block` guarda los valores actuales de las variables *v_1*, ..., *v_m*, si los tienen, a la entrada del bloque y luego las evalúa a sí mismas, es decir les saca el valor temporalmente. A las variables locales se les puede asignar cualquier valor dentro del bloque, pero al salir de éste, los valores inicialmente almacenados quedan restaurados, al tiempo que los asignados dentro del bloque se pierden.

La declaración `local(v_1, ..., v_m)` dentro de un bloque almacena las propiedades asociadas a los símbolos *v_1*, ..., *v_m*, borra cualesquiera otras propiedades antes de evaluar las expresiones y restaura las propiedades guardadas antes de abandonar el bloque. Algunas declaraciones, como `:=`, `array`, `dependencies`, `atvalue`,

`matchdeclare`, `atomgrad`, `constant`, `nonscalar`, `assume` y otras se implementan como propiedades de símbolos. El efecto producido por `local` consiste en hacer que tales declaraciones tengan efecto sólo dentro del bloque, en otro caso las declaraciones dentro del bloque tendrían un efecto global que afectarían al exterior de `block`.

Un `block` puede aparecer dentro de otro `block`. Las variables locales se inicializan cada vez que se entra dentro de un nuevo bloque. Las variables locales de un bloque se consideran globales dentro de otro anidado dentro del primero. Si una variable es no local dentro de un bloque, su valor es el que le corresponde en el bloque superior. Este criterio se conoce con el nombre de "alcance dinámico".

El valor del bloque es el de la última sentencia o el argumento de la función `return`, que puede utilizarse para salir del bloque. La función `go` puede usarse para transferir el control a la sentencia del bloque que esté etiquetada con el argumento de `go`. Para etiquetar una sentencia basta que vaya precedida de un argumento atómico como cualquier otra sentencia dentro del bloque. Por ejemplo, `block ([x], x:1, tururu, x: x+1, ..., go(tururu), ...)`. El argumento de `go` debe ser el nombre de una etiqueta colocada dentro del bloque. No se puede utilzar `go` para trasladarse a una etiqueta de un bloque que no sea el que contenga a `go`.

Normalmente los bloques aparecerán al lado derecho de las definiciones de funciones, pero también pueden utilizarse en otros contextos.

`break` (*expr_1*, ..., *expr_n*) [Función]

Calcula e imprime *expr_1*, ..., *expr_n* para luego provocar la detención de Maxima, de modo que el usuario pueda examinar y cambiar el entorno de ejecución. Pulsando posteriormente `exit;` el cálculo se reanuda.

`catch` (*expr_1*, ..., *expr_n*) [Función]

Evalúa *expr_1*, ..., *expr_n* una a una; si alguna de ellas conlleva la evaluación de una expresión de la forma `throw (arg)`, entonces el valor de `catch` es el de `throw (arg)` y ya no se evalúan más expresiones. Esta respuesta pasa todos los niveles de anidamiento hasta el `catch` más próximo. Si no hay ningún `catch` que contenga un `throw` se emite un mensaje de error.

Si la evaluación de los argumentos no conlleva la evaluación de ningún `throw`, entonces el valor de `catch` es el devuelto por *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
```

La función `g` devuelve las imágenes por `f` de todos los elementos de la lista `l` si ésta contiene únicamente números no negativos; si no es este el caso, entonces `g` captura el primer negativo que encuentra y lo devuelve por medio del `throw`.

`compfile` (*filename*, *f_1*, ..., *f_n*) [Function]

`compfile` (*filename*, *functions*) [Function]

`compfile` (*filename*, *all*) [Function]

Traduce funciones de Maxima a código Lisp, guardándolo luego en el fichero *filename*.

Con la llamada `compfile(filename, f_1, ..., f_n)` se traducen las funciones especificadas, mientras que `compfile(filename, functions)` y `compfile(filename, all)` traducen las funciones definidas por el usuario.

El código Lisp traducido no se evalúa, ni el fichero de salida es procesado por el compilador de Lisp. La función `translate` crea y evalúa las traducciones Lisp, mientras que `compile_file` traduce primero de Maxima a Lisp y luego ejecuta el compilador Lisp.

Véanse también `translate`, `translate_file` y `compile_file`.

`compile` (*f_1*, ..., *f_n*) [Función]
`compile` (*functions*) [Función]
`compile` (*all*) [Función]

Traduce las funciones de Maxima *f_1*, ..., *f_n* a Lisp, evaluando el código resultante, y llama a la función Lisp `COMPILE` para cada función traducida. La función `compile` devuelve una lista con los nombres de las funciones compiladas.

Las llamadas `compile (all)` o `compile (functions)` compilan todas las funciones definidas por el usuario.

La función `compile` no evalúa sus argumentos, pero con el operador comilla-comilla (`''`) sí lo hace.

`define` (*f*(*x_1*, ..., *x_n*), *expr*) [Función]
`define` (*f*[*x_1*, ..., *x_n*], *expr*) [Función]
`define` (*funmake* (*f*, [*x_1*, ..., *x_n*]), *expr*) [Función]
`define` (*arraymake* (*f*, [*x_1*, ..., *x_n*]), *expr*) [Función]
`define` (*ev* (*expr_1*), *expr_2*) [Función]

Define una función de nombre *f* con argumentos *x_1*, ..., *x_n* y cuerpo *expr*. `define` evalúa siempre su segundo argumento, a menos que se indique lo contrario con el operador de comilla simple. La función así definida puede ser una función ordinaria de Maxima (con sus argumentos encerrados entre paréntesis) o una función array (con sus argumentos encerrados entre corchetes).

Cuando el último o único argumento *x_n* es una lista de un solo elemento, la función definida por `define` acepta un número variable de argumentos. Los valores de los argumentos se van asignando uno a uno a *x_1*, ..., *x_(n - 1)*, y los que queden, si los hay, se asignan a *x_n* en forma de lista.

Cuando el primer argumento de `define` es una expresión de la forma `f(x_1, ..., x_n)` o `f[x_1, ..., x_n]`, se evalúan los argumentos de la función, pero no *f*, incluso cuando se trate de una función o variable ya existente con ese nombre.

Cuando el primer argumento es una expresión con operador `funmake`, `arraymake` o `ev`, se evalúa este primer argumento, lo que permite calcular la función.

Todas las definiciones de funciones aparecen en el mismo espacio de nombres; definiendo una función `f` dentro de otra función `g` no limita automáticamente el alcance de `f` a `g`. Sin embargo, `local(f)` hace que la definición de la función `f` sea efectiva sólo dentro del bloque o expresión compuesta en el que aparece `local`.

Si un argumento formal *x_k* es un símbolo afectado por el operador comilla simple (expresión nominal), la función definida por `define` no evalúa el correspondiente valor de argumento. En cualquier otro caso, los argumentos que se pasan son evaluados.

Véanse también `:=` y `::=`.

Ejemplos:

`define` evalúa siempre su segundo argumento, a menos que se indique lo contrario con el operador de comilla simple.

```
(%i1) expr : cos(y) - sin(x);
(%o1)                     cos(y) - sin(x)
(%i2) define (F1 (x, y), expr);
(%o2)               F1(x, y) := cos(y) - sin(x)
(%i3) F1 (a, b);
(%o3)                     cos(b) - sin(a)
(%i4) F2 (x, y) := expr;
(%o4)                    F2(x, y) := expr
(%i5) F2 (a, b);
(%o5)                     cos(y) - sin(x)
```

La función así definida puede ser una función ordinaria de Maxima o una función array.

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

Cuando el último o único argumento *x_n* es una lista de un solo elemento, la función definida por `define` acepta un número variable de argumentos.

```
(%i1) define (H ([L]), '(apply ("+", L)));
(%o1)                 H([L]) := apply("+", L)
(%i2) H (a, b, c);
(%o2)                         c + b + a
```

Cuando el primer argumento es una expresión con operador `funmake`, `arraymake` o `ev`, se evalúa este primer argumento.

```
(%i1) [F : I, u : x];
(%o1)                          [I, x]
(%i2) funmake (F, [u]);
(%o2)                           I(x)
(%i3) define (funmake (F, [u]), cos(u) + 1);
(%o3)                    I(x) := cos(x) + 1
(%i4) define (arraymake (F, [u]), cos(u) + 1);
(%o4)                    I  := cos(x) + 1
                          x
(%i5) define (foo (x, y), bar (y, x));
(%o5)                 foo(x, y) := bar(y, x)
(%i6) define (ev (foo (x, y)), sin(x) - cos(y));
(%o6)              bar(y, x) := sin(x) - cos(y)
```

`define_variable` (*name*, *default_value*, *mode*) [Función]

Introduce una variable global en el entorno de Maxima. La función `define_variable` puede ser útil en los paquetes escritos por los usuarios que vayan a ser compilados o traducidos con frecuencia.

La función `define_variable` ejecuta los siguientes pasos:

1. `mode_declare (`*name*`, `*mode*`)` declara el modo de *name* al traductor. Véase `mode_declare` para ver la lista de modos aceptables.
2. Si aún no tiene asignación, se le da a la variable *default_value* el valor *name*.
3. `declare (`*name*`, special)` la declara como especial.
4. Asocia *name* a una función de comprobación para asegurar que a *name* sólo se le asignan valores del modo declarado.

La propiedad `value_check` se puede asociar a cualquier variable que haya sido definida mediante `define_variable` en cualquiera de los modos diferentes a `any`. La propiedad `value_check` puede ser una expresión lambda o una función de una variable, que será invocada al intentar asignar un valor a la variable; el argumento pasado a la función `value_check` es el valor que se le quiere asignar a la variable.

La función `define_variable` evalúa `default_value` pero no `name` ni `mode`; el valor que devuelve es el valor actual de `name`, el cual es `default_value` si a `name` no se le ha aplicado ninguna asignación, o el valor de dicha asignación en caso contrario.

Ejemplos:

`foo` es una variable booleana con 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` es una variable entera, cuyo valor habrá de ser 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 is not prime.
#0: prime_test(y=1440)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i6) bar;
(%o6)                          1439
```

`baz_quux` es una variable a la que no se le podrá asignar valor alguno. El modo `any_check` es como `any`, pero `any_check` activa el mecanismo `value_check`, cosa que `any` no hace.

```
(%i1) define_variable (baz_quux, 'baz_quux, any_check);
(%o1)                         baz_quux
(%i2) F: lambda ([y], if y # 'baz_quux then
                 error ("Cannot assign to ‘baz_quux’."));
(%o2) lambda([y], if y # 'baz_quux
                  then error(Cannot assign to ‘baz_quux’.))
(%i3) qput (baz_quux, ''F, value_check);
(%o3) lambda([y], if y # 'baz_quux
                  then error(Cannot assign to ‘baz_quux’.))
(%i4) baz_quux: 'baz_quux;
(%o4)                         baz_quux
(%i5) baz_quux: sqrt(2);
Cannot assign to ‘baz_quux’.
#0: lambda([y],if y # 'baz_quux then
             error("Cannot assign to ‘baz_quux’."))(y=sqrt(2))
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i6) baz_quux;
(%o6)                         baz_quux
```

`dispfun` (*f_1*, ..., *f_n*) [Función]

`dispfun` (*all*) [Función]

Muestra la deficnión de las funciones de usuario *f_1*, ..., *f_n*. Cada argumento puede ser el nombre de una macro (definida mediante `::=`), una función ordinaria (definida mediante `:=` o `define`), una función arreglo (definida mediante `:=` o `define`, pero encerrando los argumentos dentro de corchetes `[ ]`), una función de subíndice (definida mediante `:=` o `define`, pero encerrando algunos argumentos entre corchetes y otros entre paréntesis `( )`), una función de subíndice seleccionada por un subíndice variable, o una función de subíndice definida con un subíndice constante.

La llamada `dispfun (all)` muestra todas las funciones de usuario tal como las dan las listas `functions`, `arrays` y `macros`, omitiendo las funciones con subíndices definidas con subíndices constantes.

La función `dispfun` crea una etiqueta (`%t1`, `%t2`, etc.) para cada función mostrada, y asigna la definición de la función a la etiqueta. En contraste, `fundef` devuelve las definiciones de las funciones.

La función `dispfun` no evalúa sus argumentos; el operador de comilla-comilla `''` permite la evaluación.

La función `dispfun` devuelve la lista de etiquetas de expresiones intermedias correspondientes a las funciones mostradas.

Ejemplos:

```
(%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
```

`fullmap` (*f*, *expr_1*, ...) [Función]

Similar a `map`, pero conservará el mapeado descendente de todas las subexpresiones hasta que los operadores principales ya no sean los mismos.

La función `fullmap` es utilizada por el simplificador de Maxima en algunas transformaciones matriciales, por lo que Maxima generará en algunas ocasiones mensajes de error relacionados con `fullmap` aunque el usuario no haya invocado explícitamente esta función.

```
(%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*, ...) [Función]

Similar a `fullmap`, pero `fullmapl` sólo hace mapeo sobre listas y matrices.

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

`functions` [Variable del sistema]

Valor por defecto: `[]`

La variable `functions` es una lista que contiene los nombres de las funciones ordinarias de Maxima. Una función ordinaria es aquella que ha sido construida mediante cualquiera de los métodos `define` o `:=` y que es invocada utilizando paréntesis. Una función puede definirse durante una sesión de Maxima o en un fichero que posteriormente será cargado en memoria por `load` o `batch`.

Las funciones array, que son invocadas con corchetes (`F[x]`), y las funciones subindicadas, que son las invocadas con corchetes y paréntesis (`F[x](y)`) se registran en la variable global `arrays`, no en `functions`.

Las funciones Lisp no se registran en ninguna lista.

Ejemplos:

```
(%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*) [Función]

Devuelve la definición de la función *f*.

Cada argumento puede ser el nombre de una macro (definida mediante `::=`), una función ordinaria (definida mediante `:=` o `define`), una función arreglo (definida mediante `:=` o `define`, pero encerrando los argumentos dentro de corchetes `[ ]`), una función de subíndice (definida mediante `:=` o `define`, pero encerrando algunos argumentos entre corchetes y otros entre paréntesis `( )`), una función de subíndice seleccionada por un subíndice variable, o una función de subíndice definida con un subíndice constante.

La función `fundef` no evalúa sus argumentos; el operador comilla-comilla `''` permite la evaluación.

La llamada de función `fundef (`*f*`)` devuelve la definición de *f*. Por el contrario, `dispfun (`*f*`)` crea una etiqueta intermedia y le asigna la definición a la etiqueta.

`funmake` (*F*, [*arg_1*, ..., *arg_n*]) [Función]

Devuelve una expresión *`F(arg_1, ..., arg_n)`*. El valor así retornado es simplificado pero no evaluado, de forma que la función *F* no es invocada, incluso cuando exista.

La función `funmake` no hace distinciones entre funciones array y funciones ordinarias; cuando *F* es el nombre de una función array, `funmake` devuelve *`F(...)`*, esto es, una llamada a función con paréntesis en lugar de corchetes. La función `arraymake` devuelve una llamada a función con corchetes para estos casos.

La función `funmake` evalúa sus argumentos.

Ejemplos:

La función `funmake` aplicada a una función ordinaria de 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)
```

La función `funmake` aplicada a una 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
```

La función `funmake` aplicada a una función subindicada.

```
(%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)
```

La función `funmake` aplicada a un símbolo que no está asociado a función alguna.

```
(%i1) funmake (A, [u]);
(%o1)                            A(u)
(%i2) ''%;
(%o2)                            A(u)
```

La función `funmake` evalúa sus argumentos, pero no el valor retornado.

```
(%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 el valor retornado de `funmake`.

```
(%i1) funmake (sin, [%pi / 2]);
(%o1)                           1
```

`lambda` ([*x_1*, ..., *x_m*], *expr_1*, ..., *expr_n*) [Función]
`lambda` ([[*L*]], *expr_1*, ..., *expr_n*) [Function]
`lambda` ([*x_1*, ..., *x_m*, [*L*]], *expr_1*, ..., *expr_n*) [Function]

Define y devuelve una expresión lambda (es decir, una función anónima). La función puede tener argumentos *x_1*, ..., *x_m* y/o argumentos opcionales *L*, que aparecerán dentro del cuerpo de la función como una lista. El valor que devuelve la función es *expr_n*. Una expresión lambda puede asignarse a una variable y ser evaluada como si fuese una función ordinaria. Además, puede aparecer en algunos contextos en los que sea necesario un nombre de función.

Cuando se evalúa la función, se crean las variables *x_1*, ..., *x_m* sin asignación de valores. Una función `lambda` puede aparecer dentro de un `block` o de otra `lambda`. Las variables locales se inicializan cada vez que se entra dentro de un nuevo bloque o de otra función `lambda`. Las variables locales se consideran globales dentro de un bloque o función `lambda` anidado dentro del primero. Si una variable es no local dentro de un bloque o función `lambda`, su valor es el que le corresponde en el bloque o función `lambda` superior. Este criterio se conoce con el nombre de "alcance dinámico".

Una vez establecidas las variables locales *expr_1* a *expr_n* son secuencialmente evaluadas. La variable especial `%%` representa el valor de la expresión inmediata anterior. Las sentencias `throw` y `catch` pueden aparecer también en la lista de expresiones.

La función `return` no puede aparecer en una expresión `lambda` a menos que se encuentre acotada dentro de un bloque (`block`), en cuyo caso `return` establece el valor de retorno del bloque, pero no de la expresión `lambda`, a menos que el bloque resulte ser precisamente *expr_n*. De igual manera, `go` no puede aparecer en una expresión `lambda` si no es dentro de un `block`.

Las funciones `lambda` no evalúan sus argumentos; el operador comilla-comilla `''` permite su evaluación.

Ejemplo:

- Una función lambda puede asignarse a una variable y ser evaluada como si fuese una función ordinaria.

```
(%i1) f: lambda ([x], x^2);
```

```
(%o1)                     lambda([x], x )
(%i2) f(a);
                                 2
(%o2)                           a
```

- Una expresión lambda puede aparecer en algunos contextos en los que sea necesario un nombre de función.

```
(%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 ]
```

- Los argumentos son variables locales. Otras variables se consideran globales. Las variables globales son evaluadas en el momento que lo es la expresión, a menos que la evaluación de las mismas sea forzada, como cuando se hace uso de ''.

```
(%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
```

- Las expresiones lambda pueden anidarse. Las variables locales de expresiones lambda exteriores se consideran globales en expresiones internas, a menos que se enmascaren con variables locales de igual nombre.

```
(%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
```

- Puesto que `lambda` no evalúa sus argumentos, la expresión lambda `i` de más abajo no define una función del tipo "multiplicar por `a`". Tal tipo de función se puede definir a través de `buildq`, como en la expresión lambda `i2` de más abajo.

```
(%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
```

- Una expresión lambda puede tener un número variable de argumentos, los cuales se indican mediante `[`*L*`]`, bien sea solo o como un último argumento. Estos argumentos aparecerán dentro del cuerpo de la función en forma de 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*) [Función]

La declaración `local(`*v_1*`, ..., `*v_m*`)` dentro de un bloque almacena las propiedades asociadas a los símbolos *v_1*, ..., *v_m*, borra cualesquiera otras propiedades antes de evaluar las expresiones y restaura las propiedades guardadas antes de abandonar el bloque.

Algunas declaraciones, como `:=`, `array`, `dependencies`, `atvalue`, `matchdeclare`, `atomgrad`, `constant`, `nonscalar`, `assume` y otras se implementan como propiedades de símbolos. El efecto producido por `local` consiste en hacer que tales declaraciones tengan efecto sólo dentro del bloque, en otro caso las declaraciones dentro del bloque tendrían un efecto global que afectarían al exterior de `block`.

La función `local` sólo puede usarse dentro de un `block`, en el cuerpo de definición de funciones o de expresiones `lambda` o en la función `ev`, siéndole permitido aparecer una sóla vez en cada una de ellas.

La función `local` no evalúa sus argumentos y devuelve `done`.

Ejemplo:

Definición local de una función.

```
(%i1) foo (x) := 1 - x;
```

```
(%o1)                     foo(x) := 1 - x
(%i2) foo (100);
(%o2)                           - 99
(%i3) block (local (foo), foo (x) := 2 * x, foo (100));
(%o3)                            200
(%i4) foo (100);
(%o4)                           - 99
```

`macroexpansion` [Variable opcional]

Valor por defecto: `false`

La variable `macroexpansion` controla si la expansión (esto es, el valor de retorno) de una función macro se sustituye por la llamada a la función macro. Una sustitución puede acelerar futuras evaluaciones de la expresión, bajo el coste que implica tener que almacenar la expansión.

`false` La expansión de una función macro no se sustituye por la llamada a la función macro.

`expand` La primera vez que se evalúa una llamada a función macro se almacena la expansión. De esta manera la expansión no se recalcula en llamadas posteriores; cualesquiera efectos laterales (como `print` o asignaciones a variables globales) tan solo tienen lugar la primera vez que la función macro es evaluada. La expansión en una expresión no afecta a otras expresiones que llamen a la misma función macro.

`displace` La primera vez que se evalúa una llamada a una función macro, la expansión se sustituye por la llamada, modificando así la expresión desde la que se hizo la llamada a la función macro. La expansión no se recalcula en llamadas posteriores; cualesquiera efectos laterales tan solo tienen lugar la primera vez que la función macro es evaluada. La expansión en una expresión no afecta a otras expresiones que llamen a la misma función macro.

Ejemplos:

Si `macroexpansion` vale `false`, una función macro es llamada cada vez que la expresión de llamada es evaluada.

```
(%i1) f (x) := h (x) / g (x);
                                      h(x)
(%o1)                      f(x) := ----
                                      g(x)
(%i2) g (x) ::= block (print ("x + 99 is equal to", x),
                       return (x + 99));
(%o2) g(x) ::= block(print("x + 99 is equal to", x),
                     return(x + 99))
(%i3) h (x) ::= block (print ("x - 99 is equal to", x),
                       return (x - 99));
(%o3) h(x) ::= block(print("x - 99 is equal to", x),
                     return(x - 99))
(%i4) macroexpansion: false;
```

```
(%o4)                          false
(%i5) f (a * b);
x - 99 is equal to x
x + 99 is equal to x
                             a b - 99
(%o5)                        --------
                             a b + 99
(%i6) dispfun (f);
                                   h(x)
(%t6)                      f(x) := ----
                                   g(x)

(%o6)                          done
(%i7) f (a * b);
x - 99 is equal to x
x + 99 is equal to x
                             a b - 99
(%o7)                        --------
                             a b + 99
```

Si `macroexpansion` vale `expand`, una función macro tan solo es llamada una vez.

```
(%i1) f (x) := h (x) / g (x);
                                   h(x)
(%o1)                      f(x) := ----
                                   g(x)
(%i2) g (x) ::= block (print ("x + 99 is equal to", x),
                       return (x + 99));
(%o2) g(x) ::= block(print("x + 99 is equal to", x),
                     return(x + 99))
(%i3) h (x) ::= block (print ("x - 99 is equal to", x),
                       return (x - 99));
(%o3) h(x) ::= block(print("x - 99 is equal to", x),
                     return(x - 99))
(%i4) macroexpansion: expand;
(%o4)                          expand
(%i5) f (a * b);
x - 99 is equal to x
x + 99 is equal to x
                             a b - 99
(%o5)                        --------
                             a b + 99
(%i6) dispfun (f);
                                   h(x)
(%t6)                      f(x) := ----
                                   g(x)

(%o6)                          done
```

```
(%i7) f (a * b);
                              a b - 99
(%o7)                         --------
                              a b + 99
```

Si `macroexpansion` vale `expand`, una función macro es llamada una vez y la expresión de llamada se modifica.

```
(%i1) f (x) := h (x) / g (x);
                                     h(x)
(%o1)                      f(x) := ----
                                     g(x)
(%i2) g (x) ::= block (print ("x + 99 is equal to", x), return (x + 99));
(%o2) g(x) ::= block(print("x + 99 is equal to", x),
                                                  return(x + 99))
(%i3) h (x) ::= block (print ("x - 99 is equal to", x), return (x - 99));
(%o3) h(x) ::= block(print("x - 99 is equal to", x),
                                                  return(x - 99))
(%i4) macroexpansion: displace;
(%o4)                         displace
(%i5) f (a * b);
x - 99 is equal to x
x + 99 is equal to x
                              a b - 99
(%o5)                         --------
                              a b + 99
(%i6) dispfun (f);
                                 x - 99
(%t6)                    f(x) := ------
                                 x + 99

(%o6)                           done
(%i7) f (a * b);
                              a b - 99
(%o7)                         --------
                              a b + 99
```

`mode_checkp` [Variable opcional]

Valor por defecto: `true`

Cuando `mode_checkp` vale `true`, `mode_declare` chequea los modos de las variables con valores asignados.

`mode_check_errorp` [Variable opcional]

Valor por defecto: `false`

Cuando `mode_check_errorp` vale `true`, `mode_declare` llama a error.

`mode_check_warnp` [Variable opcional]

Valor por defecto: `true`

Cuando `mode_check_warnp` vale `true`, se detallan los errores de modo.

`mode_declare` (*y_1*, *modo_1*, ..., *y_n*, *modo_n*) [Función]

La función `mode_declare` se utiliza para declarar los modos de variables y funciones para la ulterior traducción a Lisp o compilación de funciones. Se coloca habitualmente al comienzo de la definición de una función, de un script en Maxima o se ejecuta en tiempo real.

Los argumentos de `mode_declare` son pares formados por una variable y un modo, el cual debe ser `boolean`, `fixnum`, `number`, `rational` o `float`. Cada variable puede ser sustituida por una lista de variables, en cuyo caso todas ellas tendrán el mismo modo.

Código numérico que utilice arreglos puede ejecutarse más rápido declarando el tamaño que va a ocupar el arreglo, como en:

```
mode_declare (array (a [10, 10]), float)
```

para un arreglo de números en coma flotante de dimensiones 10 x 10.

Se puede declarar el modo del resultado de una función poniendo `function (f_1, f_2, ...)` como argumento; aquí `f_1`, `f_2`, ... son los nombres de las funciones. Por ejemplo, la expresión

```
mode_declare ([function (f_1, f_2, ...)], fixnum)
```

declara que el valor a devolver por `f_1`, `f_2`, ... son enteros de modo "single-word".

El nombre `modedeclare` es sinónimo de `mode_declare`.

`mode_identity` (*arg_1*, *arg_2*) [Función]

Es una forma especial usada con `mode_declare` y `macros` para declarar, por ejemplo, una lista de listas de números.

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

Desliga las definiciones de función de sus símbolos *f_1*, ..., *f_n*. Los argumentos pueden ser nombres de funciones ordinarias (creadas con `:=` o `define`) o de funciones macro (creadas con `::=`).

La instrucción `remfunction (all)` desliga todas las definiciones de funciones.

La función `remfunction` no evalúa sus argumentos.

La función `remfunction` devuelve una lista con los símbolos para los que la definición de función fue desligada. Devuelve `false` en el lugar de cualquier símbolo para el que no hay función definida.

La función `remfunction` no se puede aplicar a arrays de funciones ni a funciones subindicadas. Sí es aplicable en tales casos la función `remarray`.

`savedef` [Variable opcional]

Valor por defecto: `true`

Si `savedef` vale `true`, se mantiene la versión Maxima de una función definida por el usuario cuando ésta se traduce, lo que permite mostrar su código con `dispfun` y que la función pueda ser editada.

Si `savedef` vale `false`, los nombres de las funciones traducidas se eliminan de la lista `functions`.

`transcompile` [Variable opcional]

Valor por defecto: `true`

Si `transcompile` vale `true`, `translate` y `translate_file` generan declaraciones para hacer el código traducido más apto para la compilación.

La función `compfile` hace la asignación `transcompile: true`.

`translate` (*f_1*, ..., *f_n*) [Función]
`translate` (*functions*) [Función]
`translate` (*all*) [Función]

Traduce las funciones definidas por el usuario *f_1*, ..., *f_n* del lenguaje de Maxima a Lisp y evalúa las traducciones Lisp. Normalmente las funciones traducidas se ejecutan más rapidamente que las originales.

Las llamadas `translate (all)` o `translate (functions)` traducen todas las funciones de usuario.

Las funciones a ser traducidas deberían incluir una llamada a `mode_declare` al comienzo siempre que sea posible, a fin de producir código más eficiente. Por ejemplo:

```
f (x_1, x_2, ...) := block ([v_1, v_2, ...],
    mode_declare (v_1, modo_1, v_2, modo_2, ...), ...)
```

donde *x_1*, *x_2*, ... son los parámetros que se pasan a la función y *v_1*, *v_2*, ... son las variables locales.

Los nombres de las funciones traducidas son eliminados de la lista `functions` si `savedef` vale `false` (ver más abajo) y son añadidos a las listas `props`.

Las funciones no deberían ser traducidas hasta no estar completamente depuradas.

Se supone que las expresiones están simplificadas; en caso de no estarlo, se generará código correcto pero ineficiente. Así, el usuario no debería asignar a `simp` el valor `false`, el cual inhibe la simplificación de la expresión a ser traducida.

Cuando la variable `translate` vale `true`, se traducen automáticamente las funciones de usuario a Lisp.

Nótese que las funciones traducidas puede que no se ejecuten exactamente igual a como lo hacían antes de la traducción, debido a posibles incompatibilidades entre las versiones de Maxima y Lisp. En general, la función `rat` con más de un argumento y la función `ratvars` no deberían utilizarse si algunas de las variables son declaradas como expresiones racionales canónicas (CRE) mediante `mode_declare`. Además, la asignación `prederror: false` no traducirá.

Si `savedef` vale `true`, entonces la versión de Maxima de una función de usuario permanecerá cuando la función sea traducida por `translate`. Con esto se hace posible que se muestre la definición llamando a `dispfun` y que la función sea editada.

Si `transrun` vale `false` entonces las versiones interpretadas de todas las funciones serán ejecutadas en lugar de las versiones traducidas.

El resultado devuelto por `translate` es una lista con los nombres de las funciones traducidas.

`translate_file` (*nombre_fichero_maxima*) [Función]
`translate_file` (*nombre_fichero_maxima*, *nombre_fichero_lisp*) [Función]

Traduce un fichero en código Maxima a un fichero en código Lisp. La función `translate_file` devuelve una lista con los nombres de tres ficheros: el nombre del ficheero en Maxima, el nombre del fichero en Lisp y el nombre del fichero que contiene información adicional sobre la traducción. La función `translate_file` evalúa sus argumentos.

La llamada `translate_file ("foo.mac"); load("foo.LISP")` es lo mismo que `batch ("foo.mac")`, excepto por la presencia de ciertas restricciones, como el uso de `''` y `%`, por ejemplo.

La llamada `translate_file (`*nombre_fichero_maxima*`)` traduce un fichero en Maxima, *nombre_fichero_maxima*, a otro en Lisp de nombre similar. Por ejemplo, `foo.mac` se traduce en `foo.LISP`. El nombre del fichero en Maxima puede incluir el nombre de un directorio, en cuyo caso el fichero de salida Lisp se guardará en el mismo directorio desde el que se leyó la fuente Maxima.

La llamada `translate_file (`*nombre_fichero_maxima*`, `*nombre_fichero_lisp*`)` traduce el fichero Maxima *nombre_fichero_maxima* en el fichero Lisp *nombre_fichero_lisp*. La función `translate_file` ignora la extensión del fichero, en caso de que exista, de `nombre_fichero_lisp`; la extensión del fichero de salida Lisp será invariablemente `LISP`. El nombre del fichero Lisp puede incluir la ruta del directorio, en cuyo caso se almacenará en el directorio especificado.

La función `translate_file` también escribe un fichero de mensajes de avisos del traductor con diversos niveles de gravedad. La extensión de este fichero es `UNLISP`. Este fichero puede contener información valiosa, aunque de difícil interpretación, para detectar fallos en el código traducido. El fichero `UNLISP` se guarda siempre en el mismo directorio desde el que se leyó la fuente de Maxima.

La función `translate_file` emite código Lisp que incluye algunas declaraciones y definiciones que entran en efecto tan pronto como el código Lisp es compilado. Véase `compile_file` para más información sobre este particular.

Véanse también `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`, y `tr_warn_undefined_variable`.

`transrun` [Variable opcional]

Valor por defecto: `true`

Si `transrun` vale `false` entonces se ejecutarán las versiones interpretadas de todas las funciones, en lugar de las versiones traducidas.

`tr_array_as_ref` [Variable opcional]

Valor por defecto: `true`

Si `translate_fast_arrays` vale `false`, referencias de arreglos en el código Lisp creadas por `translate_file` se ven afectadas por `tr_array_as_ref`.

El valor de la variable `tr_array_as_ref` no tiene ningún efecto cuando `translate_fast_arrays` vale `true`.

`tr_bound_function_applyp` [Variable opcional]
Valor por defecto: `true`

Si `tr_bound_function_applyp` vale `true`, Maxima envía un aviso si encuentra una variable con valor asignado que está siendo utilizada como una función. `tr_bound_function_applyp` no influye en el código generado bajo estas circunstancias.

Por ejemplo, una expresión como `g (f, x) := f (x+1)` provocará un mensaje de esta naturaleza.

`tr_file_tty_messagesp` [Variable opcional]
Valor por defecto: `false`

Si `tr_file_tty_messagesp` vale `true`, los mensajes generados por `translate_file` durante la traducción de un fichero se muestran en la consola y se insertan en el fichero UNLISP. Si vale `false`, los mensajes sobre la traducción del fichero sólo se incorporan al fichero UNLISP.

`tr_float_can_branch_complex` [Variable opcional]
Valor por defecto: `true`

Le dice al traductor de Maxima a Lisp que las funciones `acos`, `asin`, `asec` y `acsc` pueden devolver valores complejos.

`tr_function_call_default` [Variable opcional]
Valor por defecto: `general`

El valor `false` significa llama a `meval`, `expr` significa que Lisp asignó los argumentos de la función, `general`, el valor por defecto, devuelve código apropiado para `mexprs` y `mlexprs` pero no para `macros`. La opción `general` asegura que las asignaciones de las variables son correctas en el código compilado. En modo `general`, cuando se traduce F(X), si F es una variable con valor, entonces se entiende que se quiere calcular `apply (f, [x])`, y como tal se traduce, con el apropiado aviso. No es necesario desactivar esto. Con los valores por defecto la falta de mensajes de aviso implica compatibilidad completa entre el código traducido y compilado con el interpretado por Maxima.

`tr_numer` [Variable opcional]
Valor por defecto: `false`

Si `tr_numer` vale `true` se utilizan las propiedades numéricas en aquellos átomos que las posean, como en `%pi`.

`tr_optimize_max_loop` [Variable opcional]
Valor por defecto: 100

El valor de `tr_optimize_max_loop` es el número máximo de veces que el traductor repetirá la macro-expansión y la optimización en el tratamiento de una expresión.

`tr_semicompile` [Variable opcional]
Valor por defecto: `false`

Si `tr_semicompile` vale `true`, las salidas de `translate_file` y `compfile` serán macro-expandidas pero no compiladas a código máquina por el compilador de Lisp.

`tr_state_vars` [Variable del sistema]

Valor por defecto:

```
[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]
```

Es la lista de variables que afectan la forma en que se obtiene la salida del código traducido. Esta información es útil para desarrolladores que pretendan corregir posibles fallos del traductor. Comparando el código traducido con el que se debería obtener bajo unas ciertas condiciones, es posible hacer el seguimiento de los fallos.

`tr_warnings_get ()` [Función]

Devuelve una lista con los avisos dados por el traductor.

`tr_warn_bad_function_calls` [Variable opcional]

Valor por defecto: `true`

Devuelve un aviso cuando se hacen llamadas a funciones que quizás no sean correctas debido a declaraciones inapropiadas realizadas durante la traducción.

`tr_warn_fexpr` [Variable opcional]

Valor por defecto: `compfile`

Devuelve un aviso si se encuentra con alguna FEXPR. Las FEXPR no deberían aparecer en el código traducido.

`tr_warn_meval` [Variable opcional]

Valor por defecto: `compfile`

Devuelve un aviso si la función `meval` es llamada. Si `meval` es invocada, es señal de la presencia de problemas en la traducción.

`tr_warn_mode` [Variable opcional]

Valor por defecto: `all`

Devuelve un aviso cuando a las variables se les asignan valores incompatibles con su modo.

`tr_warn_undeclared` [Variable opcional]

Valor por defecto: `compile`

Determina cuando enviar mensajes sobre variables no declaradas.

`tr_warn_undefined_variable` [Variable opcional]

Valor por defecto: `all`

Devuelve un aviso cuando se detectan variables globales no definidas.

`compile_file` (*`nombre_fich`*) [Función]
`compile_file` (*`nombre_fich`*, *`nombre_fich_compilado`*) [Función]
`compile_file` (*`nombre_fich`*, *`nombre_fich_compilado`*, *`nombre_fich_lisp`*) [Función]

Traduce el fichero Maxima *nombre_fich* a Lisp, ejecuta el compilador de Lisp y, en caso de ser exitosa la compilación, carga el código compilado en Maxima.

La función `compile_file` devuelve una lista con los nombres de tres ficheros: el fichero original en Maxima, la traducción Lisp, notas sobre la traducción y el código compilado. Si la compilación falla, el cuarto elemento es `false`.

Algunas declaraciones y definiciones entran en efecto tan pronto como el código Lisp es compilado (sin cargar el código compilado). Éstas incluyen funciones definidas con el operador `:=`, macros definidas con el operador `::=`, `alias`, `declare`, `define_variable`, `mode_declare` y `infix`, `matchfix`, `nofix`, `postfix`, `prefix` y `compfile`.

Asignaciones y llamadas a funciones no se evalúan hasta que el código compilado es cargado. En particular, dentro del fichero Maxima, asignaciones a los controles ("flags") de traducción (`tr_numer`, etc.) no tienen efecto durante la traducción.

El *nombre_fich* no puede contener sentencias del tipo `:lisp`.

La función `compile_file` evalúa sus argumentos.

`declare_translated` (*f_1*, *f_2*, ...) [Función]

Cuando se traduce un fichero de código Maxima a Lisp, es importante para el traductor saber qué funciones de las que están en el fichero van a ser llamadas como traducidas o compiladas, y cuáles son simplemente funciones Maxima o que no están definidas. Se genera el código `(MFUNCTION-CALL fn arg1 arg2 ...)` cuando el traductor no sabe si `fn` va a ser una función lisp.

37 Programación

37.1 Lisp y Maxima

Maxima fue escrito en Lisp, y es muy fácil tener acceso a funciones y variables Lisp desde Maxima y viceversa. Los símbolos Lisp y los símblos Maxima están claramente diferenciados por medio de una convención de nombres. Un símblo Lisp el cual comienza con un signo pesos `$` corresponde a un símbolo Maxima sin el signo pesos. Un símbolo Maxima el cual comienza con un signo de cierre de interrogación `?` corresponde a un símbolo Lisp sin dicho signo. Por ejemplo, el símbolo Maxima `foo` corresponde a el símbolo Lisp `$FOO`, mientras que el símbolo Maxima `?foo` corresponde a el símbolo Lisp `FOO`, tenga en cuenta que `?foo` esta escrito sin espacio entre `?` y `foo`; de otra manera se estaría invocando a `describe ("foo")`.

El guión `-`, asterisco `*`, u otros carácteres especiales en símbolos Lisp deben ser escritos mediante un backslash `\` si aparecen en código Maxima. Por ejemplo, el identificador Lisp `*foo-bar*` se debe escribir `?\*foo\-bar\*` en Maxima.

Se puede ejecutar código Lisp desde una sesión de Maxima. Una línea Lisp (que contenga una o más formas) puede ser ejecutada por medio de un comando especial `:lisp`. Por ejemplo,

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

se llama a la función Lisp `foo` con variables Maxima `x` y `y` como argumentos. La instrucción `:lisp` puede aparecer en el prompt interactivo o en un archivo que sea procesado por `batch` o `demo`, pero no en un archivo que sea procesado por `load`, `batchload`, `translate_file` o `compile_file`.

La función `to_lisp()` abre una sesión interactiva con el interprete Lisp. Escribiendo `(to-maxima)` se cierra la sesión con Lisp y se retorna a Maxima.

Las funciones y variables Lisp las cuales esten para ser visibles en Maxima como funciones y variables con nombres oridinarios (sin una puntuación especial), deben tener nombres tipo Lisp que comiencen con el signo pesos `$`.

Maxima distingue entre letras minúsculas y mayúsculas en identificadores. Existen algunas reglas que gobiernan la traducción de nombres entre Lisp y Maxima.

1. Un identificador Lisp que no se encuentra encerrado en barras verticales corresponde a un identificador Maxima en minúscula. Que el idenficador Lisp esté en mayúscula, minúscula o una combinación de ambas, no afecta en nada. Por ejemplo, los identificadores Lisp `$foo`, `$FOO`, y `$Foo`, todos corresponden al identificador Maxima `foo`. Esto es así porque `$foo`, `$FOO` y `$Foo` se convierten por defecto al símbolo `$FOO` de Lisp.
2. Un identificador Lisp el cual se encuentre todo en mayúscula o todo en minúscula y encerrado entre barras verticales corresponde a un identicador Maxima con el caso contrario. Esto es, de mayúsculas cambia a minúsculas y de minúsculas cambia a mayúsculas. E.g., el identificador Lisp `|$FOO|` y `|$foo|` corresponden los identificadores Maxima `foo` y `FOO`, respectivamente.
3. Un identificador Lisp el cual esta escrito mezclando letras mayúsculas y minúsculas y se encuentra entre barras verticales corresponde a un identificador Maxima con la misma escritura. E.g., el identificador Lisp `|$Foo|` corresponde a el identificador Maxima `Foo`.

La macro Lisp `#$` permite el uso de expresiones Maxima dentro de código Lisp. `#$`*expr*`$` extiende a una expresión Lisp equivalente a la expresión Maxima *expr*.

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

Esto tiene el mismo efecto que:

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

La función Lisp `displa` imprime una expresión en formato Maxima.

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

Las funciones definidas en Maxima no son funciones Lisp ordinarias. La función Lisp `mfuncall` llama a una función Maxima. Por ejemplo:

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

Algunas funciones Lisp son compartidas en el paquete Maxima, las cuales se listan a continuación:

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

37.2 Recolector de basura

La computación simbólica tiende a crear una buena cantidad de basura (resultados temporales que ya no serán utilizados), y un manejo efectivo de esto puede ser crucial para el término exitoso de algunos programas.

Bajo GCL (GNU Common Lisp), en aquellos sistemas UNIX donde la llamada al sistema mprotect está disponible (incluyendo SUN OS 4.0 y algunas variantes de BSD) se dispone de un recolector de basura estratificado. Véase la documentación de GCL para ALLOCATE y GBC. A nivel Lisp, ejecutando (setq si::*notify-gbc* t) pemitirá determinar qué áreas necesitan más espacio.

En cuanto al resto de Lisps bajo los que funciona Maxima, se remite al lector a la documentación correspondiente para controlar la recolección de basura.

37.3 Introducción a la programación

Maxima dispone de los bucles `do` para hacer iteraciones, así como estructuras más primitivas del estilo de `go`.

37.4 Funciones y variables para la programación

`backtrace` () [Función]
`backtrace` (*n*) [Función]

Devuelve la pila de llamadas, esto es, la lista de funciones que han llamado a la función actualmente activa.

La llamada a `backtrace()` devuelve la pila completa de llamadas.

Ejemplos:

```
(%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
```

La llamada `backtrace (`*n*`)` devuelve las *n* funciones más recientes, incluyendo a la función actualmente activa.

Ejemplos:

```
(%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]

La sentencia `do` se utiliza para realizar iteraciones. Debido a su generalidad la sentencia `do` se describirá en dos partes. En primer lugar se mostrará su forma más usual, análoga a la de otros lenguajes de programación (Fortran, Algol, PL/I, etc.); después se mencionarán otras formas de uso.

Hay tres variantes de esta sentencia que se diferencian entre sí únicamente por las condiciones de fin de bucle. Son las siguientes:

- `for` *variable*`:` *valor_inicial* `step` *incremento* `thru` *límite* `do` *cuerpo*
- `for` *variable*`:` *valor_inicial* `step` *incremento* `while` *condición* `do` *cuerpo*
- `for` *variable*`:` *valor_inicial* `step` *incremento* `unless` *condición* `do` *cuerpo*

El *valor_inicial*, el *incremento*, el *límite* y el *cuerpo* pueden ser cualquier tipo de expresión válida de Maxima. Si el incremento es igual a la unidad (1) entonces `"step 1"` puede omitirse.

La ejecución de la sentencia `do` se realiza asignando el valor_inicial a la variable (llamada de aquí en adelante variable-control). A continuación: (1) si la variable-control ha excedido el límite de la especificación dada por un `thru`, o si la condición impuesta por `unless` es verdadera (`true`), o si la condición dada por `while` es falsa (`false`) entonces la iteración `do` termina. (2) El cuerpo se evalúa. (3) El incremento es sumado a la variable-control. El proceso de (1) a (3) se repite hasta que la condición de fin de iteración se satisfaga. También es posible especificar varias condiciones de terminación del bucle, en cuyo caso `do` terminará cuando se satisfaga alguna de ellas.

En general la condición `thru` se satisfará cuando la variable-control sea mayor que el límite si el incremento es no negativo, o cuando la variable-control sea menor que el límite cuando el incremento es negativo. El incremento y el límite pueden ser expresiones no numéricas, tanto en cuanto esta desigualdad pueda quedar determinada. Sin embargo, a menos que el incremento sea un número negativo en el momento de comenzar el cómputo de `do`, Maxima supondrá que se evaluará a una cantidad positiva. En caso de no ser efectivamente positivo, la sentencia `do` puede dar un resultado inesperado.

Nótese que el límite, el incremento y la condición de terminación se evalúan en cada iteración del bucle. Así, si alguna de expresiones necesitan de muchos cálculos y devuelven un resultado que no va a cambiar durante toda la ejecución del cuerpo, será más eficiente dar este valor a una variable antes de comenzar la sentencia `do` y utilizarla luego durante su ejecución.

El valor que habitualmente devuelva la sentencia `do` será el átomo `done`. Sin embargo, la función `return` puede usarse dentro del cuerpo para salir de `do` de forma prematura retornando un valor determinado. Nótese no obstante que un `return` dentro de un `do` que está dentro de un bloque (`block`) provocará una salida de `do` pero no de `block`. Repárese también en que la función `go` no puede usarse para salir de `do` e ir a algún lugar de `block`.

La variable-control es siempre local respecto de `do`, por lo que se puede utilizar cualquier nombre de variable sin afectar el valor de cualquier otra variable externa a `do` y que tenga el mismo nombre. La variable-control no tendrá asignado ningún valor una vez se haya concluido el `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
```

Nótese que la condición `while i <= 10` es equivalente a `unless i > 10` y a `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
```

lo que da ocho términos del desarrollo de Taylor de la función `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
```

Este ejemplo calcula la raíz cuadrada negativa de 10 haciendo 10 iteraciones del método de Newton-Raphson. De no haberse alcanzado el criterio de convergencia el valor devuelto hubiese sido `done`.

En lugar de añadir siempre una cantidad a la variable-control a veces se puede querer que cambie en cada iteración siguiendo algún otro criterio. En tal caso se puede hacer uso de `next` *`expresión`* en lugar de `step` *`incremento`*. Esto hará que a la variable-control se le asigne el resultado de evaluar la expresión en cada iteración del bucle.

```
(%i6) for count: 2 next 3*count thru 20 do display (count)$
                              count = 2

                              count = 6

                             count = 18
```

En ocasiones puede interesar realizar una iteración en la que la variable-control no se utilice nunca. Se podrá entonces dar únicamente las condiciones de terminación del bucle omitiendo la inicialización y actualizando la información, tal como se hace en el siguiente ejemplo para calcular la raíz cuadrada de 5 utilizando un valor inicial alejado de la solución.

```
(%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
```

Si así se quiere, incluso es posible omitir las condiciones de terminación completamente y escribir únicamente `do` *body*, lo que provocará entrar en un bucle infinito. En tal caso, debería usarse la función `return` a fin de terminar con la ejecución de `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
```

(En este ejemplo, cuando se ejecuta `return` obliga a que sea `x` el valor devuelto por `do`. Al salirse del bloque, `x` es también el valor que devuelve `block` por ser `do` la última sentencia del bloque.)

Hay todavía otra forma de `do` en Maxima. Su sintaxis es:

`for` *variable* `in` *lista* *test_de_parada* `do` *cuerpo*

Los elementos de *list* son cualesquiera expresiones que se irán asignando sucesivamente a la variable en cada repetición del cuerpo. El test de parada *end_tests* (que es opcional) puede usarse para terminar la ejecución de `do`; de otro modo las iteraciones se pararán cuando la lista se haya agotado o cuando se ejecute un `return` dentro del cuerpo. (De hecho, la lista puede ser cualquier expresión no atómica, de la cual se irán extrayendo de forma sucesiva sus diferentes partes.)

```
(%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*) [Función]

Evalúa las expresiones *expr_1*, ..., *expr_n* una a una y devuelve `[`*expr_n*`]` (una lista) en caso de que no ocurra ningún error. En caso de aparecer algún error durante el cálculo de alguno de los argumentos, `errcatch` evita que el error se propague y devuelve la lista vacía `[]` sin evaluar más argumentos.

La función `errcatch` es útil en ficheros `batch` donde se sospeche que pueda aparecer algún error, el cual provocaría la terminación de la ejecución del `batch` de no ser previamente detectado.

`error` (*expr_1*, ..., *expr_n*) [Función]

`error` [Variable del sistema]

Calcula y devuelve *expr_1*, ..., *expr_n*, enviando posteriormente una seãl de error a Maxima o al `errcatch` más cercano.

A la variable `error` se le asigna una lista con la descripción del error. El primer elemento de `error` es una cadena de formato, la cual une todas las cadenas de los argumentos *expr_1*, ..., *expr_n*, siendo los demás elementos de la lista los valores de los argumentos que no son cadenas.

La llamada a `errormsg()` formatea e imprime `error`. Se reimprime así el mensaje de error más reciente.

`error_size` [Variable opcional]

Valor por defecto: 10

La variable `error_size` modifica los mensajes de error de acuerdo con el tamaño de las expresiones que aparecen en él. Si el tamaño de una expresión (tal como lo determina la función Lisp `ERROR-SIZE`) es mayor que `error_size`, la expresión se reemplaza en el mensaje por un símbolo, asignándole a éste una expresión. Los símbolos se toman de la lista `error_syms`.

En caso contrario, si la expresión es menor que `error_size`, la expresión se muestra en el propio mensaje.

Véanse también `error` y `error_syms`.

Ejemplo:

El tamaño de `U`, tal como lo determina `ERROR-SIZE`, es 24.

```
(%i1) U: (C^D^E + B + A)/(cos(X-1) + 1)$

(%i2) error_size: 20$

(%i3) error ("Example expression is", U);

Example expression is errexp1
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i4) errexp1;
                             E
                            D
                           C   + B + A
(%o4)                      --------------
                           cos(X - 1) + 1
(%i5) error_size: 30$

(%i6) error ("Example expression is", U);

                            E
                           D
                          C   + B + A
Example expression is --------------
                      cos(X - 1) + 1
 -- an error.  Quitting.  To debug this try debugmode(true);
```

`error_syms` [Variable opcional]

Valor por defecto: `[errexp1, errexp2, errexp3]`

En los mensajes de error, las expresiones mayores que `error_size` son reemplazadas por símbolos a los cuales se les asignas estas expresiones. Los símbolos se toman de la lista `error_syms`. La primera expresión que resulte ser demasiado larga se reemplaza por `error_syms[1]`, la segunda por `error_syms[2]` y así sucesivamente.

Si hay más expresiones largas que elementos en `error_syms`, los símbolos se construyen automáticamente, siendo el *n*-ésimo símbolo equivalente a `concat ('errexp, n)`.

Véanse también `error` y `error_size`.

`errormsg ()` [Función]

Reimprime el mensaje de error más reciente. La variable `error` guarda el mensaje y `errormsg` lo formatea e imprime.

`errormsg` [Variable opcional]

Valor por defecto: `true`

Cuando `errormsg` vale `false` se suprimen los contenidos de los mensajes de error.

La variable `errormsg` no se puede asignar a un valor local dentro de un bloque. El valor global de `errormsg` está siempre presente.

Ejemplos:

```
(%i1) errormsg;
(%o1)                                true
(%i2) sin(a,b);
Wrong number of arguments to sin
 -- an error. To debug this try: debugmode(true);
(%i3) errormsg:false;
(%o3)                                false
(%i4) sin(a,b);

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

La variable `errormsg` no se puede asignar a un valor local dentro de un bloque.

```
(%i1) f(bool):=block([errormsg:bool],
                     print ("value of errormsg is",errormsg))$
(%i2) errormsg:true;
(%o2)                                true
(%i3) f(false);
value of errormsg is true
(%o3)                                true
(%i4) errormsg:false;
(%o4)                                false
(%i5) f(true);
value of errormsg is false
(%o5)                                false
```

`for` [Operador especial]

Utilizado en las iteraciones. Véase `do` para una descripción de las técnicas de iteración en Maxima.

`go (`*`etiqueta`*`)` [Función]

Se utiliza dentro de un bloque (`block`) para transferir el control a la sentencia del bloque que esté etiquetada con el argumento de `go`. Una sentencia queda etiquetada cuando está precedida por un argumento de tipo átomo como cualquier otra sentencia de `block`. Por ejemplo:

```
block ([x], x:1, tururu, x+1, ..., go(tururu), ...)
```

El argumento de `go` debe ser el nombre de una etiqueta que aparezca en el mismo bloque (`block`). No se puede utilizar `go` para transferir el control a un bloque que no sea aquel que contenga la sentencia `go`.

`if` [Operador especial]

Evaluación condicionada. Se reconocen varias formas de expresiones `if`.

La expresión `if` *`cond_1`* `then` *`expr_1`* `else` *`expr_0`* devuelve *expr_1* si *cond_1* vale `true`, en caso contrario la respuesta es `expr_0`.

La expresión `if` *`cond_1`* `then` *`expr_1`* `elseif` *`cond_2`* `then` *`expr_2`* `elseif ... else` *`expr_0`* devuelve *expr_k* si *cond_k* vale `true` y todas las condiciones anteriores toman el valor `false`. Si ninguna de las condiciones vale `true`, la respuesta es `expr_0`.

La falta de un `else` final se interpreta como un `else false`; esto es, la expresión `if` *`cond_1`* `then` *`expr_1`* equivale a `if` *`cond_1`* `then` *`expr_1`* `else false`, y `if` *`cond_1`* `then` *`expr_1`* `elseif ... elseif` *`cond_n`* `then` *`expr_n`* equivale a su vez a `if` *`cond_1`* `then` *`expr_1`* `elseif ... elseif` *`cond_n`* `then` *`expr_n`* `else false`.

Las alternativas *expr_0*, ..., *expr_n* pueden ser expresiones válidas de Maxima, incluidas expresiones `if` anidadas. Las alternativas ni se simplifican ni se evalúan, a menos que su condición asociada valga `true`.

Las condiciones *cond_1*, ..., *cond_n* deben ser expresiones capaces de dar como resultado `true` o `false` al ser evaluadas. Si en un momento dado una condición no da como resultado un valor de verdad (`true` o `false`), el comportamiento de `if` se controla con la variable global `prederror`. Si `prederror` vale `true`, se considera un error que la condición evaluada no dé como resultado un valor de verdad; en caso contrario, las condiciones que no den como resultado un valor de verdad se aceptan, dándose el resultado como una expresión condicional.

Las condiciones pueden contener operadores lógicos y relacionales, así como otros elementos, tal como se indica a continuación:

Operación	Símbolo	Tipo
menor que	<	operador relacional infijo
menor o igual que	<=	operador relacional infijo
igualdad (sintáctica)	=	operador relacional infijo
negación de =	#	operador relacional infijo
igualdad (por valor)	equal	operador relacional infijo
negación de equal	notequal	operador relacional infijo
mayor o igual que	>=	operador relacional infijo
mayor que	>	operador relacional infijo
y	and	operador lógico infijo
o	or	operador lógico infijo
no	not	operador lógico prefijo

`map` (*f*, *expr_1*, ..., *expr_n*) [Función]

Devuelve una expresión cuyo operador principal es el mismo que aparece en las expresiones *expr_1*, ..., *expr_n* pero cuyas subpartes son los resultados de aplicar *f* a cada una de las subpartes de las expresiones; *f* puede ser tanto el nombre de una función de *n* argumentos como una expresión `lambda` de *n* argumentos.

Uno de los usos que tiene `map` es la de aplicar (o mapear) una función (por ejemplo, `partfrac`) sobre cada término de una expresión extensa en la que normalmente no se podría utilizar la función debido a insuficiencias en el espacio de almacenamiento durante el curso de un cálculo.

```
(%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]
```

Véase también `maperror` .

`mapatom` (*expr*) [Función]

Devuelve `true` si y sólo *expr* es tratado por las rutinas de mapeo como un átomo.

`maperror` [Variable opcional]

Valor por defecto: `true`

Cuando `maperror` toma el valor `false`, hace que todas las funciones de mapeo, como por ejemplo

```
map (f, expr_1, expr_2, ...)
```

(1) paren cuando hayan terminado de procesar la *expr_i* más corta, a menos que todas ellas sean del mismo tamaño y (2) apliquen `f` a `[expr_1, expr_2, ...]` si es el caso que las `expr_i` no son todas del mismo tipo de objeto.

Cuando `maperror` toma el valor `true` entonces se emite un mensaje de error cuando se presenta cualquiera de los dos casos anteriores.

`mapprint` [Variable opcional]

Valor por defecto: `true`

Si `mapprint` vale `true`, se producirán ciertos mensajes por parte de las funciones `map`, `mapl` y `fullmap` en determinadas situaciones, como cuando `map` hace uso de `apply`.

Si `mapprint` vale `false`, no se emitirán tales mensajes.

`maplist` (*f*, *expr_1*, ..., *expr_n*) [Función]

Devuelve una lista con las aplicaciones de *f* a las partes de las expresiones *expr_1*, ..., *expr_n*; *f* es el nombre de una función ou una expresión lambda.

La función `maplist` difiere de `map (f, expr_1, ..., expr_n)`, la cual devuelve una expresión con el mismo operador principal que tenga *expr_i*, excepto en simplificaciones y en el caso en el que `map` hace un `apply`.

`prederror` [Variable opcional]

Valor por defecto: `false`

Cuando `prederror` toma el valor `true`, se emite un mensaje de error siempre que el predicado de una sentencia `if` o de una función `is` no se pueda evaluar ni a verdadero (`true`) ni a falso (`false`).

Si toma el valor `false`, se devuelve bajo las mismas circunstancias anteriores el valor `unknown`. El modo `prederror: false` no está soportado en el código traducido; sin embargo, `maybe` está soportado en código traducido.

Véanse también `is` y `maybe`.

`return` (*valor*) [Función]

Puede utilizarse para salir de un bloque, devolviendo su argumento. Véase `block` para más información.

`scanmap` (*f*, *expr*) [Función]
`scanmap` (*f*, *expr*, *bottomup*) [Función]

Aplica recursivamente *f* sobre *expr*, de arriba hacia abajo. Esto es más útil cuando se busca una factorización completa, por ejemplo:

```
(%i1) exp:(a^2+2*a+1)*y + x^2$
(%i2) scanmap(factor,exp);
                                        2      2
(%o2)                            (a + 1)  y + x
```

Nótese que cómo `scanmap` aplica la función dada `factor` a las subexpresiones que forman a *expr*; si se presenta otra forma de *expr* a `scanmap` entonces el resultado puede ser diferente. Así, `%o2` no se restaura cuando `scanmap` se aplica a la forma expandida de exp:

```
(%i3) scanmap(factor,expand(exp));
                              2                   2
(%o3)                         a  y + 2 a y + y + x
```

Aquí hay otro ejemplo de la forma en que `scanmap` aplica recursivamente una función dada a todas las subexpresiones, incluyendo exponentes:

```
(%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 abajo hacia arriba. Por ejemplo, para `f` no definida,

```
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))
```

En este caso se obtiene la misma respuesta por cualquiera de los dos métodos.

`throw` (*expr*) [Función]

Evalúa *expr* y devuelve el valor del `catch` más reciente. La función `throw` se utiliza junto con `catch` como un mecanismo de retorno no local.

`while` [Operador especial]

`unless` [Operador especial]

Véase `do`.

`outermap` (*f*, *a_1*, ..., *a_n*) [Función]

Aplica la función *f* a cada uno de los elementos del producto vectorial *a_1* por *a_2* ... por *a_n*.

El argumento *f* debe ser el nombre de una función de *n* argumentos, o una expresión lambda de *n* argumentos. Cada uno de los argumentos *a_k* puede ser una lista, una lista anidada, una matriz o cualquier otro tipo de expresión.

El valor devuelto por `outermap` es una estructura anidada. Si *x* es la respuesta dada por `outermap`, entonces tiene la misma estructura que la primera lista, lista anidada o matriz, `x[i_1]...[i_m]` tiene la misma estructura que la segunda lista, lista anidada o matriz, `x[i_1]...[i_m][j_1]...[j_n]` tiene la misma estructura que la tercera lista, lista anidada o matriz, y así sucesivamente, siendo *m*, *n*, ... los números índice necesarios para acceder a los elementos de cada argumento: uno para las listas, dos para las matrices y uno o más para las listas anidadas. Aquellos argumentos que no sean listas ni matrices no tienen efecto alguno sobre la estructura del valor retornado.

Nótese que el efecto producido por `outermap` es diferente del que se obtiene al aplicar *f* a cada uno de los elementos del producto devuelto por `cartesian_product`. La función `outermap` mantiene la estructura de los argumentos en la respuesta, miemtras que `cartesian_product` no lo hace.

La función `outermap` evalúa sus argumentos.

Véanse también `map`, `maplist` y `apply`.

Ejemplos:

Ejemplos elementales de uso de `outermap`. Con el fin de mostrar con mayor claridad las combinaciones del argumento, se mantiene sin definir `F`.

```
(%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]]
```

El siguiente ejemplo permite hacer un análisis más profundo del valor retornado por `outermap`. Los tres primeros argumentos son una matriz, una lista y otra matriz, en este orden. El valor devuelto es una matriz, cuyos elementos son listas y cada elemento de cada una de estas listas es a su vez una 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]
```

La función `outermap` mantiene la estructura de los argumentos en su respuesta, mientras que `cartesian_product` no lo hace.

```
(%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
```

38 Depurado

38.1 Depuración del código fuente

Maxima es capaz de dar asistencia en la depuración del código fuente. Un usuario puede establecer un punto de referencia dentro del código de una función a partir del cual se siga la ejecución línea a línea. La compliación puede ser posteriormente examinada, conjuntamente con los valores que se han ido asignando a las variables.

La instrucción `:help`, o `:h`, muestra la lista de comandos para la depuración. (En general, los comandos pueden abreviarse; en algunos casos la lista de alternativas podrá ser listada.) Dentro del depurador, el usuario podrá examinar también cualquier función propia de Maxima, definirla y manipular variables y expresiones.

El punto de referencia se establecerá con la instrucción `:br`. Ya dentro del depurador, el usuario podrá avanzar una línea de cada vez utilizando la instrucción `:n` (de "next", en inglés). La orden `:bt` (de "backtrace") muestra la lista de la pila. Finalmente, con el comando `:r` ("resume") se abandona el depurador continuando con la ejecución. El uso de estas instrucciones se muestra en el siguiente ejemplo.

```
(%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                  <-- pulsando :bt se retrocede
#0: foo(y=5)(foobar.mac line 1)
#1: bar(x=2,y=3)(foobar.mac line 9)

(dbm:1) :n                   <-- pulsando :n se avanza una línea
(foobar.mac 2)
/tmp/foobar.mac:2::

(dbm:1) :n                   <-- pulsando :n se avanza otra línea
(foobar.mac 3)
/tmp/foobar.mac:3::

(dbm:1) u;                   <-- se pide el valor de u
28

(dbm:1) u: 33;               <-- se cambia el valor de u a 33
33
```

```
(dbm:1) :r                    <-- pulsando :r se termina la depuración

(%o2)                                 1094
```

El fichero `/tmp/foobar.mac` contiene lo siguiente:

```
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 DEL DEPURADOR EN EMACS

Si el usuario está corriendo el código bajo GNU emacs en un entorno de texto (dbl shell), o está ejecutando el entorno gráfico `xmaxima`, entonces cuando una función pare en el punto de referencia, podrá observar su posición actual en el archivo fuente, el cual será mostrado en la otra mitad de la ventana, bien resaltada en rojo, o con una pequeña flecha apuntando a la línea correcta. El usuario puede avanzar líneas simples tecleando M-n (Alt-n).

Bajo Emacs se debe ejecutar el programa en una ventana de texto `dbl`, la cual requiere el archivo `dbl.el` que está en el directorio elisp. El usuario debe instalar los archivos elisp o agregar el directorio elisp de Maxima a la ruta de búsqueda: por ejemplo, se puede añadir lo siguiente al archivo `.emacs` o al `site-init.el`

```
(setq load-path (cons "/usr/share/maxima/5.9.1/emacs" load-path))
(autoload 'dbl "dbl")
```

entonces en emacs

```
M-x dbl
```

debería abrir una ventana del sistema en la cual se pueden ejecutar programas, por ejemplo Maxima, gcl, gdb, etc. En esta ventana también se puede ejecutar el depurador, mostrando el código fuente en la otra ventana.

El usuario puede colocar un punto de referencia en una línea determinada sin más que teclear `C-x space`. Con esto se le hace saber al depurador en qué función está el cursor y en qué línea del mismo. Si el cursor está en la línea 2 de `foo`, entonces insertará en la otra ventana la instrucción "`:br foo 2`", a fin de detener `foo` justo en la segunda línea. Para tener esto operativo, el usuario debe tener activo maxima-mode.el (modo-maxima.el) en la ventana en la que está `foobar.mac`. Hay otros comandos disponibles en la ventana, como evaluar la función dentro de Maxima tecleando `Alt-Control-x`.

38.2 Claves de depuración

Las claves de depuración son palabras que no son interpretadas como expresiones de Maxima. Una clave de depuración puede introducirse dentro de Maxima o del depurador. Las

claves de depuración comienzan con dos puntos, ':'. Por ejemplo, para evaluar una expresión Lisp, se puede teclear `:lisp` seguido de la expresión a ser evaluada.

```
(%i1) :lisp (+ 2 3)
5
```

El número de argumentos depende del comando en particular. Además, tampoco es necesario teclear el nombre completo de la instrucción, tan solo lo justo para diferenciarla de las otras instrucciones. Así, `:br` sería suficiente para `:break`.

Las claves de depuración se listan a continuación.

`:break F n`
: Establece un punto de referencia en la función `F` en la línea `n` contando a partir del comienzo de la función. Si `F` es una cadena, entonces se entiende que se trata de un fichero, siendo entonces `n` el número de línea a partir del comienzo del fichero. El valor `n` es opcional; en caso de no ser suministrado, se entenderá que vale cero (primera línea de la función o fichero).

`:bt`
: Retrocede en la pila.

`:continue`
: Continua el cómputo de la función.

`:delete`
: Borra los punto de referencia especificados, o todos si no se especifica ninguno.

`:disable`
: Deshabilita los puntos de referencia especificados, o todos si no se especifica ninguno.

`:enable`
: Habilita los puntos de referencia especificados, o todos si no se especifica ninguno.

`:frame n`
: Imprime el elemento `n` de la pila, o el actualmente activo si no se especifica ninguno.

`:help`
: Imprime la ayuda sobre un comando del depurador, o de todos los comandos si no se especifica ninguno.

`:info`
: Imprime información sobre un elemento.

`:lisp expresión`
: Evalúa la `expresión` Lisp.

`:lisp-quiet expresión`
: Evalúa la `expresión` Lisp sin devolver el resultado.

`:next`
: Como `:step`, excepto que `:next` se salta las llamadas a funciones.

`:quit`
: Sale del nivel actual del depurador sin completar el cómputo.

`:resume`
: Continúa con el cómputo.

`:step`
: Sigue con el cómputo de la función o fichero hasta que alcance una nueva línea fuente.

`:top`
: Retorna a Maxima desde cualquier nivel del depurador sin completar el cómputo.

38.3 Funciones y variables para depurado

`debugmode` [Variable opcional]

Valor por defecto: `false`

Cuando en Maxima ocurre un error, Maxima inicializará el depurador si `debugmode` tiene el valor `true`. El usuario puede ingresar comandos para examinar la pila de llamadas, los puntos de interrupción; en pocas palabras ir a través del código de Maxima. Vea `debugging` para una lista de los comandos del depurador.

Habilitando `debugmode` no se capturarán los errores tipo Lisp.

`refcheck` [Variable opcional]

Valor por defecto: `false`

Cuando `refcheck` vale `true`, Maxima imprime un mensaje cada vez que una variable es utilizada por vez primera en un cálculo.

`setcheck` [Variable opcional]

Valor por defecto: `false`

Cuando el valor de `setcheck` es una lista de variables (se admite que tengan subíndices) Maxima devuelve un mensaje indicando si los valores que han sido asignados a las variables lo han sido con el operador ordinario `:`, o con el operador de asignación `::` o como resultado de haberse realizado una llamada de función, pero en ningún caso cuando la asignación haya sido hecha mediante los operadores `:=` o `::=`. El mensaje contiene el nombre de la variable y su valor.

La variable `setcheck` admite también los valores `all` o `true` con lo que el informe incluirá todas las variables.

Cada nueva asignación de `setcheck` establece una nueva lista de variables a ser monitorizada, de forma que cualquier otra variable previamente asignada a `setcheck` es olvidada.

Los nombres asignados a `setcheck` deben estar precedidos del apóstrofo `'` a fin de evitar que las variables sean evaluadas antes de ser almacenadas en `setcheck`. Por ejemplo, si `x`, `y` y `z` ya guardan algún valor entoces se hará

```
setcheck: ['x, 'y, 'z]$
```

para colocarlas en la lista de variables a monitorizar.

No se generará ninguna salida cuando una variable de la lista `setcheck` sea asignada a ella misma, como en `X: 'X`.

`setcheckbreak` [Variable opcional]

Valor por defecto: `false`

Si `setcheckbreak` es igual `true`, Maxima se detendrá siempre que a una variable de la lista `setcheck` se le asigne un nuevo valor. La detención tendrá lugar justo antes de hacerse la asignación. En ese momento `setval` guarda el valor que se le va a dar a la variable. Entonces el usuario podrá darle un valor diferente pasándoselo a la variable `setval`.

Véanse también `setcheck` y `setval`.

`setval` [Variable del sistema]

Guarda el valor que va a ser asignado a una variable cuando `setcheckbreak` realiza una detención. Entonces se podrá asignarle otro valor pasándoselo previamente a `setval`.

Véanse también `setcheck` y `setcheckbreak`.

`timer` (*f_1*, ..., *f_n*) [Función]
`timer` (*all*) [Función]
`timer` () [Función]

Dadas las funciones *f_1*, ..., *f_n*, `timer` coloca cada una de ellas en la lista de funciones para las cuales se generarán estadísticas relativas al tiempo de cómputo. Así, `timer(f)$ timer(g)$` coloca a `f` y luego a `g` en dicha lista de forma acumulativa.

La sentencia `timer(all)` coloca todas las funciones de usuario (las referenciadas por la variable global `functions`) en la lista de funciones cuyos tiempos de ejecución se quieren monitorizar.

Si no se le pasan argumentos a `timer` se obtendrá la lista de funciones cuyos tiempos de ejecución se quieren monitorizar.

Maxima almacena la duración del cómputo de cada función de la lista, de forma que `timer_info` devolverá las estadísticas correspondientes, incluyendo el tiempo medio de cada llamada a la función, el número de llamadas realizadas y el tiempo total transcurrido. La instrucción `untimer` borra las funciones de la lista.

La función `timer` no evalúa sus argumentos, de forma que `f(x) := x^2$ g:f$ timer(g)$` no coloca a `f` en la lista.

Si `trace(f)` está activada, entonces `timer(f)` está desactivada; `trace` y `timer` no pueden estar operativas al mismo tiempo.

Véase también `timer_devalue`.

`untimer` (*f_1*, ..., *f_n*) [Función]
`untimer` () [Función]

Dadas las funciones *f_1*, ..., *f_n*, `untimer` las elimina de la lista de funciones cuyos tiempos de ejecución se quiere monitorizar.

Si no se le suministran argumentos, `untimer` borra completamente la lista.

Tras la ejecución de `untimer (f)`, `timer_info (f)` aún devuelve las estadísticas de tiempo previamente registradas, pero `timer_info()` (sin argumentos) no devuelve información sobre aquellas funciones que ya no están en la lista. La ejecución de `timer (f)` inicializa todas las estadísticas a cero y coloca `f` nuevamente en la lista.

`timer_devalue` [Variable opcional]

Valor por defecto: `false`

Si `timer_devalue` es igual a `true`, Maxima le resta a cada función cuyos tiempos de ejecución se quiere monitorizar el tiempo gastado en llamadas a otras funciones presentes también en la lista de monitorización. En caso contrario, los tiempos que se obtienen para cada función incluyen también los consumidos en otras funciones. Nótese que el tiempo consumido en llamadas a otras funciones que no están en la lista de monitorización no se resta del tiempo total.

Véanse también `timer` y `timer_info`.

`timer_info` (*f_1*, ..., *f_n*) [Función]
`timer_info` () [Función]

Dadas las funciones *f_1*, ..., *f_n*, `timer_info` devuelve una matriz con información relativa a los tiempos de ejecución de cada una de estas funciones. Sin argumentos, `timer_info` devuelve la información asociada a todas las funciones cuyos tiempos de ejecución se quiere monitorizar.

La matriz devuelta por `timer_info` incluye los nombres de las funciones, tiempo de ejecución en cada llamada, número de veces que ha sido llamada, tiempo total de ejecución y tiempo consumido en la recolección de basura, `gctime` (del inglés, "garbage collection time") en la versión original de Macsyma, aunque ahora toma el valor constante cero.

Los datos con los que `timer_info` construye su respuesta pueden obtenerse también con la función `get`:

```
get(f, 'calls);  get(f, 'runtime);  get(f, 'gctime);
```

Véase también `timer`.

`trace` (*f_1*, ..., *f_n*) [Función]
`trace` (*all*) [Función]
`trace` () [Función]

Dadas las funciones *f_1*, ..., *f_n*, `trace` imprime información sobre depuración cada vez que estas funciones son llamadas; `trace(f)$ trace(g)$` coloca de forma acumulativa a `f` y luego a `g` en la lista de funciones a ser rastradas.

La sentencia `trace(all)` coloca todas las funciones de usuario (las referenciadas por la variable global `functions`) en la lista de funciones a ser rastreadas.

Si no se suministran argumentos, `trace` devuelve una lista con todas las funciones a ser rastreadas.

La función `untrace` desactiva el rastreo. Véase también `trace_options`.

La función `trace` no evalúa sus argumentos, de forma que `f(x) := x^2$ g:f$ trace(g)$` no coloca a `f` en la lista de rastreo.

Cuando una función se redefine es eliminada de la lista de rastreo. Así, tras `timer(f)$ f(x) := x^2$`, la función `f` dejará de estar en dicha lista.

Si `timer (f)` está activado, entonces `trace (f)` está desactivado, ya que `trace` y `timer` no pueden estar ambos activos para la misma función.

`trace_options` (*f*, *option_1*, ..., *option_n*) [Función]
`trace_options` (*f*) [Función]

Establece las opciones de rastreo para la función *f*. Cualquier otra opción previamente especificada queda reemplazada por las nuevas. La ejecución de `trace_options (f, ...)` no tiene ningún efecto, a menos que se haya invocado previamente a `trace (f)` (es indiferente que esta invocación sea anterior o posterior a `trace_options`).

`trace_options (f)` inicializa todas las opciones a sus valores por defecto.

Las claves de opciones son:

- `noprint`: No se imprime mensaje alguno ni a la entrada ni a la salida de la función.

- `break`: Coloca un punto de referencia antes de que la función comience a ejecutarse y otro después de que termine su ejecución. Véase `break`.
- `lisp_print`: Muestra los argumentos y valores retornados como objetos de Lisp.
- `info`: Imprime `-> true` tanto a la entrada como a la salida de la función.
- `errorcatch`: Detecta errores, otorgando la posibilidad de marcar un error, reintentar la llamada a la función o especificar un valor de retorno.

Las opciones de rastreo se especifican de dos formas. La única presencia de la clave de opción ya activa la opción. (Nótese que la opción *foo* no se activa mediante *`foo`*`: true` u otra forma similar; se tendrá en cuenta también que las claves no necesitan ir precedidas del apóstrofo.) Especificando la clave de opción junto con una función de predicado se hace que la opción quede condicionada al predicado.

La lista de argumentos para las funciones de predicado es siempre `[level, direction, function, item]` donde `level` es el nivel de recursión para la función, `direction` puede ser tanto `enter` como `exit`, `function` es el nombre de la función y `item` es la lista de argumentos (a la entrada) o el valor de retorno (a la salida).

A continuación un ejemplo de opciones de rastreo no condicionales:

```
(%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);
```

Para la misma función, con la opción `break` condicionada a un 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*) [Función]
`untrace` () [Función]

Dadas las funciones *f_1*, ..., *f_n*, `untrace` desactiva el rastreo previamente activado por la función `trace`. Si no se aportan argumentos, `untrace` desactiva el rastreo de todas las funciones.

La llamada a `untrace` devuelve una lista con las funciones para las que el rastreo se ha desactivado.

39 augmented_lagrangian

39.1 Funciones y variables para augmented_lagrangian

`augmented_lagrangian_method` (*FOM*, *xx*, *C*, *yy*) [Función]
`augmented_lagrangian_method` (*FOM*, *xx*, *C*, *yy*, *optional_args*) [Función]
`augmented_lagrangian_method` ([*FOM*, *grad*], *xx*, *C*, *yy*) [Función]
`augmented_lagrangian_method` ([*FOM*, *grad*], *xx*, *C*, *yy*, *optional_args*) [Función]

Devuelve una aproximación del valor mínimo de la expresión *FOM* respecto de las variables *xx*, manteniendo las restricciones *C* igual a cero. La lista *yy* contiene las soluciones iniciales para *xx*. El algoritmo que se utiliza es el método del lagrangiano aumentado (ver referencias [1] y [2]).

Si *grad* está presente en la llamada a la función, se interpreta como el gradiente de *FOM* respecto de *xx*, representado como una lista de tantas expresiones como variables tenga *xx*. Si el argumento *grad* no está, se calculará de forma automática.

Tanto *FOM* como cada uno de los elementos de *grad*, si se da como argumento, deben ser expresiones ordinarias; no admitiéndose ni nombres de funciones ni expresiones lambda.

El argumento `optional_args` hace referencia a otros argumentos adicionales, los cuales se especifican de la forma *`symbol = value`*. Los argumentos opcionales reconocidos son:

`niter` Número de iteraciones del algoritmo.

`lbfgs_tolerance`
Tolerancia que se pasa a LBFGS.

`iprint` Parámetro IPRINT (lista de dos enteros que controlan la frecuencia de mensajes) que se pasa a LBFGS.

`%lambda` Valor inicial de `%lambda` que será utilizado para calcular el lagrangiano aumentado.

Esta función minimiza el lagrangiano aumentado haciendo uso del algoritmo LBFGS, que es un método de los llamados quasi-Newton.

Antes de hacer uso de esta función ejecútense `load("augmented_lagrangian")`.

Véase también `lbfgs`.

Referencias:

[1] http://www-fp.mcs.anl.gov/otc/Guide/OptWeb/continuous/constrained/nonlinearcon/auglag.html

[2] http://www.cs.ubc.ca/spider/ascher/542/chap10.pdf

Ejemplos:

```
(%i1) load ("lbfgs");
(%o1)     /maxima/share/lbfgs/lbfgs.mac
(%i2) load ("augmented_lagrangian");
(%o2)
```

```
   /maxima/share/contrib/augmented_lagrangian.mac
(%i3) FOM: x^2 + 2*y^2;
                                2    2
(%o3)                        2 y  + x
(%i4) xx: [x, y];
(%o4)                         [x, y]
(%i5) C: [x + y - 1];
(%o5)                       [y + x - 1]
(%i6) yy: [1, 1];
(%o6)                         [1, 1]
(%i7) augmented_lagrangian_method(FOM, xx, C, yy, iprint=[-1,0]);
(%o7) [[x = 0.66665984108002, y = 0.33334027245545],
                                  %lambda = [- 1.333337940892525]]
```

Mismo ejemplo que en el caso anterior, pero ahora el gradiente se suministra como argumento.

```
(%i1) load ("lbfgs")$
(%i2) load ("augmented_lagrangian")$
(%i3) FOM: x^2 + 2*y^2;
                                2    2
(%o3)                        2 y  + x
(%i4) FOM: x^2 + 2*y^2;
                                2    2
(%o4)                        2 y  + x
(%i5) xx: [x, y];
(%o5)                         [x, y]
(%i6) grad : [2*x, 4*y];
(%o6)                       [2 x, 4 y]
(%i7) C: [x + y - 1];
(%o7)                       [y + x - 1]
(%i8) yy: [1, 1];
(%o8)                         [1, 1]
(%i9) augmented_lagrangian_method ([FOM, grad], xx, C, yy,
                                   iprint = [-1, 0]);
(%o9) [[x = 0.666659841080025, y = .3333402724554462],
                                  %lambda = [- 1.333337940892543]]
```

40 Bernstein

40.1 Funciones y variables para Bernstein

`bernstein_poly` (*k*, *n*, *x*) [Función]

Si `k` no es un entero negativo, los polinomios de Bernstein se definen como `bernstein_poly(k,n,x) = binomial(n,k) x^k (1-x)^(n-k)`; en cambio, si `k` es un entero negativo, el polinomio de Bernstein `bernstein_poly(k,n,x)` se anula. Cuando o bien `k` o `n` no son enteros, la variable opcional `bernstein_explicit` controla la expansión de los polinomios de Bernstein a su forma explícita.

Ejemplo:

```
(%i1) load("bernstein")$

(%i2) bernstein_poly(k,n,x);
(%o2)                bernstein_poly(k, n, x)
(%i3) bernstein_poly(k,n,x), bernstein_explicit : true;
                                        n - k  k
(%o3)            binomial(n, k) (1 - x)      x
```

Los polinomios de Bernstein tienen definidas su derivada e integral:

```
(%i4) diff(bernstein_poly(k,n,x),x);
(%o4) (bernstein_poly(k - 1, n - 1, x)
                                   - bernstein_poly(k, n - 1, x)) n
(%i5) integrate(bernstein_poly(k,n,x),x);
(%o5)
                                                                 k + 1
 hypergeometric([k + 1, k - n], [k + 2], x) binomial(n, k) x
 ------------------------------------------------------------------
                                k + 1
```

Cuando los argumentos contienen números decimales en coma flotante, los polinomios de Bernstein también devuelven resultados decimales.

```
(%i6) bernstein_poly(5,9, 1/2 + %i);
                        39375 %i   39375
(%o6)                   -------- + -----
                          128       256
(%i7) bernstein_poly(5,9, 0.5b0 + %i);
(%o7)           3.076171875b2 %i + 1.5380859375b2
```

Para hacer uso de `bernstein_poly`, ejecútese primero `load("bernstein")`.

`bernstein_explicit` [Variable opcional]

Valor por defecto: `false`

Cuando o bien `k` o `n` no son enteros, la variable opcional `bernstein_explicit` controla la expansión de los polinomios de Bernstein a su forma explícita.

Ejemplo:

```
(%i1) bernstein_poly(k,n,x);
```

```
(%o1)                        bernstein_poly(k, n, x)
(%i2) bernstein_poly(k,n,x), bernstein_explicit : true;
                                                 n - k  k
(%o2)                    binomial(n, k) (1 - x)      x
```

Cuando tanto `k` como `n` son enteros, `bernstein(k,n,x)` se expande siempre a su forma explícita.

`multibernstein_poly` (*[k1,k2,...,kp]*,*[n1,n2,..., np]*,*[x1,x2,..., xp]*) [Función]

La sentencia `multibernstein_poly` (*[k1,k2,...,kp]*,*[n1,n2,..., np]*,*[x1,x2,..., xp]*) es el producto de polinomios de Bernstein `bernstein_poly(k1,n1,x1) bernstein_poly(k2,n2,x2) ... bernstein_poly(kp,np,xp)`.

Para hacer uso de `multibernstein_poly`, ejecútese primero `load("bernstein")`.

`bernstein_approx` (*f*,*[x1,x1,...,xn]*,*n*) [Función]

Devuelve el polinomio de Bernstein uniforme de `n`-ésimo orden que aproxima la función `(x1,x2,..xn) |--> f`.

Ejemplos:

```
(%i1) bernstein_approx(f(x),[x], 2);
                          2       1                          2
(%o1)             f(1) x  + 2 f(-) (1 - x) x + f(0) (1 - x)
                                2
(%i2) bernstein_approx(f(x,y),[x,y], 2);
                 2  2       1                2                  2  2
(%o2) f(1, 1) x  y  + 2 f(-, 1) (1 - x) x y  + f(0, 1) (1 - x)  y
                           2
           1   2                 1  1
 + 2 f(1, -) x  (1 - y) y + 4 f(-, -) (1 - x) x (1 - y) y
           2                     2  2
           1         2                      2        2
 + 2 f(0, -) (1 - x)  (1 - y) y + f(1, 0) x  (1 - y)
           2
       1                       2                  2        2
 + 2 f(-, 0) (1 - x) x (1 - y)  + f(0, 0) (1 - x)  (1 - y)
       2
```

Para hacer uso de `bernstein_approx`, ejecútese primero `load("bernstein")`.

`bernstein_expand` (*e*, *[x1,x1,...,xn]*) [Función]

Expresa el polinomio `e` como una combinación lineal de polinomios de Bernstein multivariantes.

```
(%i1) bernstein_expand(x*y+1,[x,y]);
(%o1)           2 x y + (1 - x) y + x (1 - y) + (1 - x) (1 - y)
(%i2) expand(%);
(%o2)                               x y + 1
```

Maxima devuelve un error si el primer argumento no es un polinomio.

Para hacer uso de `bernstein_expand`, ejecútese primero `load("bernstein")`.

41 bode

41.1 Funciones y variables para bode

`bode_gain` (*H*, *range*, *...plot_opts...*) [Función]

Función para dibujar el gráfico de ganancia de Bode.

Ejemplos (1 a 7 de

http://www.swarthmore.edu/NatSci/echeeve1/Ref/Bode/BodeHow.html,

8 de Ron Crummett):

```
(%i1) load("bode")$

(%i2) H1 (s) := 100 * (1 + s) / ((s + 10) * (s + 100))$

(%i3) bode_gain (H1 (s), [w, 1/1000, 1000])$

(%i4) H2 (s) := 1 / (1 + s/omega0)$

(%i5) bode_gain (H2 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i6) H3 (s) := 1 / (1 + s/omega0)^2$

(%i7) bode_gain (H3 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i8) H4 (s) := 1 + s/omega0$

(%i9) bode_gain (H4 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i10) H5 (s) := 1/s$

(%i11) bode_gain (H5 (s), [w, 1/1000, 1000])$

(%i12) H6 (s) := 1/((s/omega0)^2 + 2 * zeta * (s/omega0) + 1)$

(%i13) bode_gain (H6 (s), [w, 1/1000, 1000]),
                  omega0 = 10, zeta = 1/10$

(%i14) H7 (s) := (s/omega0)^2 + 2 * zeta * (s/omega0) + 1$

(%i15) bode_gain (H7 (s), [w, 1/1000, 1000]),
                  omega0 = 10, zeta = 1/10$

(%i16) H8 (s) := 0.5 / (0.0001 * s^3 + 0.002 * s^2 + 0.01 * s)$

(%i17) bode_gain (H8 (s), [w, 1/1000, 1000])$
```

Antes de hacer uso de esta función ejecútese `load("bode")`. Véase también `bode_phase`.

`bode_phase` (*H*, *range*, *...plot_opts...*) [Función]

Función para dibujar el gráfico de fase de Bode.

Ejemplos (1 a 7 de

http://www.swarthmore.edu/NatSci/echeeve1/Ref/Bode/BodeHow.html,

8 de Ron Crummett):

```
(%i1) load("bode")$

(%i2) H1 (s) := 100 * (1 + s) / ((s + 10) * (s + 100))$

(%i3) bode_phase (H1 (s), [w, 1/1000, 1000])$

(%i4) H2 (s) := 1 / (1 + s/omega0)$

(%i5) bode_phase (H2 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i6) H3 (s) := 1 / (1 + s/omega0)^2$

(%i7) bode_phase (H3 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i8) H4 (s) := 1 + s/omega0$

(%i9) bode_phase (H4 (s), [w, 1/1000, 1000]), omega0 = 10$

(%i10) H5 (s) := 1/s$

(%i11) bode_phase (H5 (s), [w, 1/1000, 1000])$

(%i12) H6 (s) := 1/((s/omega0)^2 + 2 * zeta * (s/omega0) + 1)$

(%i13) bode_phase (H6 (s), [w, 1/1000, 1000]),
                   omega0 = 10, zeta = 1/10$

(%i14) H7 (s) := (s/omega0)^2 + 2 * zeta * (s/omega0) + 1$

(%i15) bode_phase (H7 (s), [w, 1/1000, 1000]),
                   omega0 = 10, zeta = 1/10$

(%i16) H8 (s) := 0.5 / (0.0001 * s^3 + 0.002 * s^2 + 0.01 * s)$

(%i17) bode_phase (H8 (s), [w, 1/1000, 1000])$

(%i18) block ([bode_phase_unwrap : false],
              bode_phase (H8 (s), [w, 1/1000, 1000]));
```

```
(%i19) block ([bode_phase_unwrap : true],
               bode_phase (H8 (s), [w, 1/1000, 1000]));
```

Antes de hacer uso de esta función ejecútese `load("bode")`. Véase también `bode_gain`.

42 cobyla

42.1 Introducción a cobyla

`fmin_cobyla` es una traducción a Common Lisp hecha con el programa `f2cl` de la rutina Fortran COBYLA, (Powell, [1][2][3]), para optimización con restricciones.

COBYLA minimiza una función objetivo F(X) sujeta a M restricciones con desigualdades de la forma g(X) >= 0 sobre X, siendo X un vector de variables de N componentes.

Las restricciones en forma de igualdades se pueden implementar por pares de desigualdades de la forma g(X)>=0 y -g(X)>= 0. El interfaz Maxima para COBYLA admite restricciones de igualdad, transformándolas luego internamente a pares de desigualdades.

El algoritmo hace uso de aproximaciones lineales, tanto de la función objetivo como de las funciones de restricción; tales aproximaciones se hacen mediante interpolación lineal de N+1 puntos en el espacio de variables. Los puntos de interpolación se consideran vértices de un simplejo o símplex. El parámetro RHO controla el tamaño del simplejo y se reduce automáticamente de RHOBEG a RHOEND. Para cada RHO la subrutina intenta alcanzar un buen vector de variables para el tamaño actual, reduciéndose entonces RHO hasta alcanzar el valor de RHOEND. Por eso, tanto a RHOBEG como a RHOEND se les deben asignar valores razonables, lo que requiere cierto trabajo empírico previo. La rutina trata cada restricción individualmente cuando se calcula un en las variables. El nombre de la rutina se deriva de la frase *Constrained Optimization BY Linear Approximations*.

Referecias:

[1] Código Fortran procede de http://plato.asu.edu/sub/nlores.html#general

[2] M. J. D. Powell, "A direct search optimization method that models the objective and constraint functions by linear interpolation," en Advances in Optimization and Numerical Analysis, eds. S. Gomez and J.-P. Hennart (Kluwer Academic: Dordrecht, 1994), p. 51-67.

[3] M. J. D. Powell, "Direct search algorithms for optimization calculations," Acta Numerica 7, 287-336 (1998). Also available as University of Cambridge, Department of Applied Mathematics and Theoretical Physics, Numerical Analysis Group, Report NA1998/04 from https://web.archive.org/web/20160607190705/http://www.damtp.cam.ac.uk:80/user/na/reports.html

42.2 Funciones y variables para cobyla

`fmin_cobyla` (*F*, *X*, *Y*) [Función]
`fmin_cobyla` (*F*, *X*, *Y*, *optional_args*) [Función]

Devuelve una aproximación del valor mínimo de la expresión *F* respecto de las variables *X*, sujeta a un conjunto opcional de restricciones. *Y* es una lista que contiene una solución semilla inicial en *X*.

F debe ser una expresión ordinaria, no valen nombres de funciones ni expresiones lambda.

`optional_args` hace referencia a argumentos adicionales, que se especifican de la forma *`symbol = value`*. Los argumentos opcionales que se reconocen son:

`constraints`
: Lista de restricciones en forma de desigualdades e igualdades que debe satisfacer *X*. Las desigualdades deben ser de la forma `g(X) >= h(X)` o `g(X) <= h(X)`. Las restricciones de igualdad deben ser de la forma `g(X) = h(X)`.

`rhobeg`
: Valor inicial de la variable interna RHO, que controla el tamaño del simplejo. Su valor por defecto es 1.0.

`rhoend`
: El valor final deseado para el parámetro RHO. Es aproximadamente la precisión de las variables. Su valor por defecto es 1d-6.

`iprint`
: Nivel de información de salida. Su valor por defecto es 0.
  - 0 - Sin información de salida
  - 1 - Sumario al final de los cálculos
  - 2 - Se van mostrando los nuevos valores de RHO y SIGMA, incluyendo el vector de variables.
  - 3 - Como en 2, pero la información se muestra cuando se calcula F(X).

`maxfun`
: Número máximo de evaluaciones de la función. Su valor por defecto es 1000.

El resultado devuelto es un vector:

1. Los valores de las variables con las que se alcanza el valor mínimo. Es una lista de elementos de la forma *`var = value`* para cada una de las variables listadas en *X*.
2. El valor mínimo de la función objetivo.
3. El número de evaluaciones de la función.
4. Código de retorno con los siguientes significados:
   1. 0 - No ha habido errores.
   2. 1 - Alcanzado el máximo número permitido de evaluaciones de la función.
   3. 2 - Errores de redondeo han impedido el avance del proceso.

El código `load("fmin_cobyla")` carga en memoria esta función..

`bf_fmin_cobyla` (*F*, *X*, *Y*) [Función]
`bf_fmin_cobyla` (*F*, *X*, *Y*, *optional_args*) [Función]

Esta función es idéntica a `fmin_cobyla`, excepto por el hecho de que utiliza aritmética de precisión arbitraria (bigfloat) y que el valor por defecto de *rhoend* es `10^(fpprec/2)`.

Véase `fmin_cobyla`.

El código `load("fmin_cobyla")` carga en memoria esta función..

42.3 Ejemplos para cobyla

Minimizar x1*x2 bajo la condición 1-x1^2-x2^2 >= 0. La solución teórica es x1 = 1/sqrt(2), x2 = -1/sqrt(2).

```
(%i1) load("fmin_cobyla")$

(%i2) fmin_cobyla(x1*x2, [x1, x2], [1,1], constraints = [x1^2+x2^2<=1], iprint=1);

   Normal return from subroutine COBYLA

   NFVALS =   66   F =-5.000000E-01    MAXCV = 1.999845E-12
   X = 7.071058E-01  -7.071077E-01
(%o2) [[x1 = 0.70710584934848, x2 = - 0.7071077130248], - 0.49999999999926,
       [[-1.999955756559757e-12],[]], 66]
```

Hay más ejemplos en el directorio `share/cobyla/ex`.

43 contrib_ode

43.1 Introducción a contrib_ode

La función `ode2` de Maxima resuelve ecuaciones diferenciales ordinarias (EDO) simples de primer y segundo orden. La función `contrib_ode` extiende las posibilidades de `ode2` con métodos adicionales para ODEs lineales y no lineales de primer orden y homogéneas lineales de segundo orden. El código se encuentra en estado de desarrollo y la syntaxis puede cambiar en futuras versiones. Una vez el código se haya estabilizado podrá pasar a integrarse dentro de Maxima.

El paquete debe cargarse con la instrucción `load("contrib_ode")` antes de utilizarlo.

La sintaxis de `contrib_ode` es similar a la de `ode2`. Necesita tres argumentos: una EDO (sólo se necesita el miembro izquierdo si el derecho es igual cero), la variable dependiente y la independiente. Si encuentra la solución, devolverá una lista de resultados.

La forma de los resultados devueltos es diferente de la utilizada por `ode2`. Puesto que las ecuaciones no lineales pueden tener múltiples soluciones, `contrib_ode` devuelve una lista de soluciones. Las soluciones pueden tener diferentes formatos:

- una función explícita para la variable dependiente,
- una función implícita para la variable dependiente,
- una solución paramétrica en términos de la variable `%t` o
- una transformación en otra EDO de variable `%u`.

`%c` hace referencia a la constante de integración en las ecuaciones de primer orden. `%k1` y `%k2` son las constantes para las ecuaciones de segundo orden. Si por cualquier razón contrib_ode no pudiese encontrar una solución, devolverá `false`, quizás después de mostrar un mensaje de error.

Ejemplos:

En ocasiones es necesario devolver una lista de soluciones, pues algunas EDOs pueden tener múltiples soluciones:

```
(%i1) load("contrib_ode")$

(%i2) eqn:x*'diff(y,x)^2-(1+x*y)*'diff(y,x)+y=0;

                     dy 2             dy
(%o2)              x (--)  - (x y + 1) -- + y = 0
                     dx               dx
(%i3) contrib_ode(eqn,y,x);

                                                 x
(%o3)               [y = log(x) + %c, y = %c %e ]
(%i4) method;

(%o4)                          factor
```

Las EDOs no lineales pueden tener soluciones singulares sin constantes de integración, como en la segunda solución del ejemplo siguiente:

```
(%i1) load("contrib_ode")$

(%i2) eqn:'diff(y,x)^2+x*'diff(y,x)-y=0;

                        dy 2     dy
(%o2)                  (--)  + x -- - y = 0
                        dx       dx
(%i3) contrib_ode(eqn,y,x);

                                              2
                                   2         x
(%o3)              [y = %c x + %c , y = - --]
                                             4
(%i4) method;

(%o4)                          clairault
```

La siguiente ODE tiene dos soluciones paramétricas en términos de la variable `%t`. En este caso, las soluciones paramétricas se pueden manipular para dar soluciones explícitas.

```
(%i1) load("contrib_ode")$

(%i2) eqn:'diff(y,x)=(x+y)^2;

                           dy          2
(%o2)                      -- = (y + x)
                           dx
(%i3) contrib_ode(eqn,y,x);

(%o3) [[x = %c - atan(sqrt(%t)), y = - x - sqrt(%t)],
                     [x = atan(sqrt(%t)) + %c, y = sqrt(%t) - x]]
(%i4) method;

(%o4)                          lagrange
```

En el siguiente ejemplo (Kamke 1.112) se obtiene una solución implícita.

```
(%i1) load("contrib_ode")$

(%i2) assume(x>0,y>0);

(%o2)                       [x > 0, y > 0]
(%i3) eqn:x*'diff(y,x)-x*sqrt(y^2+x^2)-y;

                       dy            2    2
(%o3)                x -- - x sqrt(y  + x ) - y
                       dx
(%i4) contrib_ode(eqn,y,x);
```

```
                                        y
(%o4)                  [x - asinh(-) = %c]
                                        x
(%i5) method;

(%o5)                           lie
```

La siguiente ecuación de Riccati se transforma en una EDO lineal de segundo orden de variable `%u`. Maxima es incapaz de resolver la nueva EDO, por lo que la devuelve si resolver:

```
(%i1) load("contrib_ode")$

(%i2) eqn:x^2*'diff(y,x)=a+b*x^n+c*x^2*y^2;

                    2 dy      2  2      n
(%o2)              x  -- = c x  y  + b x  + a
                      dx
(%i3) contrib_ode(eqn,y,x);

                d%u
                ---                                 2
                dx        2     n - 2   a     d %u
(%o3)  [[y = - ----, %u c  (b x      + --) + ---- c = 0]]
                %u c                     2      2
                                        x     dx
(%i4) method;

(%o4)                         riccati
```

Para EDOs de primer orden, `contrib_ode` llama a `ode2`. Entonces trata de aplicar los siguientes métodos: factorización, Clairault, Lagrange, Riccati, Abel y Lie. El método de Lie no se intenta aplicar a las ecuaciones de Abel si el propio método de Abel no obtiene solución, pero sí se utiliza si el método de Riccati devuelve una EDO de segundo orden sin resolver.

Para EDOs de segundo orden, `contrib_ode` llama a `ode2` y luego a `odelin`.

Se mostrarán mensajes de depurado si se ejecuta la sentencia `put('contrib_ode,true,'verbose)`.

43.2 Funciones y variables para contrib_ode

`contrib_ode` (*eqn*, *y*, *x*) [Función]

Devuelve la lista de soluciones de la ecuación diferencia ordinaria (EDO) *eqn* de variable independiente *x* y variable dependiente *y*.

`odelin` (*eqn*, *y*, *x*) [Función]

La función `odelin` resulve EDOs homogéneas lineales de primer y segundo orden con variable independiente *x* y variable dependiente *y*. Devuelve un conjunto fundamental de soluciones de la EDO.

Para EDOs de segundo orden, `odelin` utiliza un método desarrollado por Bronstein y Lafaille, que busca las soluciones en términos de funciones especiales dadas.

```
(%i1) load("contrib_ode");

(%i2) odelin(x*(x+1)*'diff(y,x,2)+(x+5)*'diff(y,x,1)+(-4)*y,y,x);
...trying factor method
...solving 7 equations in 4 variables
...trying the Bessel solver
...solving 1 equations in 2 variables
...trying the F01 solver
...solving 1 equations in 3 variables
...trying the spherodial wave solver
...solving 1 equations in 4 variables
...trying the square root Bessel solver
...solving 1 equations in 2 variables
...trying the 2F1 solver
...solving 9 equations in 5 variables
       gauss_a(- 6, - 2, - 3, - x)  gauss_b(- 6, - 2, - 3, - x)
(%o2) {--------------------------, --------------------------}
                    4                            4
                   x                            x
```

`ode_check` (*eqn*, *soln*) [Función]

Devuelve el valor de la ecuación diferencia ordinaria (EDO) *eqn* después de sustituir una posible solución *soln*. El valor es cero si *soln* es una solución de *eqn*.

```
(%i1) load("contrib_ode")$

(%i2) eqn:'diff(y,x,2)+(a*x+b)*y;

                          2
                         d y
(%o2)                    --- + (a x + b) y
                           2
                         dx
(%i3) ans:[y = bessel_y(1/3,2*(a*x+b)^(3/2)/(3*a))*%k2*sqrt(a*x+b)
         +bessel_j(1/3,2*(a*x+b)^(3/2)/(3*a))*%k1*sqrt(a*x+b)];

                                    3/2
                      1  2 (a x + b)
(%o3) [y = bessel_y(-, --------------) %k2 sqrt(a x + b)
                      3       3 a
                                         3/2
                              1  2 (a x + b)
                  + bessel_j(-, --------------) %k1 sqrt(a x + b)]
                              3       3 a
(%i4) ode_check(eqn,ans[1]);
```

```
(%o4)                          0
```

`method` [Variable opcional]
A la variable `method` se le asigna el método aplicado.

`%c` [Variable]
`%c` es la constante de integración para EDOs de primer orden.

`%k1` [Variable]
`%k1` es la primera constante de integración para EDOs de segundo orden.

`%k2` [Variable]
`%k2` es la segunda constante de integración para EDOs de segundo orden.

`gauss_a` (*a*, *b*, *c*, *x*) [Función]
`gauss_a(a,b,c,x)` y `gauss_b(a,b,c,x)` son funciones geométricas 2F1 . Representan dos soluciones independientes cualesquiera de la ecuación diferencial hipergeométrica `x(1-x) diff(y,x,2) + [c-(a+b+1)x] diff(y,x) - aby = 0` (A&S 15.5.1).

El único uso que se hace de estas funciones es en las soluciones de EDOs que devuelven `odelin` y `contrib_ode`. La definición y utilización de estas funciones puede cambiar en futuras distribuciones de Maxima.

Véanse también `gauss_b`, `dgauss_a` y `gauss_b`.

`gauss_b` (*a*, *b*, *c*, *x*) [Función]
Véase también `gauss_a`.

`dgauss_a` (*a*, *b*, *c*, *x*) [Función]
The derivative with respect to x of `gauss_a(a,b,c,x)`.

`dgauss_b` (*a*, *b*, *c*, *x*) [Función]
Derivada de `gauss_b(a,b,c,x)` respecto de *x*.

`kummer_m` (*a*, *b*, *x*) [Función]
Función M de Kummer, tal como la definen Abramowitz y Stegun, *Handbook of Mathematical Functions*, Sección 13.1.2.

El único uso que se hace de esta función es en las soluciones de EDOs que devuelven `odelin` y `contrib_ode`. La definición y utilización de estas funciones puede cambiar en futuras distribuciones de Maxima.

Véanse también `kummer_u`, `dkummer_m` y `dkummer_u`.

`kummer_u` (*a*, *b*, *x*) [Función]
Función U de Kummer, tal como la definen Abramowitz y Stegun, *Handbook of Mathematical Functions*, Sección 13.1.3.

Véase también `kummer_m`.

`dkummer_m` (*a*, *b*, *x*) [Función]
Derivada de `kummer_m(a,b,x)` respecto de *x*.

`dkummer_u` (*a*, *b*, *x*) [Función]
Derivada de `kummer_u(a,b,x)` respecto de *x*.

43.3 Posibles mejoras a contrib_ode

Este paquete aún se encuentra en fase de desarrollo. Aspectos pendientes:

- Extender el método FACTOR `ode1_factor` para que trabaje con raíces múltiples.
- Extender el método FACTOR `ode1_factor` para que intente resolver factores de orden superior. En este momento sólo intenta resolver factores lineales.
- Modificar la rutina LAGRANGE `ode1_lagrange` para que prefiera raíces reales a las complejas.
- Añadir más métodos para las ecuaciones de RIccati.
- Mejorar la identificación de las ecuaciones de Abel de segunda especie. El procedimiento actual no es muy bueno.
- Trabajar la rutina del grupo simétrico de Lie `ode1_lie`. Existen algunos problemas: algunas partes no están implementadas, algunos ejemplos no terminan de ejecutarse, otros producen errors, otros devuelven respuestas muy complejas.
- Hacer más pruebas.

43.4 Pruebas realizadas con contrib_ode

Los procedimientos fueron probados con cerca de mil ecuaciones tomadas de Murphy, Kamke, Zwillinger y otros. Éstas se encuentran en el directorio de pruebas.

- La rutina de Clairault `ode1_clairault` encuentra todas las soluciones conocidas, incluídas las singulares, de las ecuaciones de Clairault en Murphy y Kamke.
- Las otras rutinas a veces devuelven una sola solución cuando existen más.
- Algunas de las soluciones devueltas por `ode1_lie` son demasiado complejas e imposibles de interpretar.
- A veces se producen detenciones imprevistas del procedimiento.

43.5 Referencias para contrib_ode

1. E. Kamke, Differentialgleichungen Losungsmethoden und Losungen, Vol 1, Geest & Portig, Leipzig, 1961
2. G. M. Murphy, Ordinary Differential Equations and Their Solutions, Van Nostrand, New York, 1960
3. D. Zwillinger, Handbook of Differential Equations, 3rd edition, Academic Press, 1998
4. F. Schwarz, Symmetry Analysis of Abel's Equation, Studies in Applied Mathematics, 100:269-294 (1998)
5. F. Schwarz, Algorithmic Solution of Abel's Equation, Computing 61, 39-49 (1998)
6. E. S. Cheb-Terrab, A. D. Roche, Symmetries and First Order ODE Patterns, Computer Physics Communications 113 (1998), p 239. (`http://lie.uwaterloo.ca/papers/ode_vii.pdf`)
7. E. S. Cheb-Terrab, T. Kolokolnikov, First Order ODEs, Symmetries and Linear Transformations, European Journal of Applied Mathematics, Vol. 14, No. 2, pp. 231-246 (2003). (`http://arxiv.org/abs/math-ph/0007023`, `http://lie.uwaterloo.ca/papers/ode_iv.pdf`)

8. G. W. Bluman, S. C. Anco, Symmetry and Integration Methods for Differential Equations, Springer, (2002)
9. M Bronstein, S Lafaille, Solutions of linear ordinary differential equations in terms of special functions, Proceedings of ISSAC 2002, Lille, ACM Press, 23-28. (http://www-sop.inria.fr/cafe/Manuel.Bronstein/publications/issac2002.pdf)

44 descriptive

44.1 Introducción a descriptive

El paquete `descriptive` contiene funciones para realizar cĺculos y gráficos estadísticos descriptivos. Junto con el código fuente se distribuyen tres conjuntos de datos: `pidigits.data`, `wind.data` y `biomed.data`.

Cualquier manual de estadística se puede utilizar como referencia al paquete `descriptive`.

Para comentarios, fallos y sugerencias, por favor contactar con *'riotorto AT yahoo DOT com'*.

Aquí un sencillo ejemplo sobre cómo operan las funciones de `descriptive`, dependiendo de la naturaleza de sus argumentos, listas o matrices,

```
(%i1) load ("descriptive")$
(%i2) /* muestra univariate */   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) /* muestra multivariante */ mean (%);
                          e + c + a  f + d + b
(%o4)                    [---------, ---------]
                              3          3
```

Nótese que en las muestras multivariantes la media se calcula para cada columna.

En caso de varias muestras de diferente tamaño, la función `map` de Maxima puede utilizarse para obtener los resultados deseados para cada muestra,

```
(%i1) load ("descriptive")$
(%i2) map (mean, [[a, b, c], [d, e]]);
                           c + b + a  e + d
(%o2)                     [---------, -----]
                              3        2
```

En este caso, dos muestras de tamaños 3 y 2 han sido almacenadas en una lista.

Muestras univariantes deben guardarse en listas como en

```
(%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]
```

y muestras multivariantes en matrices como las del siguiente ejemplo

```
(%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  ]
```

En este caso, el número de columnas es igual al de la dimensión de la variable aleatoria y el número de filas coincide con el tamaño muestral.

Los datos pueden suministrarse manualmente, pero las muestras grandes se suelen almacenar en ficheros de texto. Por ejemplo, el fichero `pidigits.data` contiene los 100 primeros dígitos del número `%pi`:

```
3
1
4
1
5
9
2
6
5
3 ...
```

A fin de leer estos dígitos desde Maxima,

```
(%i1) s1 : read_list (file_search ("pidigits.data"))$
(%i2) length (s1);
(%o2)                          100
```

Por otro lado, el archivo `wind.data` contiene los promedios diarios de la velocidad del viento en cinco estaciones meteorológicas en Irlanda (esta muestra es parte de un conjunto de datos correspondientes a 12 estaciones meteorológicas. El fichero original se puede descargar libremente del 'StatLib Data Repository' y se analiza en 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). Así se leen los datos:

```
(%i1) s2 : read_matrix (file_search ("wind.data"))$
(%i2) length (s2);
(%o2)                          100
(%i3) s2 [%]; /* last record */
(%o3)             [3.58, 6.0, 4.58, 7.62, 11.25]
```

Algunas muestras contienen datos no numéricos. Como ejemplo, el archivo `biomed.data` (el cual es parte de otro mayor descargado también del 'StatLib Data Repository') contiene cuatro mediciones sanguíneas tomadas a dos grupos de pacientes, `A` y `B`, de diferentes edades,

```
(%i1) s3 : read_matrix (file_search ("biomed.data"))$
(%i2) length (s3);
(%o2)                          100
```

```
(%i3) s3 [1]; /* first record */
(%o3)            [A, 30, 167.0, 89.0, 25.6, 364]
```

El primer individuo pertenece al grupo `A`, tiene 30 años de edad y sus medidas sanguíneas fueron 167.0, 89.0, 25.6 y 364.

Debe tenerse cuidado cuando se trabaje con datos categóricos. En el siguiente ejemplo, se asigna al símbolo `a` cierto valor en algún momento previo y luego se toma una muestra con el valor categórico `a`,

```
(%i1) a : 1$
(%i2) matrix ([a, 3], [b, 5]);
                             [ 1  3 ]
(%o2)                        [      ]
                             [ b  5 ]
```

44.2 Funciones y variables para el tratamiento de datos

`build_sample` (*list*) [Función]
`build_sample` (*matrix*) [Función]

Construye una muestra a partir de una tabla de frecuencias absolutas. La tabla de entrada puede ser una una matriz o una lista de listas, todas ellas de igual tamaño. El número de columnas o la longitud de las listas debe ser mayor que la unidad. El último elemento de cada fila o lista se interpreta como la frecuencia absoluta. El resultado se devuelve siempre en formato de matriz.

Ejemplos:

Tabla de frecuencias univariante.

```
(%i1) load ("descriptive")$
(%i2) sam1: build_sample([[6,1], [j,2], [2,1]]);
                         [ 6 ]
                         [   ]
                         [ j ]
(%o2)                    [   ]
                         [ j ]
                         [   ]
                         [ 2 ]
(%i3) mean(sam1);
                          2 j + 8
(%o3)                    [-------]
                             4
(%i4) barsplot(sam1) $
```

Tabla de frecuencias multivariante.

```
(%i1) load ("descriptive")$
(%i2) sam2: build_sample([[6,3,1], [5,6,2], [u,2,1],[6,8,2]]) ;
                               [ 6  3 ]
                               [      ]
                               [ 5  6 ]
                               [      ]
```

```
                              [ 5  6 ]
(%o2)                         [      ]
                              [ u  2 ]
                              [      ]
                              [ 6  8 ]
                              [      ]
                              [ 6  8 ]
(%i3) cov(sam2);
      [   2                   2                              ]
      [  u  + 158   (u + 28)      2 u + 174   11 (u + 28)    ]
      [  -------- - ---------     --------- - -----------    ]
(%o3) [     6          36             6           12         ]
      [                                                      ]
      [ 2 u + 174   11 (u + 28)            21               ]
      [ --------- - -----------            --               ]
      [     6           12                 4                ]
(%i4) barsplot(sam2, grouping=stacked) $
```

`continuous_freq` (*list*) [Función]
`continuous_freq` (*list*, *m*) [Función]

El argumento de `continuous_freq` debe ser una lista de números. Divide el rango en intervalos y cuenta cuántos valores hay en ellos. El segundo argumento es opcional y puede ser el número de clases deseado, 10 por defecto, o una lista que contenga los límites de las clases y el número de éstas, o una lista que contenga únicamente los límites. Si los valores muestrales son todos iguales, esta función devuelve solamente una clase de amplitud 2.

Ejemplos:

El argumento opcional indica el número de clases deseadas. La primera lista de la respuesta contiene los límites de los intervalos y la segunda los totales correspondientes: hay 16 dígitos en el intervalo `[0, 1.8]`, 24 en `(1.8, 3.6]` y así sucesivamente.

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) continuous_freq (s1, 5);
(%o3) [[0, 1.8, 3.6, 5.4, 7.2, 9.0], [16, 24, 18, 17, 25]]
```

El argumento opcional indica que queremos 7 clases con límites -2 y 12:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) continuous_freq (s1, [-2,12,7]);
(%o3) [[- 2, 0, 2, 4, 6, 8, 10, 12], [8, 20, 22, 17, 20, 13, 0]]
```

El argumento opcional indica que queremos el número por defecto de clases y límites -2 y 12:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) continuous_freq (s1, [-2,12]);
                  3  4  11  18     32  39  46  53
(%o3)  [[- 2, - -, -, --, --, 5, --, --, --, --, 12],
```

```
                        5  5  5   5      5   5   5   5
                       [0, 8, 20, 12, 18, 9, 8, 25, 0, 0]]
```

`discrete_freq` (*list*) [Función]

Calcula las frecuencias absolutas en muestras discretas, tanto numéricas como categóricas. Su único argumento debe ser una lista.

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) discrete_freq (s1);
(%o3) [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
                                [8, 8, 12, 12, 10, 8, 9, 8, 12, 13]]
```

La primera lista son los valores de la muestra y la segunda sus frecuencias absolutas. Las instrucciones `? col` y `? transpose` pueden ayudar a comprender la última entrada.

`standardize` (*list*) [Función]

`standardize` (*matrix*) [Función]

Resta a cada elemento de la lista la media muestral y luego divide el resultado por la desviación típica. Si la entrada es una matriz, `standardize` resta a cada fila la media multivariante y luego divide cada componente por la desviación típica correspondiente.

`subsample` (*data_matrix*, *predicate_function*) [Función]

`subsample` (*data_matrix*, *predicate_function*, *col_num*, *col_num*, ...) [Función]

Esta es una variante de la función `submatrix` de Maxima. El primer argumento es una matriz de datos, el segundo es una función de predicado y el resto de argumentos opcionales son los números de las columnas a tomar en consideración.

Estos son los registros multivariantes en los que la velocidad del viento en la primera estación meteorológica fue menor de 18 nudos. Véase cómo en la expresión lambda la *i*-ésima componente se la referencia como `v[i]`.

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) subsample (s2, lambda([v], v[1] > 18));
              [ 19.38  15.37  15.12  23.09  25.25 ]
              [                                    ]
              [ 18.29  18.66  19.08  26.08  27.63 ]
(%o3)         [                                    ]
              [ 20.25  21.46  19.95  27.71  23.38 ]
              [                                    ]
              [ 18.79  18.96  14.46  26.38  21.84 ]
```

En el siguiente ejemplo, se solicitan únicamente la primera, segunda y quinta componentes de aquellos registros con velocidades del viento mayores o iguales que 16 nudos en la estación número 1 y menores que 25 nudos en la estación número 4. La muestra sólo contiene los datos referidos a las estaciones 1, 2 y 5. En este caso, la función de predicado se define por medio de una función de Maxima ordinaria.

```
(%i1) load ("descriptive")$
```

```
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) g(x):= x[1] >= 16 and x[4] < 25$
(%i4) subsample (s2, g, 1, 2, 5);
                     [ 19.38  15.37  25.25 ]
                     [                     ]
                     [ 17.33  14.67  19.58 ]
(%o4)                [                     ]
                     [ 16.92  13.21  21.21 ]
                     [                     ]
                     [ 17.25  18.46  23.87 ]
```

He aquí un ejemplo con las variables categóricas de `biomed.data`. Se piden los registros correspondientes a aquellos pacientes del grupo `B` mayores de 38 anños,

```
(%i1) load ("descriptive")$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i3) h(u):= u[1] = B and u[2] > 38 $
(%i4) subsample (s3, h);
                [ B  39  28.0  102.3  17.1  146 ]
                [                               ]
                [ B  39  21.0  92.4   10.3  197 ]
                [                               ]
                [ B  39  23.0  111.5  10.0  133 ]
                [                               ]
                [ B  39  26.0  92.6   12.3  196 ]
(%o4)           [                               ]
                [ B  39  25.0  98.7   10.0  174 ]
                [                               ]
                [ B  39  21.0  93.2   5.9   181 ]
                [                               ]
                [ B  39  18.0  95.0   11.3  66  ]
                [                               ]
                [ B  39  39.0  88.5   7.6   168 ]
```

Es probable que el análisis estadístico requiera únicamente de las medidas sanguíneas.

```
(%i1) load ("descriptive")$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i3) subsample (s3, lambda([v], v[1] = B and v[2] > 38),
          3, 4, 5, 6);
                  [ 28.0  102.3  17.1  146 ]
                  [                        ]
                  [ 21.0  92.4   10.3  197 ]
                  [                        ]
                  [ 23.0  111.5  10.0  133 ]
                  [                        ]
                  [ 26.0  92.6   12.3  196 ]
(%o3)             [                        ]
                  [ 25.0  98.7   10.0  174 ]
                  [                        ]
```

```
                        [ 21.0  93.2   5.9   181 ]
                        [                        ]
                        [ 18.0  95.0   11.3  66  ]
                        [                        ]
                        [ 39.0  88.5   7.6   168 ]
```

Esta es la media multivariante de `s3`.

```
(%i1) load ("descriptive")$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i3) mean (s3);
       65 B + 35 A  317          6 NA + 8145.0
(%o3) [-----------, ---, 87.178, -------------, 18.123,
           100      10                100
                                                  3 NA + 19587
                                                  ------------]
                                                      100
```

Aquí la primera componente carece de significado, ya que tanto `A` como `B` son categóricas, la segunda componente es la edad media de los individuos en forma racional, al tiempo que los valores cuarto y quinto muestran cierto comportamiento extraño; lo cual se debe a que el símbolo `NA` se utiliza para indicar datos *no disponibles*, por lo que ambas medias no tienen sentido. Una posible solución puede ser extraer de la matriz aquellas filas con símbolos `NA`, lo que acarrearía cierta pérdida de información.

```
(%i1) load ("descriptive")$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i3) g(v):= v[4] # NA and v[6] # NA $
(%i4) mean (subsample (s3, g, 3, 4, 5, 6));
(%o4) [79.4923076923077, 86.2032967032967, 16.93186813186813,
                                                        2514
                                                        ----]
                                                         13
```

`transform_sample` (*matriz*, *varlist*, *exprlist*) [Función]

Transforma la *matriz* de datos, en la que a cada columna se le asigna un nombre de acuerdo con la lista *varlist*, según las expresiones de *exprlist*.

Ejemplos:

El segundo argumento asigna nombres a las tres columnas, con ellos la lista de expresiones define la transformación de la muestra.

```
(%i1) load ("descriptive")$
(%i2) data: matrix([3,2,7],[3,7,2],[8,2,4],[5,2,4]) $
```

```
(%i3) transform_sample(data, [a,b,c], [c, a*b, log(a)]);
                                [ 7  6   log(3) ]
                                [               ]
                                [ 2  21  log(3) ]
(%o3)                           [               ]
                                [ 4  16  log(8) ]
                                [               ]
                                [ 4  10  log(5) ]
```

Añade una columna constante y elimina la tercera variable.

```
(%i1) load ("descriptive")$
(%i2) data: matrix([3,2,7],[3,7,2],[8,2,4],[5,2,4]) $
(%i3) transform_sample(data, [a,b,c], [makelist(1,k,length(data)),a,b]);
                                  [ 1  3  2 ]
                                  [         ]
                                  [ 1  3  7 ]
(%o3)                             [         ]
                                  [ 1  8  2 ]
                                  [         ]
                                  [ 1  5  2 ]
```

44.3 Funciones y variables de parámetros descriptivos

mean (*list*) [Función]
mean (*matrix*) [Función]

Es la media muestral, definida como

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) mean (s1);
                                 471
(%o3)                            ---
                                 100
(%i4) %, numer;
(%o4)                           4.71
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) mean (s2);
(%o6)     [9.9485, 10.1607, 10.8685, 15.7166, 14.8441]
```

var (*list*) [Función]
var (*matrix*) [Función]

Es la varianza muestral, definida como

$$\frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^2$$

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) var (s1), numer;
(%o3)                  8.425899999999999
```

Véase también `var1`.

`var1` (*list*) [Función]
`var1` (*matrix*) [Función]

Es la cuasivarianza muestral, definida como

$$\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) var1 (s1), numer;
(%o3)                   8.5110101010101
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) var1 (s2);
(%o5) [17.39586540404041, 15.13912778787879, 15.63204924242424,
                            32.50152569696971, 24.66977392929294]
```

Véase también `var`.

`std` (*list*) [Función]
`std` (*matrix*) [Función]

Es la desviación típica muestral, raíz cuadrada de `var`.

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) std (s1), numer;
(%o3)                  2.902740084816414
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) std (s2);
(%o5) [4.149928523480858, 3.871399812729241, 3.933920277534866,
                            5.672434260526957, 4.941970881136392]
```

Véanse también `var` y `std1`.

`std1` (*list*) [Función]
`std1` (*matrix*) [Función]

Es la cuasidesviación típica muestral, raíz cuadrada de `var1`.

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
```

```
(%i3) std1 (s1), numer;
(%o3)                    2.917363553109228
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) std1 (s2);
(%o5) [4.17083509672109, 3.89090320978032, 3.953738641137555,
                              5.701010936401517, 4.966867617451963]
```

Véanse también `var1` y `std`.

`noncentral_moment` (*`list`*, *`k`*) [Función]
`noncentral_moment` (*`matrix`*, *`k`*) [Función]

Es el momento no central de orden k, definido como

$$\frac{1}{n}\sum_{i=1}^{n} x_i^k$$

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) noncentral_moment (s1, 1), numer; /* the mean */
(%o3)                           4.71
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%o5) [319793.8724761506, 320532.1923892463, 391249.5621381556,
                              2502278.205988911, 1691881.797742255]
(%i6) noncentral_moment (s2, 5);
```

Véase también `central_moment`.

`central_moment` (*`list`*, *`k`*) [Función]
`central_moment` (*`matrix`*, *`k`*) [Función]

Es el momento central de orden k, definido como

$$\frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^k$$

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) central_moment (s1, 2), numer; /* the variance */
(%o3)                    8.425899999999999
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%o5) [11.29584771375004, 16.97988248298583, 5.626661952750102,
                               37.5986572057918, 25.85981904394192]
(%i6) central_moment (s2, 3);
```

Véanse también `central_moment` y `mean`.

`cv` (*`list`*) [Función]
`cv` (*`matrix`*) [Función]

Es el coeficiente de variación, o cociente entre la desviación típica muestral (`std`) y la media (`mean`),

```
(%i1) load ("descriptive")$
```

```
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) cv (s1), numer;
(%o3)                    .6193977819764815
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) cv (s2);
(%o5) [.4192426091090204, .3829365309260502, 0.363779605385983,
                              .3627381836021478, .3346021393989506]
```

Véanse también `std` y `mean`.

`smin` (*list*) [Función]
`smin` (*matrix*) [Función]

Es el valor mínimo de la muestra *list*. Cuando el argumento es una matriz, `smin` devuelve una lista con los valores mínimos de las columnas, las cuales están asociadas a variables estadísticas.

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) smin (s1);
(%o3)                            0
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) smin (s2);
(%o5)             [0.58, 0.5, 2.67, 5.25, 5.17]
```

Véase también `smax`.

`smax` (*list*) [Función]
`smax` (*matrix*) [Función]

Es el valor máximo de la muestra *list*. Cuando el argumento es una matriz, `smax` devuelve una lista con los valores máximos de las columnas, las cuales están asociadas a variables estadísticas.

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) smax (s1);
(%o3)                            9
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) smax (s2);
(%o5)          [20.25, 21.46, 20.04, 29.63, 27.63]
```

Véase también `smin`.

`range` (*list*) [Función]
`range` (*matrix*) [Función]

Es la diferencia entre los valores extremos.

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) range (s1);
(%o3)                            9
(%i4) s2 : read_matrix (file_search ("wind.data"))$
```

```
(%i5) range (s2);
(%o5)          [19.67, 20.96, 17.37, 24.38, 22.46]
```

quantile (*list*, *p*) [Función]
quantile (*matrix*, *p*) [Función]

Es el *p*-cuantil, siendo *p* un número del intervalo $[0, 1]$, de la muestra *list*. Aunque existen varias definiciones para el cuantil muestral (Hyndman, R. J., Fan, Y. (1996) *Sample quantiles in statistical packages*. American Statistician, 50, 361-365), la programada en el paquete `descriptive` es la basada en la interpolación lineal.

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) /* 1st and 3rd quartiles */
         [quantile (s1, 1/4), quantile (s1, 3/4)], numer;
(%o3)                      [2.0, 7.25]
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) quantile (s2, 1/4);
(%o5)    [7.2575, 7.477500000000001, 7.82, 11.28, 11.48]
```

median (*list*) [Función]
median (*matrix*) [Función]

Una vez ordenada una muestra, si el tamaño muestral es impar la mediana es el valor central, en caso contrario será la media de los dos valores centrales.

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) median (s1);
                                 9
(%o3)                            -
                                 2
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) median (s2);
(%o5)         [10.06, 9.855, 10.73, 15.48, 14.105]
```

La mediana es el cuantil 1/2.

Véase también `quantile`.

qrange (*list*) [Función]
qrange (*matrix*) [Función]

El rango intercuartílico es la diferencia entre el tercer y primer cuartil, `quantile(`*list*`,3/4) - quantile(`*list*`,1/4)`,

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) qrange (s1);
                                21
(%o3)                           --
                                4
```

```
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) qrange (s2);
(%o5) [5.385, 5.572499999999998, 6.0225, 8.729999999999999,
                                                  6.650000000000002]
```

Véase también `quantile`.

`mean_deviation` (*list*) [Función]
`mean_deviation` (*matrix*) [Función]

Es la desviación media, definida como

$$\frac{1}{n}\sum_{i=1}^{n}|x_i - \bar{x}|$$

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) mean_deviation (s1);
                                51
(%o3)                           --
                                20
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) mean_deviation (s2);
(%o5) [3.287959999999999, 3.075342, 3.23907, 4.715664000000001,
                                                  4.028546000000002]
```

Véase también `mean`.

`median_deviation` (*list*) [Función]
`median_deviation` (*matrix*) [Función]

Es la desviación mediana, definida como

$$\frac{1}{n}\sum_{i=1}^{n}|x_i - med|$$

siendo `med` la mediana de *list*.

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) median_deviation (s1);
                                 5
(%o3)                            -
                                 2
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) median_deviation (s2);
(%o5)           [2.75, 2.755, 3.08, 4.315, 3.31]
```

Véase también `mean`.

harmonic_mean (*list*) [Función]
harmonic_mean (*matrix*) [Función]

Es la media armónica, definida como

$$\frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) y : [5, 7, 2, 5, 9, 5, 6, 4, 9, 2, 4, 2, 5]$
(%i3) harmonic_mean (y), numer;
(%o3)                   3.901858027632205
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) harmonic_mean (s2);
(%o5) [6.948015590052786, 7.391967752360356, 9.055658197151745,
                            13.44199028193692, 13.01439145898509]
```

Véanse también mean y geometric_mean.

geometric_mean (*list*) [Función]
geometric_mean (*matrix*) [Función]

Es la media geométrica, definida como

$$\left(\prod_{i=1}^{n} x_i \right)^{\frac{1}{n}}$$

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) y : [5, 7, 2, 5, 9, 5, 6, 4, 9, 2, 4, 2, 5]$
(%i3) geometric_mean (y), numer;
(%o3)                   4.454845412337012
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) geometric_mean (s2);
(%o5) [8.82476274347979, 9.22652604739361, 10.0442675714889,
                            14.61274126349021, 13.96184163444275]
```

Véanse también mean y harmonic_mean.

kurtosis (*list*) [Función]
kurtosis (*matrix*) [Función]

Es el coeficiente de curtosis, definido como

$$\frac{1}{ns^4} \sum_{i=1}^{n} (x_i - \bar{x})^4 - 3$$

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
```

```
(%i3) kurtosis (s1), numer;
(%o3)                   - 1.273247946514421
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) kurtosis (s2);
(%o5) [- .2715445622195385, 0.119998784429451,
   - .4275233490482866, - .6405361979019522, - .4952382132352935]
```

Véanse también `mean`, `var` y `skewness`.

`skewness` (*list*) [Función]
`skewness` (*matrix*) [Función]

Es el coeficiente de asimetría, definido como

$$\frac{1}{ns^3}\sum_{i=1}^{n}(x_i - \bar{x})^3$$

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) skewness (s1), numer;
(%o3)                  .009196180476450306
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) skewness (s2);
(%o5) [.1580509020000979, .2926379232061854, .09242174416107717,
                             .2059984348148687, .2142520248890832]
```

Véanse también `mean`, `var` y `kurtosis`.

`pearson_skewness` (*list*) [Función]
`pearson_skewness` (*matrix*) [Función]

Es el coeficiente de asimetría de Pearson, definido como

$$\frac{3\ (\bar{x} - med)}{s}$$

siendo *med* la mediana de *list*.

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) pearson_skewness (s1), numer;
(%o3)                   .2159484029093895
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) pearson_skewness (s2);
(%o5) [- .08019976629211892, .2357036272952649,
         .1050904062491204, .1245042340592368, .4464181795804519]
```

Véanse también `mean`, `var` y `median`.

quartile_skewness (*list*) [Función]
quartile_skewness (*matrix*) [Función]

Es el coeficiente de asimetría cuartílico, definido como

$$\frac{c_{\frac{3}{4}} - 2\,c_{\frac{1}{2}} + c_{\frac{1}{4}}}{c_{\frac{3}{4}} - c_{\frac{1}{4}}}$$

siendo c_p el p-cuantil de la muestra *list*.

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) quartile_skewness (s1), numer;
(%o3)                  .04761904761904762
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) quartile_skewness (s2);
(%o5) [- 0.0408542246982353, .1467025572005382,
       0.0336239103362392, .03780068728522298, 0.210526315789474]
```

Véase también `quantile`.

km (*list*, *option* ...) [Función]
km (*matrix*, *option* ...) [Función]

Estimador Kaplan-Meier de la función de supervivencia o fiabilidad $S(x) = 1 - F(x)$.

Los datos se pueden introducir como una lista de pares de números o como una matriz de dos columnas. La primera componente es el tiempo observado y la segunda componente es el índice de censura (1 = no censurado, 0 = censurado por la derecha).

El argumento opcional es el nombre de la variable en la expresión devuelta, la cual es x por defecto.

Ejemplos:

Muestra como una lista de pares.

```
(%i1) load ("descriptive")$
(%i2) S: km([[2,1], [3,1], [5,0], [8,1]]);
                          charfun((3 <= x) and (x < 8))
(%o2) charfun(x < 0) + ------------------------------
                                     2
                    3 charfun((2 <= x) and (x < 3))
                  + -------------------------------
                                    4
                  + charfun((0 <= x) and (x < 2))
(%i3) load ("draw")$
(%i4) draw2d(
        line_width = 3, grid = true,
        explicit(S, x, -0.1, 10))$
```

Estimación de probabilidades de supervivencia.

```
(%i1) load ("descriptive")$
(%i2) S(t):= ''(km([[2,1], [3,1], [5,0], [8,1]], t)) $
```

```
(%i3) S(6);
                                1
(%o3)                           -
                                2
```

cdf_empirical (*list*, *option* ...) [Función]
cdf_empirical (*matrix*, *option* ...) [Función]

Función de distribución empírica $F(x)$.

Los datos se pueden introducir como una lista de números o como una matriz columna.

El argumento opcional es el nombre de la variable en la expresión devuelta, la cual es x por defecto.

Ejemplo:

Función de distribución empírica.

```
(%i1) load ("descriptive")$
(%i2) F(x):= ''(cdf_empirical([1,3,3,5,7,7,7,8,9]));
(%o2) F(x) := (charfun(x >= 9) + charfun(x >= 8)
               + 3 charfun(x >= 7) + charfun(x >= 5)
               + 2 charfun(x >= 3) + charfun(x >= 1))/9
(%i3) F(6);
                              4
(%o3)                         -
                              9
(%i4) load("draw")$
(%i5) draw2d(
        line_width = 3,
        grid       = true,
        explicit(F(z), z, -2, 12)) $
```

cov (*matrix*) [Función]

Es la matriz de covarianzas de una muestra multivariante, definida como

$$S = \frac{1}{n}\sum_{j=1}^{n} \left(X_j - \bar{X}\right)\, \left(X_j - \bar{X}\right)'$$

siendo X_j la j-ésima fila de la matriz muestral.

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) fpprintprec : 7$  /* change precision for pretty output */
      [ 17.22191  13.61811  14.37217  19.39624  15.42162 ]
      [                                                  ]
      [ 13.61811  14.98774  13.30448  15.15834  14.9711  ]
      [                                                  ]
(%o4) [ 14.37217  13.30448  15.47573  17.32544  16.18171 ]
      [                                                  ]
      [ 19.39624  15.15834  17.32544  32.17651  20.44685 ]
```

```
      [                                                          ]
      [ 15.42162  14.9711   16.18171  20.44685  24.42308 ]
(%i5) cov (s2);
```

Véase también `cov1`.

`cov1` (*matrix*) [Función]

Es la matriz de cuasivarianzas de una muestra multivariante, definida como

$$\frac{1}{n-1}\sum_{j=1}^{n}\left(X_j-\bar{X}\right)\,\left(X_j-\bar{X}\right)'$$

siendo X_j la j-ésima fila de la matriz muestral.

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) fpprintprec : 7$ /* change precision for pretty output */
      [ 17.39587  13.75567  14.51734  19.59216  15.5774  ]
      [                                                   ]
      [ 13.75567  15.13913  13.43887  15.31145  15.12232 ]
      [                                                   ]
(%o4) [ 14.51734  13.43887  15.63205  17.50044  16.34516 ]
      [                                                   ]
      [ 19.59216  15.31145  17.50044  32.50153  20.65338 ]
      [                                                   ]
      [ 15.5774   15.12232  16.34516  20.65338  24.66977 ]
(%i5) cov1 (s2);
```

Véase también `cov`.

`global_variances` (*matrix*) [Función]

`global_variances` (*matrix*, *options* ...) [Función]

La función `global_variances` devuelve una lista de medidas globales de variabilidad:

- *varianza total*: `trace(S_1)`,
- *varianza media*: `trace(S_1)/p`,
- *varianza generalizada*: `determinant(S_1)`,
- *desviación típica generalizada*: `sqrt(determinant(S_1))`,
- *varianza efectiva* `determinant(S_1)^(1/p)`, (definida en: Peña, D. (2002) *Análisis de datos multivariantes*; McGraw-Hill, Madrid.)
- *desviación típica efectiva*: `determinant(S_1)^(1/(2*p))`.

donde p es la dimensión de la variable aleatoria multivariante y S_1 la matriz de covarianzas devuelta por la función `cov1`.

Opción:

- `'data`, por defecto `'true`, indica si la matriz de entrada contiene los datos muestrales, en cuyo caso la matriz de covarianzas `cov1` debe ser calculada; en caso contrario, se le debe pasar ésta a la función como matriz simétrica en lugar de los datos.

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) global_variances (s2);
(%o3) [105.338342060606, 21.06766841212119, 12874.34690469686,
         113.4651792608502, 6.636590811800794, 2.576158149609762]
```

Cálculo de `global_variances` a partir de la matriz de covarianzas.

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) s : cov1 (s2)$
(%i4) global_variances (s, data=false);
(%o4) [105.338342060606, 21.06766841212119, 12874.34690469686,
         113.4651792608502, 6.636590811800794, 2.576158149609762]
```

Véanse también `cov` y `cov1`.

`cor` (*matrix*) [Función]
`cor` (*matrix*, *options* ...) [Función]

Es la matriz de correlaciones de la muestra multivariante.

Opción:

- `'data`, por defecto `'true`, indica si la matriz de entrada contiene los datos muestrales, en cuyo caso la matriz de covarianzas `cov1` debe ser calculada; en caso contrario, se le debe pasar ésta a la función como matriz simétrica en lugar de los datos.

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) fpprintprec:7$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) cor (s2);
      [   1.0     .8476339  .8803515  .8239624  .7519506 ]
      [                                                   ]
      [ .8476339    1.0     .8735834  .6902622  0.782502 ]
      [                                                   ]
(%o4) [ .8803515  .8735834    1.0     .7764065  .8323358 ]
      [                                                   ]
      [ .8239624  .6902622  .7764065    1.0     .7293848 ]
      [                                                   ]
      [ .7519506  0.782502  .8323358  .7293848    1.0    ]
```

Cálculo de la matriz de correlaciones a partir de la matriz de covarianzas.

```
(%i1) load ("descriptive")$
(%i2) fpprintprec : 7 $
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) s : cov1 (s2)$
```

```
(%i5) cor (s, data=false); /* this is faster */
      [   1.0     .8476339  .8803515  .8239624  .7519506 ]
      [                                                   ]
      [ .8476339    1.0     .8735834  .6902622  0.782502 ]
      [                                                   ]
(%o5) [ .8803515  .8735834    1.0     .7764065  .8323358 ]
      [                                                   ]
      [ .8239624  .6902622  .7764065    1.0     .7293848 ]
      [                                                   ]
      [ .7519506  0.782502  .8323358  .7293848    1.0    ]
```

Véanse también `cov` y `cov1`.

`list_correlations` (*matrix*) [Función]
`list_correlations` (*matrix*, *options* ...) [Función]

La función `list_correlations` devuelve una lista con medidas de correlación:

- *matriz de precisión*: es la inversa de la matriz de covarianzas S_1,

$$S_1^{-1} = \left(s^{ij}\right)_{i,j=1,2,\ldots,p}$$

- *multiple correlation vector*: $(R_1^2, R_2^2, ..., R_p^2)$, with

$$R_i^2 = 1 - \frac{1}{s^{ii}s_{ii}}$$

  es un indicador de la bondad de ajuste del modelo de regresión lineal multivariante de X_i cuando el resto de variables se utilizan como regresores.

- *matriz de correlaciones parciales*: en la que el elemento (i, j) es

$$r_{ij.rest} = -\frac{s^{ij}}{\sqrt{s^{ii}s^{jj}}}$$

Opción:

- `'data`, por defecto `'true`, indica si la matriz de entrada contiene los datos muestrales, en cuyo caso la matriz de covarianzas `cov1` debe ser calculada; en caso contrario, se le debe pasar ésta a la función como matriz simétrica en lugar de los datos.

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) z : list_correlations (s2)$
```

```
(%i4) fpprintprec : 5$ /* for pretty output */
      [  .38486   - .13856   - .15626   - .10239    .031179  ]
      [                                                       ]
      [ - .13856   .34107    - .15233    .038447   - .052842 ]
      [                                                       ]
(%o5) [ - .15626  - .15233    .47296    - .024816  - .10054  ]
      [                                                       ]
      [ - .10239   .038447   - .024816   .10937    - .034033 ]
      [                                                       ]
      [ .031179   - .052842  - .10054   - .034033   .14834   ]
(%o6)       [.85063, .80634, .86474, .71867, .72675]
       [  - 1.0     .38244   .36627   .49908   - .13049 ]
       [                                               ]
       [  .38244    - 1.0    .37927  - .19907   .23492  ]
       [                                               ]
(%o7)  [  .36627    .37927   - 1.0    .10911    .37956  ]
       [                                               ]
       [  .49908   - .19907  .10911   - 1.0     .26719  ]
       [                                               ]
       [ - .13049   .23492   .37956   .26719    - 1.0   ]
```

Véanse también `cov` y `cov1`.

`principal_components` (*matrix*) [Función]
`principal_components` (*matrix*, *options* ...) [Función]

Calcula las componentes principales de una muestra multivariante. Las componentes principales se utilizan en el análisis estadístico multivariante para reducir la dimensionalidad de la muestra.

Opción:

- `'data`, por defecto `'true`, indica si la matriz de entrada contiene los datos muestrales, en cuyo caso la matriz de covarianzas `cov1` debe ser calculada; en caso contrario, se le debe pasar ésta a la función como matriz simétrica en lugar de los datos.

La salida de la función `principal_components` es una lista con los siguientes resultados:

- varianzas de las componentes principales,
- porcentajes de variación total explicada por cada componente principal,
- matriz de rotación.

Ejemplos:

En este ejemplo, la primera componente explica el 83.13 por ciento de la varianza total.

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) fpprintprec:4 $
(%i4) res: principal_components(s2);
```

```
0 errors, 0 warnings
(%o4) [[87.57, 8.753, 5.515, 1.889, 1.613],
[83.13, 8.31, 5.235, 1.793, 1.531],
[ .4149  .03379   - .4757  - 0.581   - .5126 ]
[                                            ]
[ 0.369  - .3657  - .4298   .7237    - .1469 ]
[                                            ]
[ .3959  - .2178  - .2181  - .2749    .8201  ]]
[                                            ]
[ .5548   .7744    .1857    .2319    .06498  ]
[                                            ]
[ .4765  - .4669   0.712   - .09605  - .1969 ]
(%i5) /* porcentajes acumulados  */
    block([ap: copy(res[2])],
      for k:2 thru length(ap) do ap[k]: ap[k]+ap[k-1],
      ap);
(%o5)                 [83.13, 91.44, 96.68, 98.47, 100.0]
(%i6) /* dimension de la muestra */
      p: length(first(res));
(%o6)                                  5
(%i7) /* dibuja porcentajes para seleccionar el numero de
         componentes principales para el analisis ulterior */
     draw2d(
        fill_density = 0.2,
        apply(bars, makelist([k, res[2][k], 1/2], k, p)),
        points_joined = true,
        point_type    = filled_circle,
        point_size    = 3,
        points(makelist([k, res[2][k]], k, p)),
        xlabel = "Variances",
        ylabel = "Percentages",
        xtics  = setify(makelist([concat("PC",k),k], k, p))) $
```

En caso de que la matriz de covarianzas sea conocida, se le puede pasar a la función, pero debe utilizarse la opción `data=false`.

```
(%i1) load ("descriptive")$
(%i2) S: matrix([1,-2,0],[-2,5,0],[0,0,2]);
                                [  1   - 2  0 ]
                                [             ]
(%o2)                           [ - 2   5   0 ]
                                [             ]
                                [  0    0   2 ]
(%i3) fpprintprec:4 $
(%i4) /* el argumento es una matriz de covarianzas */
      res: principal_components(S, data=false);
0 errors, 0 warnings
                                                [ - .3827  0.0  .9239 ]
```

```
                                                        [                     ]
(%o4) [[5.828, 2.0, .1716], [72.86, 25.0, 2.145], [  .9239   0.0   .3827 ]]
                                                        [                     ]
                                                        [   0.0    1.0   0.0  ]
(%i5) /* transformacion para obtener las componentes principales a
         partir de los registros originales */
      matrix([a1,b2,c3],[a2,b2,c2]).last(res);
             [ .9239 b2 - .3827 a1  1.0 c3  .3827 b2 + .9239 a1 ]
(%o5)        [                                                  ]
             [ .9239 b2 - .3827 a2  1.0 c2  .3827 b2 + .9239 a2 ]
```

44.4 Funciones y variables para gráficos estadísticos

`barsplot` (*data1*, *data2*, ..., *option_1*, *option_2*, ...) [Función]
`barsplot_description` (...) [Función]

Dibuja diagramas de barras para variables estadísticas discretas, tanto para una como para más muestras.

data puede ser una lista de resultados provenientes de una muestra o una matriz de m filas y n columnas, representando n muestras de tamaño m cada una.

Las opciones disponibles son:

box_width (valor por defecto, `3/4`): ancho relativo de los rectángulos. Este valor debe pertenecer al rango `[0,1]`.

- *grouping* (valor por defecto, `clustered`): indica cómo se agrupan las diferentes muestras. Son valores válidos: `clustered` y `stacked`.
- *groups_gap* (valor por defecto, `1`): un número positivo que representa la separación entre dos grupos consecutivos de barras.
- *bars_colors* (valor por defecto, `[]`): una lista de colores para múltiples muestras. Cuando el número de muestras sea mayor que el de colores especificados, los colores adicionales necesarios se seleccionan aleatoriamente. Véase `color` para más información.
- *frequency* (valor por defecto, `absolute`): indica la escala de las ordenadas. Valores admitidos son: `absolute`, `relative` y and `percent`.
- *ordering* (valor por defecto, `orderlessp`): los valores admitidos para esta opción son: `orderlessp` y `ordergreatp`, indicando cómo se deben ordenar los resultados muestrales sobre el eje-x.
- *sample_keys* (valor por defecto, `[]`): es una lista de cadenas de texto a usar como leyendas. Cuando la lista tenga una longitud diferente de cero o del número de muestras, se devolverá un mensaje de error.
- *start_at* (valor por defecto, `0`): indica a qué altura comienza a dibujarse el gráfico de barra sobre el eje de abscisas.
- Todas las opciones globales de `draw`, excepto `xtics`, que se asigna internamente por `barsplot`. Si es necesario que el usuario le dé su propio valor a esta opción, o quiere construir una escena más compleja, debe hacer uso de `barsplot_description`. Véase el ejemplo más abajo.

- Las siguientes opciones locales de `draw`: `key`, `color`, `fill_color`, `fill_density` y `line_width`. Véase también `bars`.

La función `barsplot_description` crea un objeto gráfico útil para formar escenas complejas, junto con otros objetos gráficos. Se dispone también de la función `wxbarsplot` para crear histogramas incorporados en los interfaces wxMaxima y iMaxima.

Ejemplos:

Muestra univariante en formato matricial. Frecuencias absolutas.

```
(%i1) load ("descriptive")$
(%i2) m : read_matrix (file_search ("biomed.data"))$
(%i3) barsplot(
        col(m,2),
        title        = "Ages",
        xlabel       = "years",
        box_width    = 1/2,
        fill_density = 3/4)$
```

Dos muestras de diferente tamaño, con frecuencias relativas y colores definidos por el usuario.

```
(%i1) load ("descriptive")$
(%i2) l1:makelist(random(10),k,1,50)$
(%i3) l2:makelist(random(10),k,1,100)$
(%i4) barsplot(
        l1,l2,
        box_width    = 1,
        fill_density = 1,
        bars_colors  = [black, grey],
        frequency = relative,
        sample_keys = ["A", "B"])$
```

Cuatro muestras no numéricas de igual tamaño.

```
(%i1) load ("descriptive")$
(%i2) barsplot(
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        title  = "Asking for something to four groups",
        ylabel = "# of individuals",
        groups_gap   = 3,
        fill_density = 0.5,
        ordering     = ordergreatp)$
```

Barras apiladas verticalmente.

```
(%i1) load ("descriptive")$
```

```
(%i2) barsplot(
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        title  = "Asking for something to four groups",
        ylabel = "# of individuals",
        grouping     = stacked,
        fill_density = 0.5,
        ordering     = ordergreatp)$
```

`barsplot` en un contexto multiplot.

```
(%i1) load ("descriptive")$
(%i2) l1:makelist(random(10),k,1,50)$
(%i3) l2:makelist(random(10),k,1,100)$
(%i4) bp1 :
        barsplot_description(
         l1,
         box_width = 1,
         fill_density = 0.5,
         bars_colors = [blue],
         frequency = relative)$
(%i5) bp2 :
        barsplot_description(
         l2,
         box_width = 1,
         fill_density = 0.5,
         bars_colors = [red],
         frequency = relative)$
(%i6) draw(gr2d(bp1), gr2d(bp2))$
```

Para las opciones relacionadas con los diagramas de barras, véase `bars` del paquete `draw`.

Véanse también las funciones `histogram` y `piechart`.

`boxplot` (*data*) [Función]
`boxplot` (*data*, *option_1*, *option_2*, ...) [Función]
`boxplot_description` (...) [Función]

Dibuja diagramas de cajas (box-and-whishker). El argumento *data* puede ser una lista, lo cual no es de gran interés, puesto que estos gráficos se utilizan principalmente para comparar distintas muestras, o una matriz, de manera que sea posible comparar dos o más componentes de una muestra multivariante. También se permite que *data* sea una lista de muestras con posibles tamaños diferentes; de hecho, esta es la única función del paquete `descriptive` que admite esta estructura de datos.

La caja se dibuja desde el primer cuartil hasta el tercero, con un segmento horizontal situado a la altura del segundo cuartil o mediana. Por defecto, los bigotes inferior y superior se sitúan a la altura de los valores mínimo y máximo, respectivamente. La opción *range* se puede utilizar para indicar que los valores

mayores que `quantile(x,3/4)+range*(quantile(x,3/4)-quantile(x,1/4))`, o menores que `quantile(x,1/4)-range*(quantile(x,3/4)-quantile(x,1/4))`, deben considerarse atípicos, en cuyo caso se dibujan como puntos aislados, al tiempo que los bigotes se colocan en los extremos del resto de la muestra.

Opciones disponibles:

- *box_width* (valor por defecto, `3/4`): ancho relativo de las cajas. This value must be in the range `[0,1]`.
- *box_orientation* (valor por defecto, `vertical`): valores posibles: `vertical` y `horizontal`.
- *range* (valor por defecto, `inf`): coeficiente positivo del rango intercuartílico para declarar los límites de los datos atípicos.
- *outliers_size* (valor por defecto, `1`): tamaño de los círculos para los datos atípicos.
- Todas las opciones globales de `draw`, excepto `points_joined`, `point_size`, `point_type`, `xtics`, `ytics`, `xrange` y `yrange`, que se asignan internamente por `boxplot`. Si es necesario que el usuario le dé sus propios valores a estas opciones, o quiere construir una escena más compleja, debe hacer uso de `boxplot_description`.
- Las siguientes opciones locales de `draw`: `key`, `color`, y `line_width`.

La función `boxplot_description` crea un objeto gráfico útil para formar escenas complejas, junto con otros objetos gráficos. Se dispone también de la función `wxboxplot` para crear histogramas incorporados en los interfaces wxMaxima y iMaxima.

Ejemplos:

Diagrama de cajas de una muestra multivariante.

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix(file_search("wind.data"))$
(%i3) boxplot(s2,
        box_width  = 0.2,
        title      = "Windspeed in knots",
        xlabel     = "Stations",
        color      = red,
        line_width = 2)$
```

Diagrama de cajas de tres muestras de tamaños diferentes.

```
(%i1) load ("descriptive")$
(%i2) A :
       [[6, 4, 6, 2, 4, 8, 6, 4, 6, 4, 3, 2],
        [8, 10, 7, 9, 12, 8, 10],
        [16, 13, 17, 12, 11, 18, 13, 18, 14, 12]]$
(%i3) boxplot (A, box_orientation = horizontal)$
```

La opción *range* puede utilizarse para tratar con datos atípicos.

```
(%i1) load ("descriptive")$
(%i2) B: [[7, 15, 5, 8, 6, 5, 7, 3, 1],
          [10, 8, 12, 8, 11, 9, 20],
          [23, 17, 19, 7, 22, 19]] $
```

```
(%i3) boxplot (B, range=1)$
(%i4) boxplot (B, range=1.5, box_orientation = horizontal)$
(%i5) draw2d(
        boxplot_description(
          B,
          range           = 1.5,
          line_width      = 3,
          outliers_size   = 2,
          color           = red,
          background_color = light_gray),
        xtics = {["Low",1],["Medium",2],["High",3]}) $
```

`histogram` (*list*) [Función]
`histogram` (*list*, *option_1*, *option_2*, ...) [Función]
`histogram` (*one_column_matrix*) [Función]
`histogram` (*one_column_matrix*, *option_1*, *option_2*, ...) [Función]
`histogram` (*one_row_matrix*) [Función]
`histogram` (*one_row_matrix*, *option_1*, *option_2*, ...) [Función]
`histogram_description` (...) [Función]

Dibuja un histograma a partir de una muestra continua. Los datos muestrales deben darse en forma de lista de números o como una matriz unidimensional.

Opciones dispponibles:

- *nclasses* (valor por defecto, `10`): número de clases del histograma, o una lista indicando los límites de las clases y su número, o solamente los límites.
- *frequency* (valor por defecto, `absolute`): indica la escala de las ordenadas. Valores admitidos son: `absolute`, `relative`, `percent` y `density`. Con `density`, el histograma adquiere un área total igual a uno.
- *htics* (valor por defecto, `auto`): formato para las marcas sobre el eje de las abscisas. Valores admitidos son: `auto`, `endpoints`, `intervals` o una lista de etiquetas.
- Todas las opciones globales de `draw`, excepto `xrange`, `yrange` y `xtics`, que son asignadas internamente por `histogram`. Si es necesario que el usuario le dé sus propios valores a estas opciones, debe hacer uso de `histogram_description`. Véase el ejemplo más abajo.
- Las siguientes opciones locales de `draw`: `key`, `color`, `fill_color`, `fill_density` y `line_width`. Véase también `bars`.

La función `histogram_description` crea un objeto gráfico útil para formar escenas complejas, junto con otros objetos gráficos. Se dispone también de la función `wxhistogram` para crear histogramas incorporados en los interfaces wxMaxima y iMaxima.

Ejemplos:

Un histograma con seis clases:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) histogram (
```

```
        s1,
        nclasses     = 8,
        title        = "pi digits",
        xlabel       = "digits",
        ylabel       = "Absolute frequency",
        fill_color   = grey,
        fill_density = 0.6)$
```

Ajustando los límites del histograma a -2 y 12, con 3 clases. También se establece un formato predefinido a las marcas del eje de abscisas:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) histogram (
        s1,
        nclasses     = [-2,12,3],
        htics        = ["A", "B", "C"],
        terminal     = png,
        fill_color   = "#23afa0",
        fill_density = 0.6)$
```

Se hace uso de `histogram_description` para ajustar la opción `xrange` y añadir una curva explícita a la escena:

```
(%i1) load ("descriptive")$
(%i2) ( load("distrib"),
       m: 14, s: 2,
       s2: random_normal(m, s, 1000) ) $
(%i3) draw2d(
       grid   = true,
       xrange = [5, 25],
       histogram_description(
         s2,
         nclasses     = 9,
         frequency    = density,
         fill_density = 0.5),
       explicit(pdf_normal(x,m,s), x, m - 3*s, m + 3* s))$
```

`piechart` (*list*) [Función]
`piechart` (*list*, *option_1*, *option_2*, ...) [Función]
`piechart` (*one_column_matrix*) [Función]
`piechart` (*one_column_matrix*, *option_1*, *option_2*, ...) [Función]
`piechart` (*one_row_matrix*) [Función]
`piechart` (*one_row_matrix*, *option_1*, *option_2*, ...) [Función]
`piechart_description` (...) [Función]

Similar a `barsplot`, pero dibuja sectores en lugar de rectángulos.

Opciones disponibles:

- *sector_colors* (valor por defecto, `[]`): una lista de colores para los sectores. Cuando el número de sectores sea mayor que el de colores especificados, los colores

adicionales necesarios se seleccionan aleatoriamente. Véase `color` para más información.

- *pie_center* (valor por defecto, `[0,0]`): centro del diagrama
- *pie_radius* (valor por defecto, `1`): radio del diagrama.
- Todas las opciones globales de `draw`, excepto `key`, que se asigna internamente por `piechart`. Si es necesario que el usuario le dé su propio valor a esta opción, o quiere construir una escena más compleja, debe hacer uso de `piechart_description`.
- Las siguientes opciones locales de `draw`: `key`, `color`, `fill_density` y `line_width`. Véase también `bars`.

La función `piechart_description` crea un objeto gráfico útil para formar escenas complejas, junto con otros objetos gráficos. Se dispone también de la función `wxpiechart` para crear histogramas incorporados en los interfaces wxMaxima y iMaxima.

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) piechart(
        s1,
        xrange  = [-1.1, 1.3],
        yrange  = [-1.1, 1.1],
        title   = "Digit frequencies in pi")$
```

Véase también la función `barsplot`.

`scatterplot` (*list*) [Función]
`scatterplot` (*list*, *option_1*, *option_2*, ...) [Función]
`scatterplot` (*matrix*) [Función]
`scatterplot` (*matrix*, *option_1*, *option_2*, ...) [Función]
`scatterplot_description` (...) [Función]

Dibuja diagramas de dispersión, tanto de muestras univariantes (*list*) como multivariantes (*matrix*).

Las opciones disponibles son las mismas que admite `histogram`.

La función `scatterplot_description` crea un objeto gráfico útil para formar escenas complejas, junto con otros objetos gráficos. Se dispone también de la función `wxscatterplot` para crear histogramas incorporados en los interfaces wxMaxima y iMaxima.

Ejemplos:

Diagrama de dispersión univariante a partir de una muestra normal simulada.

```
(%i1) load ("descriptive")$
(%i2) load ("distrib")$
(%i3) scatterplot(
        random_normal(0,1,200),
        xaxis      = true,
        point_size = 2,
        dimensions = [600,150])$
```

Diagrama de dispersión bidimensional.

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) scatterplot(
       submatrix(s2, 1,2,3),
       title      = "Data from stations #4 and #5",
       point_type = diamant,
       point_size = 2,
       color      = blue)$
```

Diagrama de dispersión tridimensional.

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) scatterplot(submatrix (s2, 1,2), nclasses=4)$
```

Diagrama de dispersión de cinco dimensiones, con histogramas de cinco classes.

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) scatterplot(
        s2,
        nclasses     = 5,
        frequency    = relative,
        fill_color   = blue,
        fill_density = 0.3,
        xtics        = 5)$
```

Para dibujar puntos aislados o unidos por segmentos, tanto en dos como en tres dimensiones, véase `points`. Véase también `histogram`.

`starplot` (*data1*, *data2*, ..., *option_1*, *option_2*, ...) [Función]
`starplot_description` (...) [Función]

Dibuja diagramas de estrellas para variables estadísticas discretas, tanto para una como para más muestras.

data puede ser una lista de resultados provenientes de una muestra o una matriz de m filas y n columnas, representando n muestras de tamaño m cada una.

Las opciones disponibles son:

- *stars_colors* (valor por defecto, `[]`): una lista de colores para muestras múltiples. Cuando haya más muestras que colores especificados, los colores que faltan se eligen aleatoriamente. Véase `color` para más información.
- *frequency* (valor por defecto, `absolute`): indica la escala de los radios. Valores admitidos son: `absolute` y `relative`.
- *ordering* (valor por defecto, `orderlessp`): los valores admitidos para esta opción son: `orderlessp` y `ordergreatp`, indicando cómo se deben ordenar los resultados muestrales sobre el eje-x.
- *sample_keys* (valor por defecto, `[]`): es una lista de cadenas de texto a usar como leyendas. Cuando la lista tenga una longitud diferente de cero o del número de muestras, se devolverá un mensaje de error.

- *star_center* (valor por defecto, `[0,0]`): centro del diagrama.
- *star_radius* (valor por defecto, `1`): radio del diagrama.
- Todas las opciones globales de `draw`, excepto `points_joined`, `point_type`, and `key`, que se asignan internamente por `starplot`. Si es necesario que el usuario les dé sus propios valores a estas opciones, o quiere construir una escena más compleja, debe hacer uso de `starplot_description`.
- La siguiente opción local de `draw`: `line_width`.

La función `starplot_description` crea un objeto gráfico útil para formar escenas complejas, junto con otros objetos gráficos. Se dispone también de la función `wxstarplot` para crear histogramas incorporados en los interfaces wxMaxima y iMaxima.

Ejemplo:

Gráfico basado en frecuencias absolutas. La localización y el radios lo define el usuario.

```
(%i1) load ("descriptive")$
(%i2) l1: makelist(random(10),k,1,50)$
(%i3) l2: makelist(random(10),k,1,200)$
(%i4) starplot(
        l1, l2,
        stars_colors = [blue,red],
        sample_keys = ["1st sample", "2nd sample"],
        star_center = [1,2],
        star_radius = 4,
        proportional_axes = xy,
        line_width = 2 ) $
```

`stemplot` (*m*) [Función]
`stemplot` (*m*, *option*) [Función]

Dibuja diagrama de tallos y hojas.

La única opción disponible es:

- *leaf_unit* (valor por defecto, `1`): indica la unidad de las hojas; debe ser una potencia de 10.

Ejemplo:

```
(%i1) load ("descriptive")$
(%i2) load("distrib")$
```

```
(%i3) stemplot(
        random_normal(15, 6, 100),
        leaf_unit = 0.1);
-5|4
 0|37
 1|7
 3|6
 4|4
 5|4
 6|57
 7|0149
 8|3
 9|1334588
10|07888
11|01144467789
12|12566889
13|24778
14|047
15|223458
16|4
17|11557
18|000247
19|4467799
20|00
21|1
22|2335
23|01457
24|12356
25|455
27|79
key: 6|3 =  6.3
(%o3)                  done
```

45 diag

45.1 Funciones y variables para diag

`diag` (*lm*) [Función]

Genera una matriz cuadrada con las matrices de *lm* en la diagonal, siendo *lm* una lista de matrices o de escalares.

Ejemplo:

```
(%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 ]
```

Antes de hacer uso de esta función ejecútese `load("diag")`.

`JF` (*lambda*,*n*) [Función]

Devuelve la célula de Jordan de orden *n* con valor propio *lambda*.

Ejemplo:

```
(%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 ]
```

Antes de hacer uso de esta función ejecútese `load("diag")`.

`jordan` (*mat*) [Función]

Devuelve la forma de Jordan de la matriz *mat*, pero en formato de lista de Maxima. Para obtener la matriz correspondiente, llámese a la función `dispJordan` utilizando como argumento la salida de `jordan`.

Ejemplo:

```
(%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 ]
```

Antes de hacer uso de esta función ejecútese `load("diag")`. Véanse también `dispJordan` y `minimalPoly`.

`dispJordan` (*l*) [Función]

Devuelve la matriz de Jordan asociada a la codificación dada por la lista *l*, que habitualmente será la salida de la función `jordan`.

Ejemplo:

```
(%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 ]
```

Antes de hacer uso de esta función ejecútese `load("diag")`. Véanse también `jordan` y `minimalPoly`.

`minimalPoly` (*l*) [Función]

Devuelve el polinomio mínimo asociado a la codificación dada por la lista *l*, que habitualmente será la salida de la función `jordan`.

Ejemplo:

```
(%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)
```

Antes de hacer uso de esta función ejecútese `load("diag")`. Véanse también `jordan` y `dispJordan`.

`ModeMatrix` (*A*,*l*) [Función]

Devuelve la matriz M tal que $(Mm1).A.M = J$, donde J es la forma de Jordan de A. La lista *l* es la forma codificada de la forma de Jordan tal como la devuelve la función `jordan`.

Ejemplo:

```
(%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
```

Nótese que `dispJordan(%o3)` es la forma de Jordan de la matriz `a`.

Antes de hacer uso de esta función ejecútese `load("diag")`. Véanse también `jordan` y `dispJordan`.

`mat_function` (*f*,*mat*) [Función]

Devuelve $f(mat)$, siendo f una función analítica y *mat* una matriz. Este cálculo se basa en la fórmula integral de Cauchy, que establece que si `f(x)` es analítica y

```
mat=diag([JF(m1,n1),...,JF(mk,nk)]),
```

entonces

```
f(mat)=ModeMatrix*diag([f(JF(m1,n1)),...,f(JF(mk,nk))])
          * ModeMatrix^^(-1)
```

Nótese que hay otros métodos alternativos para realizar este cálculo.

Se presentan algunos ejemplos.

Ejemplo 1:

```
(%i1) load("diag")$

(%i2) b2:matrix([0,1,0], [0,0,1], [-1,-3,-3])$

(%i3) mat_function(exp,t*b2);
                2   - t
               t  %e          - t     - t
```

```
(%o3) matrix([-------- + t %e    + %e   ,
                2
       - t     - t                           - t
 2    %e      %e        - t           - t   %e
t  (- ----- - ----- + %e   ) + t (2 %e    - -----)
       t       2                             t
              t
                          - t          - t     - t
      - t       - t   %e        2  %e      %e
 + 2 %e   , t (%e    - -----) + t  (----- - -----)
                          t            2       t
              2   - t           - t     - t
    - t      t  %e        2    %e      %e        - t
 + %e   ], [- --------, - t  (- ----- - ----- + %e   ),
                2                t       2
                                        t
       - t     - t      2   - t
   2  %e      %e        t  %e          - t
- t  (----- - -----)], [-------- - t %e   ,
       2       t          2
       - t     - t                           - t
 2    %e      %e        - t           - t   %e
t  (- ----- - ----- + %e   ) - t (2 %e    - -----),
       t       2                             t
              t
      - t     - t                 - t
 2  %e      %e            - t   %e
t  (----- - -----) - t (%e    - -----)])
      2       t                   t
(%i4) ratsimp(%);
                [   2              - t ]
                [ (t  + 2 t + 2) %e    ]
                [ -------------------- ]
                [          2           ]
                [                      ]
                [         2   - t      ]
(%o4)  Col 1 = [        t  %e         ]
                [      - --------      ]
                [           2          ]
                [                      ]
                [     2          - t   ]
                [   (t  - 2 t) %e      ]
                [   ----------------   ]
                [          2           ]
         [      2        - t    ]
         [    (t  + t) %e       ]
         [                      ]
```

```
Col 2 = [     2            - t ]
        [ - (t  - t - 1) %e    ]
        [                      ]
        [     2          - t   ]
        [   (t  - 3 t) %e      ]
        [        2   - t       ]
        [       t  %e          ]
        [       --------       ]
        [          2           ]
        [                      ]
        [      2          - t  ]
Col 3 = [    (t  - 2 t) %e     ]
        [  - ----------------  ]
        [           2          ]
        [                      ]
        [   2              - t ]
        [ (t  - 4 t + 2) %e    ]
        [ -------------------- ]
        [          2           ]
```

Ejemplo 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     ]
```

Ejemplo 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  ]
```

Antes de hacer uso de esta función ejecútese `load("diag")`.

46 distrib

46.1 Introducción a distrib

El paquete `distrib` contiene un conjunto de funciones para la realización de cálculos probabilísticos con modelos univariantes, tanto discretos como continuos.

A continuación un breve recordatorio de las deficiones básicas sobre distribuciones de probabilidad.

Sea $f(x)$ la *función de densidad* de una variable aleatoria X absolutamente continua. La *función de distribución* se define como

$$F\left(x\right) = \int_{-\infty}^{x} f\left(u\right)\ du$$

que es igual a la probabilidad *Pr(X <= x)*.

La *media* es un parámetro de localización y se define como

$$E\left[X\right] = \int_{-\infty}^{\infty} x\, f\left(x\right)\ dx$$

La *varianza* es una medida de dispersión,

$$V\left[X\right] = \int_{-\infty}^{\infty} f\left(x\right)\ \left(x - E\left[X\right]\right)^2\ dx$$

que es un número real positivo. La raíz cuadrada de la varianza es la *desviación típica*, $D[X] = sqrt(V[X])$, siendo otra medida de dispersión.

El *coeficiente de asimetría* es una medida de forma,

$$SK\left[X\right] = \frac{\int_{-\infty}^{\infty} f\left(x\right)\ \left(x - E\left[X\right]\right)^3\ dx}{D\left[X\right]^3}$$

Y el *coeficiente de curtosis* mide el apuntamiento de la densidad,

$$KU\left[X\right] = \frac{\int_{-\infty}^{\infty} f\left(x\right)\ \left(x - E\left[X\right]\right)^4\ dx}{D\left[X\right]^4} - 3$$

Si X es normal, $KU[X] = 0$. De hecho, tanto la asimetría como la curtosis son parámetros de forma para medir la no normalidad de una distribución.

Si la variable aleatoria X es discreta, su función de densidad, o de *probabiliad*, $f(x)$ toma valores positivos dentro de un conjunto numerable de valores x_i, y cero en cualquier otro lugar. En este caso, la función de distribución es

$$F\left(x\right) = \sum_{x_i \leq x} f\left(x_i\right)$$

La media, varianza, desviación típica y los coeficientes de asimetría y curtosis adquieren las formas

$$E\left[X\right] = \sum_{x_i} x_i f\left(x_i\right),$$

$$V\left[X\right] = \sum_{x_i} f\left(x_i\right)\left(x_i - E\left[X\right]\right)^2,$$

$$D\left[X\right] = \sqrt{V\left[X\right]},$$

$$SK\left[X\right] = \frac{\sum_{x_i} f\left(x\right)\ \left(x - E\left[X\right]\right)^3\ dx}{D\left[X\right]^3}$$

y

$$KU\left[X\right] = \frac{\sum_{x_i} f\left(x\right)\ \left(x - E\left[X\right]\right)^4\ dx}{D\left[X\right]^4} - 3,$$

respectivamente.

Por favor, consúltese cualquier manual introductorio de probabilidad y estadística para más información sobre toda esta parafernalia matemática.

Se sigue cierta convención a la hora de nombrar las funciones del paquete `distrib`. Cada nombre tiene dos partes, el primero hace referencia a la función o parámetro que se quiere calcular,

```
Funciones:
   Función de densidad        (pdf_*)
   Función de distribución    (cdf_*)
   Cuantil                    (quantile_*)
   Media                      (mean_*)
   Varianza                   (var_*)
   Desviación típica          (std_*)
   Coeficiente de asimetría   (skewness_*)
   Coeficiente de curtosis    (kurtosis_*)
   Valor aleatorio            (random_*)
```

La segunda parte hace referencia explícita al modelo probabilístico,

```
Distribuciones continuas:
   Normal              (*normal)
   Student             (*student_t)
   Chi^2               (*chi2)
   Chi^2 no central    (*noncentral_chi2)
   F                   (*f)
   Exponencial         (*exp)
   Lognormal           (*lognormal)
   Gamma               (*gamma)
   Beta                (*beta)
   Continua uniforme   (*continuous_uniform)
   Logística           (*logistic)
   Pareto              (*pareto)
   Weibull             (*weibull)
   Rayleigh            (*rayleigh)
   Laplace             (*laplace)
```

```
   Cauchy              (*cauchy)
   Gumbel              (*gumbel)

Distribuciones discretas:
   Binomial             (*binomial)
   Poisson              (*poisson)
   Bernoulli            (*bernoulli)
   Geométrica           (*geometric)
   Uniforme discreta    (*discrete_uniform)
   Hipergeométrica      (*hypergeometric)
   Binomial negativa    (*negative_binomial)
   Finita discreta      (*general_finite_discrete)
```

Por ejemplo, `pdf_student_t(x,n)` es la función de densidad de la distribución de Student con n grados de libertad, `std_pareto(a,b)` es la desviación típica de la distribución de Pareto de parámetros a y b, y `kurtosis_poisson(m)` es el coeficiente de curtosis de la distribución de Poisson de media m.

Para poder hacer uso del paquete `distrib` es necesario cargarlo primero tecleando

```
(%i1) load("distrib")$
```

Para comentarios, errores o sugerencias, por favor contáctese conmigo en *'riotorto AT yahoo DOT com'*.

46.2 Funciones y variables para distribuciones continuas

`pdf_normal` (*x*,*m*,*s*) [Función]

Devuelve el valor correspondiente a x de la función de densidad de la variable aleatoria $Normal(m, s)$, con $s > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_normal` (*x*,*m*,*s*) [Función]

Devuelve el valor correspondiente a x de la función de distribución de la variable aleatoria $Normal(m, s)$, con $s > 0$. Esta función se define en términos de la función de error, `erf`, de Maxima.

```
(%i1) load ("distrib")$
(%i2) cdf_normal(x,m,s);
                                      x - m
                                 erf(---------)
                                     sqrt(2) s    1
(%o2)                            -------------- + -
                                       2          2
```

Véase también `erf`.

`quantile_normal` (*q*,*m*,*s*) [Función]

Devuelve el q-cuantil de una variable aleatoria $Normal(m, s)$, con $s > 0$; en otras palabras, es la inversa de `cdf_normal`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

```
(%i1) load ("distrib")$
```

```
(%i2) quantile_normal(95/100,0,1);
                                          9
(%o2)             sqrt(2) inverse_erf(--)
                                          10
(%i3) float(%);
(%o3)               1.644853626951472
```

`mean_normal` (*m*,*s*) [Función]
Devuelve la media de una variable aleatoria $Normal(m, s)$, con $s > 0$, es decir m. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_normal` (*m*,*s*) [Función]
Devuelve la varianza de una variable aleatoria $Normal(m, s)$, con $s > 0$, es decir *s^2*.

`std_normal` (*m*,*s*) [Función]
Devuelve la desviación típica de una variable aleatoria $Normal(m, s)$, con $s > 0$, es decir s. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_normal` (*m*,*s*) [Función]
Devuelve el coeficiente de asimetría de una variable aleatoria $Normal(m, s)$, con $s > 0$, que es siempre igual a 0. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_normal` (*m*,*s*) [Función]
Devuelve el coeficiente de curtosis de una variable aleatoria $Normal(m, s)$, con $s > 0$, que es siempre igual a 0. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_normal` (*m*,*s*) [Función]
`random_normal` (*m*,*s*,*n*) [Función]
Devuelve un valor aleatorio $Normal(m, s)$, con $s > 0$. Llamando a `random_normal` con un tercer argumento n, se simula una muestra aleatoria de tamaño n.

El algoritmo de simulación es el de Box-Mueller, tal como está descrito en Knuth, D.E. (1981) *Seminumerical Algorithms. The Art of Computer Programming.* Addison-Wesley.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_student_t` (*x*,*n*) [Función]
Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria de Student $t(n)$, con $n > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_student_t` (*x*,*n*) [Función]
Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria de Student $t(n)$, con $n > 0$.

```
(%i1) load ("distrib")$
(%i2) cdf_student_t(1/2, 7/3);
```

```
                                                7  1  28
                    beta_incomplete_regularized(-, -, --)
                                                6  2  31
(%o2)     1 - ------------------------------------
                                    2
(%i3) float(%);
(%o3)                  .6698450596140415
```

`quantile_student_t` (*q*,*n*) [Función]

Devuelve el q-cuantil de una variable aleatoria de Student $t(n)$, con $n > 0$; en otras palabras, se trata de la inversa de `cdf_student_t`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_student_t` (*n*) [Función]

Devuelve la media de una variable aleatoria de Student $t(n)$, con $n > 0$, que vale siempre 0. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_student_t` (*n*) [Función]

Devuelve la varianza de una variable aleatoria de Student $t(n)$, con $n > 2$.

```
(%i1) load ("distrib")$
(%i2) var_student_t(n);
                                  n
(%o2)                           -----
                                n - 2
```

`std_student_t` (*n*) [Función]

Devuelve la desviación típica de una variable aleatoria de Student $t(n)$, con $n > 2$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_student_t` (*n*) [Función]

Devuelve el coeficiente de asimetría de una variable aleatoria de Student $t(n)$, con $n > 3$, que vale siempre 0. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_student_t` (*n*) [Función]

Devuelve el coeficiente de curtosis una variable aleatoria de Student $t(n)$, con $n > 4$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_student_t` (*n*) [Función]
`random_student_t` (*n*,*m*) [Función]

Devuelve un valor aleatorio $t(n)$, con $n > 0$. Llamando a `random_student_t` con un segundo argumento m, se obtiene una muestra aleatoria simulada de tamaño m.

El algoritmo utilizado está basado en el hecho de que si Z es una variable aleatoria normal $N(0, 1)$ y S^2 es una chi cuadrada de n grados de libertad, $Chi^2(n)$, entonces

$$X = \frac{Z}{\sqrt{\frac{S^2}{n}}}$$

es una variable aleatoria de Student de n grados de libertad, $t(n)$.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

pdf_noncentral_student_t (*x*,*n*,*ncp*) [Función]

Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria no central de Student $nc_t(n, ncp)$, con $n > 0$ grados de libertad y parámetro de no centralidad *ncp*. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

En ocasiones es necesario hacer algún trabajo extra para obtener el resultado final.

```
(%i1) load ("distrib")$
(%i2) expand(pdf_noncentral_student_t(3,5,0.1));
                            7/2                             7/2
      0.04296414417400905 5      1.323650307289301e-6 5
(%o2) ---------------------- + ------------------------
         3/2   5/2                        sqrt(%pi)
        2    14    sqrt(%pi)
                                                        7/2
                                    1.94793720435093e-4 5
                                  + ----------------------
                                              %pi
(%i3) float(%);
(%o3)          .02080593159405669
```

cdf_noncentral_student_t (*x*,*n*,*ncp*) [Función]

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria no central de Student $nc_t(n, ncp)$, con $n > 0$ grados de libertad y parámetro de no centralidad *ncp*. Esta función no tiene expresión compacta y se calcula numéricamente. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

```
(%i1) load ("distrib")$
(%i2) cdf_noncentral_student_t(-2,5,-5);
(%o2)          .9952030093319743
```

quantile_noncentral_student_t (*q*,*n*,*ncp*) [Función]

Devuelve el q-cuantil de una variable aleatoria no central de Student $nc_t(n, ncp)$, con $n > 0$ grados de libertad y parámetro de no centralidad *ncp*; en otras palabras, se trata de la inversa de `cdf_noncentral_student_t`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

mean_noncentral_student_t (*n*,*ncp*) [Función]

Devuelve la media de una variable aleatoria no central de Student $nc_t(n, ncp)$, con $n > 1$ grados de libertad y parámetro de no centralidad *ncp*. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

```
(%i1) load ("distrib")$
(%i2) mean_noncentral_student_t(df,k);
                      df - 1
                 gamma(------) sqrt(df) k
                        2
(%o2)            ------------------------
                                 df
                    sqrt(2) gamma(--)
                                 2
```

`var_noncentral_student_t (n,ncp)` [Función]
Devuelve la varianza de una variable aleatoria no central de Student $nc_t(n, ncp)$, con $n > 2$ grados de libertad y parámetro de no centralidad ncp. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_noncentral_student_t (n,ncp)` [Función]
Devuelve la desviación típica de una variable aleatoria no central de Student $nc_t(n, ncp)$, con $n > 2$ grados de libertad y parámetro de no centralidad ncp. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_noncentral_student_t (n,ncp)` [Función]
Devuelve el coeficiente de asimetría de una variable aleatoria no central de Student $nc_t(n, ncp)$, con $n > 3$ grados de libertad y parámetro de no centralidad ncp. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_noncentral_student_t (n,ncp)` [Función]
Devuelve el coeficiente de curtosis de una variable aleatoria no central de Student $nc_t(n, ncp)$, con $n > 4$ grados de libertad y parámetro de no centralidad ncp. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_noncentral_student_t (n,ncp)` [Función]
`random_noncentral_student_t (n,ncp,m)` [Función]
Devuelve un valor aleatorio $nc_t(n, ncp)$, con $n > 0$. Llamando a `random_noncentral_student_t` con un tercer argumento m, se obtiene una muestra aleatoria simulada de tamaño m.

El algoritmo utilizado está basado en el hecho de que si X es una variable aleatoria normal $N(ncp, 1)$ y S^2 es una chi cuadrada de n grados de libertad, $Chi^2(n)$, entonces

$$U = \frac{X}{\sqrt{\frac{S^2}{n}}}$$

es una variable aleatoria no central de Student de n grados de libertad y parámetro de no centralidad ncp, $nc_t(n, ncp)$.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_chi2 (x,n)` [Función]
Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria chi-cuadrado $Chi^2(n)$, con $n > 0$. La variable aleatoria $Chi^2(n)$ equivale a una $Gamma(n/2, 2)$.

```
(%i1) load ("distrib")$
(%i2) pdf_chi2(x,n);
                              n/2 - 1   - x/2
                             x        %e
(%o2)                        ----------------
                               n/2       n
                              2    gamma(-)
                                         2
```

`cdf_chi2 (x,n)` [Función]

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria chi-cuadrado $Chi^2(n)$, con $n > 0$.

```
(%i1) load ("distrib")$
(%i2) cdf_chi2(3,4);
                                                    3
(%o2)      1 - gamma_incomplete_regularized(2, -)
                                                    2
(%i3) float(%);
(%o3)                .4421745996289256
```

`quantile_chi2 (q,n)` [Función]

Devuelve el q-cuantil de una variable aleatoria $Chi^2(n)$, con $n > 0$; en otras palabras, se trata de la inversa de `cdf_chi2`. El argumento q debe ser un número de $[0, 1]$.

Esta función no tiene expresión compacta y se calcula numéricamente.

```
(%i1) load ("distrib")$
(%i2) quantile_chi2(0.99,9);
(%o2)                   21.66599433346194
```

`mean_chi2 (n)` [Función]

Devuelve la media de una variable aleatoria $Chi^2(n)$, con $n > 0$.

La variable aleatoria $Chi^2(n)$ equivale a una $Gamma(n/2, 2)$.

```
(%i1) load ("distrib")$
(%i2) mean_chi2(n);
(%o2)                           n
```

`var_chi2 (n)` [Función]

Devuelve la varianza de una variable aleatoria $Chi^2(n)$, con $n > 0$.

La variable aleatoria $Chi^2(n)$ equivale a una $Gamma(n/2, 2)$.

```
(%i1) load ("distrib")$
(%i2) var_chi2(n);
(%o2)                         2 n
```

`std_chi2 (n)` [Función]

Devuelve la desviación típica de una variable aleatoria $Chi^2(n)$, con $n > 0$.

La variable aleatoria $Chi^2(n)$ equivale a una $Gamma(n/2, 2)$.

```
(%i1) load ("distrib")$
(%i2) std_chi2(n);
(%o2)                    sqrt(2) sqrt(n)
```

`skewness_chi2 (n)` [Función]

Devuelve el coeficiente de asimetría de una variable aleatoria $Chi^2(n)$, con $n > 0$.

La variable aleatoria $Chi^2(n)$ equivale a una $Gamma(n/2, 2)$.

```
(%i1) load ("distrib")$
(%i2) skewness_chi2(n);
```

```
                                        3/2
                                       2
(%o2)                                  -------
                                       sqrt(n)
```

`kurtosis_chi2 (n)` [Función]

Devuelve el coeficiente de curtosis una variable aleatoria $Chi^2(n)$, con $n > 0$.

La variable aleatoria $Chi^2(n)$ equivale a una $Gamma(n/2, 2)$.

```
(%i1) load ("distrib")$
(%i2) kurtosis_chi2(n);
                                    12
(%o2)                               --
                                    n
```

`random_chi2 (n)` [Función]

`random_chi2 (n,m)` [Función]

Devuelve un valor aleatorio $Chi^2(n)$, con $n > 0$. Llamando a `random_chi2` con un segundo argumento m, se simulará una muestra aleatoria de tamaño m.

La simulación está basada en el algoritmo de Ahrens-Cheng. Véase `random_gamma` para más detalles.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_noncentral_chi2 (x,n,ncp)` [Función]

Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria chi-cuadrado no centrada $nc_Chi^2(n, ncp)$, con $n > 0$ y parámetro de no centralidad $ncp >= 0$. Para hacer uso de esta función ejecútese primero `load("distrib")`.

`cdf_noncentral_chi2 (x,n,ncp)` [Función]

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria chi-cuadrado no centrada $nc_Chi^2(n, ncp)$, con $n > 0$ y parámetro de no centralidad $ncp >= 0$.

`quantile_noncentral_chi2 (q,n,ncp)` [Función]

Devuelve el q-cuantil de una variable aleatoria chi-cuadrado no centrada $nc_Chi^2(n, ncp)$, con $n > 0$ y parámetro de no centralidad $ncp >= 0$; en otras palabras, se trata de la inversa de `cdf_noncentral_chi2`. El argumento q debe ser un número de $[0, 1]$.

Esta función no tiene expresión compacta y se calcula numéricamente.

`mean_noncentral_chi2 (n,ncp)` [Función]

Devuelve la media de una variable aleatoria chi-cuadrado no centrada $nc_Chi^2(n, ncp)$, con $n > 0$ y parámetro de no centralidad $ncp >= 0$.

`var_noncentral_chi2 (n,ncp)` [Función]

Devuelve la varianza de una variable aleatoria chi-cuadrado no centrada $nc_Chi^2(n, ncp)$, con $n > 0$ y parámetro de no centralidad $ncp >= 0$.

`std_noncentral_chi2 (n,ncp)` [Función]

Devuelve la desviación típica de una variable aleatoria chi-cuadrado no centrada $nc_Chi^2(n, ncp)$, con $n > 0$ y parámetro de no centralidad $ncp >= 0$.

`skewness_noncentral_chi2 (n,ncp)` [Función]

Devuelve el coeficiente de asimetría de una variable aleatoria chi-cuadrado no centrada $nc_C hi^2(n, ncp)$, con $n > 0$ y parámetro de no centralidad $ncp >= 0$.

`kurtosis_noncentral_chi2 (n,ncp)` [Función]

Devuelve el coeficiente de curtosis una variable aleatoria chi-cuadrado no centrada $nc_C hi^2(n, ncp)$, con $n > 0$ y parámetro de no centralidad $ncp >= 0$.

`random_noncentral_chi2 (n,ncp)` [Función]
`random_noncentral_chi2 (n,ncp,m)` [Función]

Devuelve un valor aleatorio $nc_C hi^2(n, ncp)$, con $n > 0$ y parámetro de no centralidad $ncp >= 0$. Llamando a `random_noncentral_chi2` con un tercer argumento m, se simulará una muestra aleatoria de tamaño m.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_f (x,m,n)` [Función]

Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria $F(m, n)$, con $m, n > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_f (x,m,n)` [Función]

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria $F(m, n)$, con $m, n > 0$.

```
(%i1) load ("distrib")$
(%i2) cdf_f(2,3,9/4);
                                          9  3  3
(%o2)    1 - beta_incomplete_regularized(-, -, --)
                                          8  2  11
(%i3) float(%);
(%o3)                 0.66756728179008
```

`quantile_f (q,m,n)` [Función]

Devuelve el q-cuantil de una variable aleatoria $F(m, n)$, con $m, n > 0$; en otras palabras, se trata de la inversa de `cdf_f`. El argumento q debe ser un número de $[0, 1]$.

```
(%i1) load ("distrib")$
(%i2) quantile_f(2/5,sqrt(3),5);
                                 2
(%o2)              quantile_f(-, sqrt(3), 5)
                                 5
(%i3) %,numer;
(%o3)                  0.518947838573693
```

`mean_f (m,n)` [Función]

Devuelve la media de una variable aleatoria $F(m, n)$, con $m > 0, n > 2$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_f (m,n)` [Función]

Devuelve la varianza de una variable aleatoria $F(m, n)$, con $m > 0, n > 4$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_f (m,n)` [Función]

Devuelve la desviación típica de una variable aleatoria $F(m,n)$, con $m > 0, n > 4$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_f (m,n)` [Función]

Devuelve el coeficiente de asimetría de una variable aleatoria $F(m,n)$, con $m > 0, n > 6$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_f (m,n)` [Función]

Devuelve el coeficiente de curtosis una variable aleatoria $F(m,n)$, con $m > 0, n > 8$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_f (m,n)` [Función]
`random_f (m,n,k)` [Función]

Devuelve un valor aleatorio $F(m,n)$, con $m, n > 0$. Llamando a `random_f` con un tercer argumento k, se simulará una muestra aleatoria de tamaño k.

El algoritmo de simulación está basado en el hecho de que si X es una variable aleatoria $Chi^2(m)$ y Y es una $Chi^2(n)$, entonces

$$F = \frac{nX}{mY}$$

es una variable aleatoria F con m y n grados de libertad, $F(m,n)$.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_exp (x,m)` [Función]

Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria $Exponencial(m)$, con $m > 0$.

La variable aleatoria $Exponencial(m)$ equivale a una $Weibull(1, 1/m)$.

```
(%i1) load ("distrib")$
(%i2) pdf_exp(x,m);
                                    - m x
(%o2)                          m %e
```

`cdf_exp (x,m)` [Función]

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria $Exponencial(m)$, con $m > 0$.

La variable aleatoria $Exponencial(m)$ equivale a una $Weibull(1, 1/m)$.

```
(%i1) load ("distrib")$
(%i2) cdf_exp(x,m);
                                     - m x
(%o2)                         1 - %e
```

`quantile_exp (q,m)` [Función]

Devuelve el q-cuantil de una variable aleatoria $Exponencial(m)$, con $m > 0$; en otras palabras, se trata de la inversa de `cdf_exp`. El argumento q debe ser un número de $[0, 1]$.

La variable aleatoria *Exponencial*(m) equivale a una *Weibull*$(1, 1/m)$.

```
(%i1) load ("distrib")$
(%i2) quantile_exp(0.56,5);
(%o2)                    .1641961104139661
(%i3) quantile_exp(0.56,m);
                                                1
(%o3)             quantile_weibull(0.56, 1, -)
                                                m
```

`mean_exp` (*m*) [Función]

Devuelve la media de una variable aleatoria *Exponencial*(m), con $m > 0$.

La variable aleatoria *Exponencial*(m) equivale a una *Weibull*$(1, 1/m)$.

```
(%i1) load ("distrib")$
(%i2) mean_exp(m);
                                  1
(%o2)                             -
                                  m
```

`var_exp` (*m*) [Función]

Devuelve la varianza de una variable aleatoria *Exponencial*(m), con $m > 0$.

La variable aleatoria *Exponencial*(m) equivale a una *Weibull*$(1, 1/m)$.

```
(%i1) load ("distrib")$
(%i2) var_exp(m);
                                 1
(%o2)                            --
                                  2
                                 m
```

`std_exp` (*m*) [Función]

Devuelve la desviación típica de una variable aleatoria *Exponencial*(m), con $m > 0$.

La variable aleatoria *Exponencial*(m) equivale a una *Weibull*$(1, 1/m)$.

```
(%i1) load ("distrib")$
(%i2) std_exp(m);
                                  1
(%o2)                             -
                                  m
```

`skewness_exp` (*m*) [Función]

Devuelve el coeficiente de asimetría de una variable aleatoria *Exponencial*(m), con $m > 0$.

La variable aleatoria *Exponencial*(m) equivale a una *Weibull*$(1, 1/m)$.

```
(%i1) load ("distrib")$
(%i2) skewness_exp(m);
(%o2)                             2
```

`kurtosis_exp (m)` [Función]

Devuelve el coeficiente de curtosis una variable aleatoria *Exponencial*(m), con $m > 0$.

La variable aleatoria *Exponencial*(m) equivale a una *Weibull*$(1, 1/m)$.

```
(%i1) load ("distrib")$
(%i2) kurtosis_exp(m);
(%o3)                          6
```

`random_exp (m)` [Función]

`random_exp (m,k)` [Función]

Devuelve un valor aleatorio *Exponencial*(m), con $m > 0$. Llamando a `random_exp2` con un segundo argumento k, se simulará una muestra aleatoria de tamaño k.

El algoritmo de simulación está basado en el método inverso.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_lognormal (x,m,s)` [Función]

Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria *Lognormal*(m, s), con $s > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_lognormal (x,m,s)` [Función]

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria *Lognormal*(m, s), con $s > 0$. Esta función se define en términos de la función de error, `erf`, de Maxima.

```
(%i1) load ("distrib")$
(%i2) assume(x>0, s>0)$  cdf_lognormal(x,m,s);
                             log(x) - m
                         erf(----------)
                             sqrt(2) s     1
(%o2)                    --------------- + -
                               2           2
```

Véase también `erf`.

`quantile_lognormal (q,m,s)` [Función]

Devuelve el q-cuantil de una variable aleatoria *Lognormal*(m, s), con $s > 0$; en otras palabras, se trata de la inversa de `cdf_lognormal`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_lognormal (m,s)` [Función]

Devuelve la media de una variable aleatoria *Lognormal*(m, s), con $s > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_lognormal (m,s)` [Función]

Devuelve la varianza de una variable aleatoria *Lognormal*(m, s), con $s > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_lognormal (m,s)` [Función]

Devuelve la desviación típica de una variable aleatoria *Lognormal*(m, s), con $s > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_lognormal (m,s)` [Función]
Devuelve el coeficiente de asimetría de una variable aleatoria $Lognormal(m, s)$, con $s > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_lognormal (m,s)` [Función]
Devuelve el coeficiente de curtosis una variable aleatoria $Lognormal(m, s)$, con $s > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_lognormal (m,s)` [Función]
`random_lognormal (m,s,n)` [Función]
Devuelve un valor aleatorio $Lognormal(m, s)$, con $s > 0$. Llamando a `random_lognormal` con un tercer argumento n, se simulará una muestra aleatoria de tamaño n.

Las variables lognormales se simulan mediante variables normales. Véase `random_normal` para más detalles.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_gamma (x,a,b)` [Función]
Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria $Gamma(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_gamma (x,a,b)` [Función]
Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria $Gamma(a, b)$, con $a, b > 0$.

```
(%i1) load ("distrib")$
(%i2) cdf_gamma(3,5,21);
                                                  1
(%o2)     1 - gamma_incomplete_regularized(5, -)
                                                  7
(%i3) float(%);
(%o3)               4.402663157376807E-7
```

`quantile_gamma (q,a,b)` [Función]
Devuelve el q-cuantil de una variable aleatoria $Gamma(a, b)$, con $a, b > 0$; en otras palabras, se trata de la inversa de `cdf_gamma`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_gamma (a,b)` [Función]
Devuelve la media de una variable aleatoria $Gamma(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_gamma (a,b)` [Función]
Devuelve la varianza de una variable aleatoria $Gamma(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_gamma (a,b)` [Función]
Devuelve la desviación típica de una variable aleatoria $Gamma(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_gamma (a,b)` [Función]
Devuelve el coeficiente de asimetría de una variable aleatoria $Gamma(a,b)$, con $a,b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_gamma (a,b)` [Función]
Devuelve el coeficiente de curtosis una variable aleatoria $Gamma(a,b)$, con $a,b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_gamma (a,b)` [Función]
`random_gamma (a,b,n)` [Función]
Devuelve un valor aleatorio $Gamma(a,b)$, con $a,b > 0$. Llamando a `random_gamma` con un tercer argumento n, se simulará una muestra aleatoria de tamaño n.

El algoritmo de simulación es una combinación de dos procedimientos, según sea el valor del parámetro a:

Para $a >= 1$, Cheng, R.C.H. y Feast, G.M. (1979). *Some simple gamma variate generators*. Appl. Stat., 28, 3, 290-295.

Para $0 < a < 1$, Ahrens, J.H. y Dieter, U. (1974). *Computer methods for sampling from gamma, beta, poisson and binomial distributions*. Computing, 12, 223-246.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_beta (x,a,b)` [Función]
Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria $Beta(a,b)$, con $a,b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_beta (x,a,b)` [Función]
Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria $Beta(a,b)$, con $a,b > 0$.

```
(%i1) load ("distrib")$
(%i2) cdf_beta(1/3,15,2);
                                11
(%o2)                       --------
                            14348907
(%i3) float(%);
(%o3)              7.666089131388195E-7
```

`quantile_beta (q,a,b)` [Función]
Devuelve el q-cuantil de una variable aleatoria $Beta(a,b)$, con $a,b > 0$; en otras palabras, se trata de la inversa de `cdf_beta`. El argumento q debe ser un número de $[0,1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_beta (a,b)` [Función]
Devuelve la media de una variable aleatoria $Beta(a,b)$, con $a,b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_beta (a,b)` [Función]
Devuelve la varianza de una variable aleatoria $Beta(a,b)$, con $a,b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_beta` (*a*,*b*) [Función]
Devuelve la desviación típica de una variable aleatoria $Beta(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_beta` (*a*,*b*) [Función]
Devuelve el coeficiente de asimetría de una variable aleatoria $Beta(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_beta` (*a*,*b*) [Función]
Devuelve el coeficiente de curtosis de una variable aleatoria $Beta(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_beta` (*a*,*b*) [Función]
`random_beta` (*a*,*b*,*n*) [Función]
Devuelve un valor aleatorio $Beta(a, b)$, con $a, b > 0$. Llamando a `random_beta` con un tercer argumento n, se simulará una muestra aleatoria de tamaño n.

El algoritmo de simulación es el decrito en Cheng, R.C.H. (1978). *Generating Beta Variates with Nonintegral Shape Parameters*. Communications of the ACM, 21:317-322.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_continuous_uniform` (*x*,*a*,*b*) [Función]
Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria *Uniforme Continua*(a, b), con $a < b$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_continuous_uniform` (*x*,*a*,*b*) [Función]
Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria *Uniforme Continua*(a, b), con $a < b$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`quantile_continuous_uniform` (*q*,*a*,*b*) [Función]
Devuelve el q-cuantil de una variable aleatoria *Uniforme Continua*(a, b), con $a < b$; en otras palabras, se trata de la inversa de `cdf_continuous_uniform`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_continuous_uniform` (*a*,*b*) [Función]
Devuelve la media de una variable aleatoria *Uniforme Continua*(a, b), con $a < b$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_continuous_uniform` (*a*,*b*) [Función]
Devuelve la varianza de una variable aleatoria *Uniforme Continua*(a, b), con $a < b$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_continuous_uniform` (*a*,*b*) [Función]
Devuelve la desviación típica de una variable aleatoria *Uniforme Continua*(a, b), con $a < b$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_continuous_uniform (a,b)` [Función]
Devuelve el coeficiente de asimetría de una variable aleatoria *Uniforme Continua*(a, b), con $a < b$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_continuous_uniform (a,b)` [Función]
Devuelve el coeficiente de curtosis una variable aleatoria *Uniforme Continua*(a, b), con $a < b$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_continuous_uniform (a,b)` [Función]
`random_continuous_uniform (a,b,n)` [Función]
Devuelve un valor aleatorio *Uniforme Continuo*(a, b), con $a < b$. Llamando a `random_continuous_uniform` con un tercer argumento n, se simulará una muestra aleatoria de tamaño n.

Esta función es una aplicación directa de la función `random` de Maxima.

Véase también `random`. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_logistic (x,a,b)` [Función]
Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria *Logística*(a, b), con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_logistic (x,a,b)` [Función]
Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria *Logística*(a, b), con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`quantile_logistic (q,a,b)` [Función]
Devuelve el q-cuantil de una variable aleatoria *Logística*(a, b), con $b > 0$; en otras palabras, se trata de la inversa de `cdf_logistic`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_logistic (a,b)` [Función]
Devuelve la media de una variable aleatoria *Logística*(a, b), con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_logistic (a,b)` [Función]
Devuelve la varianza de una variable aleatoria *Logística*(a, b), con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_logistic (a,b)` [Función]
Devuelve la desviación típica de una variable aleatoria *Logística*(a, b), con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_logistic (a,b)` [Función]
Devuelve el coeficiente de asimetría de una variable aleatoria *Logística*(a, b), con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_logistic (a,b)` [Función]
Devuelve el coeficiente de curtosis una variable aleatoria *Logística*(a, b), con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_logistic (a,b)` [Función]
`random_logistic (a,b,n)` [Función]

Devuelve un valor aleatorio *Logístico*(a, b), con $b > 0$. Llamando a `random_logistic` con un tercer argumento n, se simulará una muestra aleatoria de tamaño n.

El algoritmo de simulación está basado en el método inverso.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_pareto (x,a,b)` [Función]

Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria de *Pareto*(a, b), con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_pareto (x,a,b)` [Función]

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria de *Pareto*(a, b), con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`quantile_pareto (q,a,b)` [Función]

Devuelve el q-cuantil de una variable aleatoria de *Pareto*(a, b), con $a, b > 0$; en otras palabras, se trata de la inversa de `cdf_pareto`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_pareto (a,b)` [Función]

Devuelve la media de una variable aleatoria de *Pareto*(a, b), con $a > 1, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_pareto (a,b)` [Función]

Devuelve la varianza de una variable aleatoria de *Pareto*(a, b), con $a > 2, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_pareto (a,b)` [Función]

Devuelve la desviación típica de una variable aleatoria de *Pareto*(a, b), con $a > 2, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_pareto (a,b)` [Función]

Devuelve el coeficiente de asimetría de una variable aleatoria de *Pareto*(a, b), con $a > 3, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_pareto (a,b)` [Función]

Devuelve el coeficiente de curtosis de una variable aleatoria de *Pareto*(a, b), con $a > 4, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_pareto (a,b)` [Función]
`random_pareto (a,b,n)` [Función]

Devuelve un valor aleatorio *Pareto*(a, b), con $a > 0, b > 0$. Llamando a `random_pareto` con un tercer argumento n, se simulará una muestra aleatoria de tamaño n.

El algoritmo de simulación está basado en el método inverso.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_weibull (x,a,b)` [Función]
Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria de $Weibull(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_weibull (x,a,b)` [Función]
Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria de $Weibull(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`quantile_weibull (q,a,b)` [Función]
Devuelve el q-cuantil de una variable aleatoria de $Weibull(a, b)$, con $a, b > 0$; en otras palabras, se trata de la inversa de `cdf_weibull`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_weibull (a,b)` [Función]
Devuelve la media de una variable aleatoria de $Weibull(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_weibull (a,b)` [Función]
Devuelve la varianza de una variable aleatoria de $Weibull(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_weibull (a,b)` [Función]
Devuelve la desviación típica de una variable aleatoria de $Weibull(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_weibull (a,b)` [Función]
Devuelve el coeficiente de asimetría de una variable aleatoria de $Weibull(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_weibull (a,b)` [Función]
Devuelve el coeficiente de curtosis una variable aleatoria de $Weibull(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_weibull (a,b)` [Función]
`random_weibull (a,b,n)` [Función]
Devuelve un valor aleatorio $Weibull(a, b)$, con $a, b > 0$. Llamando a `random_weibull` con un tercer argumento n, se simulará una muestra aleatoria de tamaño n.

El algoritmo de simulación está basado en el método inverso.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_rayleigh (x,b)` [Función]
Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria de $Rayleigh(b)$, con $b > 0$.

La variable aleatoria $Rayleigh(b)$ equivale a una $Weibull(2, 1/b)$.

```
(%i1) load ("distrib")$
(%i2) pdf_rayleigh(x,b);
                                      2  2
                              2     - b  x
(%o2)                    2 b  x %e
```

`cdf_rayleigh` (*x*,*b*) [Función]

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria de *Rayleigh*(b), con $b > 0$.

La variable aleatoria *Rayleigh*(b) equivale a una *Weibull*$(2, 1/b)$.

```
(%i1) load ("distrib")$
(%i2) cdf_rayleigh(x,b);
                                      2  2
                                   - b  x
(%o2)                       1 - %e
```

`quantile_rayleigh` (*q*,*b*) [Función]

Devuelve el q-cuantil de una variable aleatoria de *Rayleigh*(b), con $b > 0$; en otras palabras, se trata de la inversa de `cdf_rayleigh`. El argumento q debe ser un número de $[0, 1]$.

La variable aleatoria *Rayleigh*(b) equivale a una *Weibull*$(2, 1/b)$.

```
(%i1) load ("distrib")$
(%i2) quantile_rayleigh(0.99,b);
                          2.145966026289347
(%o2)                     -----------------
                                  b
```

`mean_rayleigh` (*b*) [Función]

Devuelve la media de una variable aleatoria de *Rayleigh*(b), con $b > 0$.

La variable aleatoria *Rayleigh*(b) equivale a una *Weibull*$(2, 1/b)$.

```
(%i1) load ("distrib")$
(%i2) mean_rayleigh(b);
                                sqrt(%pi)
(%o2)                           ---------
                                   2 b
```

`var_rayleigh` (*b*) [Función]

Devuelve la varianza de una variable aleatoria de *Rayleigh*(b), con $b > 0$.

La variable aleatoria *Rayleigh*(b) equivale a una *Weibull*$(2, 1/b)$.

```
(%i1) load ("distrib")$
(%i2) var_rayleigh(b);
                                    %pi
                                1 - ---
                                     4
(%o2)                           -------
                                   2
                                  b
```

`std_rayleigh` (*b*) [Función]

Devuelve la desviación típica de una variable aleatoria de *Rayleigh*(b), con $b > 0$.

La variable aleatoria *Rayleigh*(b) equivale a una *Weibull*$(2, 1/b)$.

```
(%i1) load ("distrib")$
```

```
(%i2) std_rayleigh(b);
                                 %pi
                           sqrt(1 - ---)
                                  4
(%o2)                      -------------
                                 b
```

`skewness_rayleigh (b)` [Función]

Devuelve el coeficiente de asimetría de una variable aleatoria de *Rayleigh*(b), con $b > 0$.

La variable aleatoria *Rayleigh*(b) equivale a una *Weibull*$(2, 1/b)$.

```
(%i1) load ("distrib")$
(%i2) skewness_rayleigh(b);
                            3/2
                       %pi      3 sqrt(%pi)
                       ------ - -----------
                         4           4
(%o2)                  --------------------
                                %pi 3/2
                           (1 - ---)
                                 4
```

`kurtosis_rayleigh (b)` [Función]

Devuelve el coeficiente de curtosis una variable aleatoria de *Rayleigh*(b), con $b > 0$.

La variable aleatoria *Rayleigh*(b) equivale a una *Weibull*$(2, 1/b)$.

```
(%i1) load ("distrib")$
(%i2) kurtosis_rayleigh(b);
                                    2
                               3 %pi
                          2 - ------
                                16
(%o2)                     ---------- - 3
                               %pi 2
                          (1 - ---)
                                4
```

`random_rayleigh (b)` [Función]

`random_rayleigh (b,n)` [Función]

Devuelve un valor aleatorio *Rayleigh*(b), con $b > 0$. Llamando a `random_rayleigh` con un segundo argumento n, se simulará una muestra aleatoria de tamaño n.

El algoritmo de simulación está basado en el método inverso.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_laplace (x,a,b)` [Función]

Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria de *Laplace*(a, b), con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_laplace (x,a,b)` [Función]
Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria de $Laplace(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`quantile_laplace (q,a,b)` [Función]
Devuelve el q-cuantil de una variable aleatoria de $Laplace(a, b)$, con $b > 0$; en otras palabras, se trata de la inversa de `cdf_laplace`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_laplace (a,b)` [Función]
Devuelve la media de una variable aleatoria de $Laplace(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_laplace (a,b)` [Función]
Devuelve la varianza de una variable aleatoria de $Laplace(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_laplace (a,b)` [Función]
Devuelve la desviación típica de una variable aleatoria de $Laplace(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_laplace (a,b)` [Función]
Devuelve el coeficiente de asimetría de una variable aleatoria de $Laplace(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_laplace (a,b)` [Función]
Devuelve el coeficiente de curtosis una variable aleatoria de $Laplace(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_laplace (a,b)` [Función]
`random_laplace (a,b,n)` [Función]
Devuelve un valor aleatorio $Laplace(a, b)$, con $b > 0$. Llamando a `random_laplace` con un tercer argumento n, se simulará una muestra aleatoria de tamaño n.

El algoritmo de simulación está basado en el método inverso.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_cauchy (x,a,b)` [Función]
Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria de $Cauchy(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_cauchy (x,a,b)` [Función]
Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria de $Cauchy(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`quantile_cauchy (q,a,b)` [Función]
Devuelve el q-cuantil de una variable aleatoria de $Cauchy(a, b)$, con $b > 0$; en otras palabras, se trata de la inversa de `cdf_cauchy`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_cauchy (a,b)` [Función]
`random_cauchy (a,b,n)` [Función]

Devuelve un valor aleatorio $Cauchy(a, b)$, con $b > 0$. Llamando a `random_cauchy` con un tercer argumento n, se simulará una muestra aleatoria de tamaño n.

El algoritmo de simulación está basado en el método inverso.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_gumbel (x,a,b)` [Función]

Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria de $Gumbel(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_gumbel (x,a,b)` [Función]

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria de $Gumbel(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`quantile_gumbel (q,a,b)` [Función]

Devuelve el q-cuantil de una variable aleatoria de $Gumbel(a, b)$, con $b > 0$; en otras palabras, se trata de la inversa de `cdf_gumbel`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_gumbel (a,b)` [Función]

Devuelve la media de una variable aleatoria de $Gumbel(a, b)$, con $b > 0$.

```
(%i1) load ("distrib")$
(%i2) mean_gumbel(a,b);
(%o2)                        %gamma b + a
```

donde el símbolo `%gamma` representa la constante de Euler-Mascheroni. Véase también `%gamma`.

`var_gumbel (a,b)` [Función]

Devuelve la varianza de una variable aleatoria de $Gumbel(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_gumbel (a,b)` [Función]

Devuelve la desviación típica de una variable aleatoria de $Gumbel(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_gumbel (a,b)` [Función]

Devuelve el coeficiente de asimetría de una variable aleatoria de $Gumbel(a, b)$, con $b > 0$.

```
(%i1) load ("distrib")$
(%i2) skewness_gumbel(a,b);
                                    3/2
                                12 6    zeta(3)
(%o2)                           --------------
                                       3
                                    %pi
```

donde `zeta` representa la función zeta de Riemann.

`kurtosis_gumbel` (`a`,`b`) [Función]

Devuelve el coeficiente de curtosis de una variable aleatoria de $Gumbel(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_gumbel` (`a`,`b`) [Función]

`random_gumbel` (`a`,`b`,`n`) [Función]

Devuelve un valor aleatorio $Gumbel(a, b)$, con $b > 0$. Llamando a `random_gumbel` con un tercer argumento n, se simulará una muestra aleatoria de tamaño n.

El algoritmo de simulación está basado en el método inverso.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

46.3 Funciones y variables para distribuciones discretas

`pdf_general_finite_discrete` (`x`,`v`) [Función]

Devuelve el valor correspondiente a x de la función de densidad de una variable aleatoria general discreta finita, con vector de probabilidades v, tal que `Pr(X=i) = v_i`. El vector v puede ser una lista de expresiones no negativas, cuyas componentes se normalizarán para obtener un vector de probabilidades. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

```
(%i1) load ("distrib")$
(%i2) pdf_general_finite_discrete(2, [1/7, 4/7, 2/7]);
                                4
(%o2)                           -
                                7
(%i3) pdf_general_finite_discrete(2, [1, 4, 2]);
                                4
(%o3)                           -
                                7
```

`cdf_general_finite_discrete` (`x`,`v`) [Función]

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria general discreta finita, con vector de probabilidades v.

Véase `pdf_general_finite_discrete` para más detalles.

```
(%i1) load ("distrib")$
(%i2) cdf_general_finite_discrete(2, [1/7, 4/7, 2/7]);
                                5
(%o2)                           -
                                7
(%i3) cdf_general_finite_discrete(2, [1, 4, 2]);
                                5
(%o3)                           -
                                7
(%i4) cdf_general_finite_discrete(2+1/2, [1, 4, 2]);
                                5
(%o4)                           -
                                7
```

`quantile_general_finite_discrete` (*q*,*v*) [Función]
Devuelve el *q*-cuantil de una variable aleatoria general discreta finita, con vector de probabilidades *v*.

Véase `pdf_general_finite_discrete` para más detalles.

`mean_general_finite_discrete` (*v*) [Función]
Devuelve la media de una variable aleatoria general discreta finita, con vector de probabilidades *v*.

Véase `pdf_general_finite_discrete` para más detalles.

`var_general_finite_discrete` (*v*) [Función]
Devuelve la varianza de una variable aleatoria general discreta finita, con vector de probabilidades *v*.

Véase `pdf_general_finite_discrete` para más detalles.

`std_general_finite_discrete` (*v*) [Función]
Devuelve la desviación típica de una variable aleatoria general discreta finita, con vector de probabilidades *v*.

Véase `pdf_general_finite_discrete` para más detalles.

`skewness_general_finite_discrete` (*v*) [Función]
Devuelve el coeficiente de asimetría de una variable aleatoria general discreta finita, con vector de probabilidades *v*.

Véase `pdf_general_finite_discrete` para más detalles.

`kurtosis_general_finite_discrete` (*v*) [Función]
Devuelve el coeficiente de curtosis de una variable aleatoria general discreta finita, con vector de probabilidades *v*.

Véase `pdf_general_finite_discrete` para más detalles.

`random_general_finite_discrete` (*v*) [Función]
`random_general_finite_discrete` (*v*,*m*) [Función]
Devuelve un valor aleatorio de una variable aleatoria general discreta finita, con vector de probabilidades *v*. Llamando a `random_general_finite_discrete` con un segundo argumento *n*, se simulará una muestra aleatoria de tamaño *n*.

Véase `pdf_general_finite_discrete` para más detalles.

```
(%i1) load ("distrib")$
(%i2) random_general_finite_discrete([1,3,1,5]);
(%o2)                          4
(%i3) random_general_finite_discrete([1,3,1,5], 10);
(%o3)           [4, 2, 2, 3, 2, 4, 4, 1, 2, 2]
```

`pdf_binomial` (*x*,*n*,*p*) [Función]
Devuelve el valor correspondiente a x de la función de probabilidad de una variable aleatoria $Binomial(n, p)$, con $0 \leq p \leq 1$ y n entero positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_binomial` (*x*,*n*,*p*) [Función]

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria $Binomial(n, p)$, con $0 \leq p \leq 1$ y n entero positivo.

```
(%i1) load ("distrib")$
(%i2) cdf_binomial(5,7,1/6);
                              7775
(%o2)                         ----
                              7776
(%i3) float(%);
(%o3)                .9998713991769548
```

`quantile_binomial` (*q*,*n*,*p*) [Función]

Devuelve el q-cuantil de una variable aleatoria $Binomial(n, p)$, con $0 \leq p \leq 1$ y n entero positivo; en otras palabras, se trata de la inversa de `cdf_binomial`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_binomial` (*n*,*p*) [Función]

Devuelve la media de una variable aleatoria $Binomial(n, p)$, con $0 \leq p \leq 1$ y n entero positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_binomial` (*n*,*p*) [Función]

Devuelve la varianza de una variable aleatoria $Binomial(n, p)$, con $0 \leq p \leq 1$ y n entero positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_binomial` (*n*,*p*) [Función]

Devuelve la desviación típica de una variable aleatoria $Binomial(n, p)$, con $0 \leq p \leq 1$ y n entero positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_binomial` (*n*,*p*) [Función]

Devuelve el coeficiente de asimetría de una variable aleatoria $Binomial(n, p)$, con $0 \leq p \leq 1$ y n entero positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_binomial` (*n*,*p*) [Función]

Devuelve el coeficiente de curtosis de una variable aleatoria binomial $Binomial(n, p)$, con $0 \leq p \leq 1$ y n entero positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_binomial` (*n*,*p*) [Función]
`random_binomial` (*n*,*p*,*m*) [Función]

Devuelve un valor aleatorio $Binomial(n, p)$, con $0 \leq p \leq 1$ y n entero positivo. Llamando a `random_binomial` con un tercer argumento m, se simulará una muestra aleatoria de tamaño m.

El algoritmo de simulación es el descrito en Kachitvichyanukul, V. y Schmeiser, B.W. (1988) *Binomial Random Variate Generation*. Communications of the ACM, 31, Feb., 216.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_poisson (x,m)` [Función]
Devuelve el valor correspondiente a x de la función de probabilidad de una variable aleatoria de $Poisson(m)$, con $m > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_poisson (x,m)` [Función]
Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria de $Poisson(m)$, con $m > 0$.

```
(%i1) load ("distrib")$
(%i2) cdf_poisson(3,5);
(%o2)       gamma_incomplete_regularized(4, 5)
(%i3) float(%);
(%o3)               .2650259152973623
```

`quantile_poisson (q,m)` [Función]
Devuelve el q-cuantil de una variable aleatoria de $Poisson(m)$, con $m > 0$; en otras palabras, se trata de la inversa de `cdf_poisson`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_poisson (m)` [Función]
Devuelve la media de una variable aleatoria de $Poisson(m)$, con $m > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_poisson (m)` [Función]
Devuelve la varianza de una variable aleatoria de $Poisson(m)$, con $m > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_poisson (m)` [Función]
Devuelve la desviación típica de una variable aleatoria de $Poisson(m)$, con $m > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_poisson (m)` [Función]
Devuelve el coeficiente de asimetría de una variable aleatoria de $Poisson(m)$, con $m > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_poisson (m)` [Función]
Devuelve el coeficiente de curtosis de una variable aleatoria de $Poisson(m)$, con $m > 0$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_poisson (m)` [Función]
`random_poisson (m,n)` [Función]
Devuelve un valor aleatorio $Poisson(m)$, con $m > 0$. Llamando a `random_poisson` con un segundo argumento n, se simulará una muestra aleatoria de tamaño n.

El algoritmo de simulación es el descrito en 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.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

pdf_bernoulli (*x*,*p*) [Función]

Devuelve el valor correspondiente a x de la función de probabilidad de una variable aleatoria de *Bernoulli*(p), con $0 \leq p \leq 1$.

La variable aleatoria *Bernoulli*(p) equivale a una *Binomial*$(1, p)$.

```
(%i1) load ("distrib")$
(%i2) pdf_bernoulli(1,p);
(%o2)                          p
```

cdf_bernoulli (*x*,*p*) [Función]

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria de *Bernoulli*(p), con $0 \leq p \leq 1$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

quantile_bernoulli (*q*,*p*) [Función]

Devuelve el q-cuantil de una variable aleatoria de *Bernoulli*(p), con $0 \leq p \leq 1$; en otras palabras, se trata de la inversa de `cdf_bernoulli`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

mean_bernoulli (*p*) [Función]

Devuelve la media de una variable aleatoria de *Bernoulli*(p), con $0 \leq p \leq 1$.

La variable aleatoria *Bernoulli*(p) equivale a una *Binomial*$(1, p)$.

```
(%i1) load ("distrib")$
(%i2) mean_bernoulli(p);
(%o2)                          p
```

var_bernoulli (*p*) [Función]

Devuelve la varianza de una variable aleatoria de *Bernoulli*(p), con $0 \leq p \leq 1$.

La variable aleatoria *Bernoulli*(p) equivale a una *Binomial*$(1, p)$.

```
(%i1) load ("distrib")$
(%i2) var_bernoulli(p);
(%o2)                      (1 - p) p
```

std_bernoulli (*p*) [Función]

Devuelve la desviación típica de una variable aleatoria de *Bernoulli*(p), con $0 \leq p \leq 1$.

La variable aleatoria *Bernoulli*(p) equivale a una *Binomial*$(1, p)$.

```
(%i1) load ("distrib")$
(%i2) std_bernoulli(p);
(%o2)                      sqrt((1 - p) p)
```

skewness_bernoulli (*p*) [Función]

Devuelve el coeficiente de asimetría de una variable aleatoria de *Bernoulli*(p), con $0 \leq p \leq 1$.

La variable aleatoria *Bernoulli*(p) equivale a una *Binomial*$(1, p)$.

```
(%i1) load ("distrib")$
(%i2) skewness_bernoulli(p);
                                    1 - 2 p
(%o2)                           ---------------
                                sqrt((1 - p) p)
```

`kurtosis_bernoulli (p)` [Función]

Devuelve el coeficiente de curtosis una variable aleatoria de $Bernoulli(p)$, con $0 \leq p \leq 1$.

La variable aleatoria $Bernoulli(p)$ equivale a una $Binomial(1, p)$.

```
(%i1) load ("distrib")$
(%i2) kurtosis_bernoulli(p);
                            1 - 6 (1 - p) p
(%o2)                       ---------------
                               (1 - p) p
```

`random_bernoulli (p)` [Función]

`random_bernoulli (p,n)` [Función]

Devuelve un valor aleatorio $Bernoulli(p)$, con $0 \leq p \leq 1$. Llamando a `random_bernoulli` con un segundo argumento n, se simulará una muestra aleatoria de tamaño n.

Es aplicación directa de la función `random` de Maxima.

Véase también `random`. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_geometric (x,p)` [Función]

Devuelve el valor correspondiente a x de la función de probabilidad de una variable aleatoria $Geométrica(p)$, con $0 < p \leq 1$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_geometric (x,p)` [Función]

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria $Geométrica(p)$, con $0 < p \leq 1$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`quantile_geometric (q,p)` [Función]

Devuelve el q-cuantil de una variable aleatoria $Geométrica(p)$, con $0 < p \leq 1$; en otras palabras, se trata de la inversa de `cdf_geometric`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_geometric (p)` [Función]

Devuelve la media de una variable aleatoria $Geométrica(p)$, con $0 < p \leq 1$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_geometric (p)` [Función]

Devuelve la varianza de una variable aleatoria $Geométrica(p)$, con $0 < p \leq 1$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_geometric (p)` [Función]

Devuelve la desviación típica de una variable aleatoria $Geométrica(p)$, con $0 < p \leq 1$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_geometric (p)` [Función]

Devuelve el coeficiente de asimetría de una variable aleatoria $Geométrica(p)$, con $0 < p \leq 1$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_geometric` (*p*) [Función]
Devuelve el coeficiente de curtosis de una variable aleatoria *Geométrica*(p), con $0 < p \leq 1$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_geometric` (*p*) [Función]
`random_geometric` (*p*,*n*) [Función]
Devuelve un valor aleatorio *Geométrico*(p), con $0 < p \leq 1$. Llamando a `random_geometric` con un segundo argumento n, se simulará una muestra aleatoria de tamaño n.

El algoritmo está basado en la simulación de ensayos de Bernoulli.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_discrete_uniform` (*x*,*n*) [Función]
Devuelve el valor correspondiente a x de la función de probabilidad de una variable aleatoria *Uniforme Discreta*(n), con n entero positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_discrete_uniform` (*x*,*n*) [Función]
Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria *Uniforme Discreta*(n), con n entero positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`quantile_discrete_uniform` (*q*,*n*) [Función]
Devuelve el q-cuantil de una variable aleatoria *Uniforme Discreta*(n), con n entero positivo; en otras palabras, se trata de la inversa de `cdf_discrete_uniform`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_discrete_uniform` (*n*) [Función]
Devuelve la media de una variable aleatoria *Uniforme Discreta*(n), con n entero positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_discrete_uniform` (*n*) [Función]
Devuelve la varianza de una variable aleatoria *Uniforme Discreta*(n), con n entero positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_discrete_uniform` (*n*) [Función]
Devuelve la desviación típica de una variable aleatoria *Uniforme Discreta*(n), con n entero positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_discrete_uniform` (*n*) [Función]
Devuelve el coeficiente de asimetría de una variable aleatoria *Uniforme Discreta*(n), con n entero positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_discrete_uniform` (*n*) [Función]
Devuelve el coeficiente de curtosis de una variable aleatoria *Uniforme Discreta*(n), con n entero positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_discrete_uniform (n)` [Función]
`random_discrete_uniform (n,m)` [Función]

Devuelve un valor aleatorio $UniformeDiscreto(n)$, con n entero positivo. Llamando a `random_discrete_uniform` con un segundo argumento m, se simulará una muestra aleatoria de tamaño m.

Se trata de una aplicación directa de la función `random` de Maxima.

Véase también `random`. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_hypergeometric (x,n1,n2,n)` [Función]

Devuelve el valor correspondiente a x de la función de probabilidad de una variable aleatoria *Hipergeométrica*$(n1, n2, n)$, con *n1*, *n2* y n enteros positivos y $n <= n1+n2$. Siendo *n1* el número de objetos de la clase A, *n2* el número de objetos de la clase B y n el tamaño de una muestra sin reemplazo, esta función devuelve la probabilidad del suceso "extraer exactamente x objetos de la clase A".

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_hypergeometric (x,n1,n2,n)` [Función]

Devuelve el valor correspondiente a x de la función de distribución of una variable aleatoria *Hipergeométrica*$(n1, n2, n)$, con *n1*, *n2* y n enteros positivos y $n <= n1+n2$. Véase `pdf_hypergeometric` para una descripción más completa.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`quantile_hypergeometric (q,n1,n2,n)` [Función]

Devuelve el q-cuantil de una variable aleatoria *Hipergeométrica*$(n1, n2, n)$, con *n1*, *n2* y n enteros positivos y $n <= n1 + n2$; en otras palabras, se trata de la inversa de `cdf_hypergeometric`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_hypergeometric (n1,n2,n)` [Función]

Devuelve la media de una variable aleatoria uniforme discreta $Hyp(n1, n2, n)$, con *n1*, *n2* y n enteros positivos y $n <= n1 + n2$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_hypergeometric (n1,n2,n)` [Función]

Devuelve la varianza de una variable aleatoria *Hipergeométrica*$(n1, n2, n)$, con *n1*, *n2* y n enteros positivos y $n <= n1 + n2$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_hypergeometric (n1,n2,n)` [Función]

Devuelve la desviación típica de una variable aleatoria *Hipergeométrica*$(n1, n2, n)$, con *n1*, *n2* y n enteros positivos y $n <= n1 + n2$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_hypergeometric (n1,n2,n)` [Función]

Devuelve el coeficiente de asimetría de una variable aleatoria *Hipergeométrica*$(n1, n2, n)$, con *n1*, *n2* y n enteros positivos y $n <= n1 + n2$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_hypergeometric (n1,n2,n)` [Función]
`random_hypergeometric (n1,n2,n,m)` [Función]

Devuelve un valor aleatorio *Hipergeométrico*$(n1, n2, n)$, con *n1*, *n2* y *n* enteros positivos y $n <= n1+n2$. Llamando a `random_hypergeometric` con un cuarto argumento m, se simulará una muestra aleatoria de tamaño m.

Algoritmo descrito en Kachitvichyanukul, V., Schmeiser, B.W. (1985) *Computer generation of hypergeometric random variates*. Journal of Statistical Computation and Simulation 22, 127-145.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`pdf_negative_binomial (x,n,p)` [Función]

Devuelve el valor correspondiente a x de la función de probabilidad de una variable aleatoria *Binomial Negativa*(n, p), con $0 < p \leq 1$ y n positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`cdf_negative_binomial (x,n,p)` [Función]

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria *Binomial Negativa*(n, p), con $0 < p \leq 1$ y n positivo.

```
(%i1) load ("distrib")$
(%i2) cdf_negative_binomial(3,4,1/8);
                              3271
(%o2)                        ------
                             524288
```

`quantile_negative_binomial (q,n,p)` [Función]

Devuelve el q-cuantil de una variable aleatoria *Binomial Negativa*(n, p), con $0 < p \leq 1$ y n positivo; en otras palabras, se trata de la inversa de `cdf_negative_binomial`. El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`mean_negative_binomial (n,p)` [Función]

Devuelve la media de una variable aleatoria *Binomial Negativa*(n, p), con $0 < p \leq 1$ and n positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`var_negative_binomial (n,p)` [Función]

Devuelve la varianza de una variable aleatoria *Binomial Negativa*(n, p), con $0 < p \leq 1$ and n positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`std_negative_binomial (n,p)` [Función]

Devuelve la desviación típica de una variable aleatoria *Binomial Negativa*(n, p), con $0 < p \leq 1$ and n positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`skewness_negative_binomial (n,p)` [Función]

Devuelve el coeficiente de asimetría de una variable aleatoria *Binomial Negativa*(n, p), con $0 < p \leq 1$ and n positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`kurtosis_negative_binomial (n,p)` [Función]

Devuelve el coeficiente de curtosis una variable aleatoria binomial negativa $NB(n,p)$, con $0 < p \leq 1$ and n positivo. Para hacer uso de esta función, ejecútese primero `load("distrib")`.

`random_negative_binomial (n,p)` [Función]
`random_negative_binomial (n,p,m)` [Función]

Devuelve un valor aleatorio *Binomial Negativo*(n,p), con $0 < p \leq 1$ y n positivo. Llamando a `random_negative_binomial` con un tercer argumento m, se simulará una muestra aleatoria de tamaño m.

Algoritmo descrito en Devroye, L. (1986) *Non-Uniform Random Variate Generation*. Springer Verlag, p. 480.

Para hacer uso de esta función, ejecútese primero `load("distrib")`.

47 draw

47.1 Introducción a draw

`draw` es un interfaz para comunicar Maxima con Gnuplot.

Tres son las funciones principales a utilizar a nivel de Maxima: `draw2d`, `draw3d` y `draw`.

Síganse estos enlaces para ver ejemplos más elaborados de este paquete:

http://riotorto.users.sourceforge.net/Maxima/gnuplot/

y

http://riotorto.users.sourceforge.net/Maxima/vtk/

Se necesita tener instalado Gnuplot 4.2 o superior para ejecutar este paquete.

47.2 Funciones y variables para draw

47.2.1 Escenas

`gr2d` (*`Opción gráfica`*, ..., *`graphic_object`*, ...) [Constructor de escena]

La función `gr2d` construye un objeto que describe una escena 2d. Los argumentos son *opciones gráficas* y *objetos gráficos* o listas que contengan elementos de ambos tipos. Esta escena se interpreta secuencialmente: las *opciones gráficas* afectan a aquellos *objetos gráficos* colocados a su derecha. Algunas *opciones gráficas* afectan al aspecto global de la escena.

La lista de *objetos gráficos* disponibles para escenas en dos dimensiones: `bars`, `ellipse`, `explicit`, `image`, `implicit`, `label`, `parametric`, `points`, `polar`, `polygon`, `quadrilateral`, `rectangle`, `triangle`, `vector` y `geomap` (este último definido en el paquete `worldmap`).

Véanse también `draw` y `draw2d`.

Para utilizar este objecto, ejecútese primero `load("draw")`.

`gr3d` (*`Opción gráfica`*, ..., *`graphic_object`*, ...) [Constructor de escena]

La función `gr3d` construye un objeto que describe una escena 3d. Los argumentos son *opciones gráficas* y *objetos gráficos* o listas que contengan elementos de ambos tipos. Esta escena se interpreta secuencialmente: las *opciones gráficas* afectan a aquellos *objetos gráficos* colocados a su derecha. Algunas *opciones gráficas* afectan al aspecto global de la escena.

La lista de *objetos gráficos* disponibles para escenas en tres dimensiones: `cylindrical`, `elevation_grid`, `explicit`, `implicit`, `label`, `mesh`, `parametric`, `parametric_surface`, `points`, `quadrilateral`, `spherical`, `triangle`, `tube`, `vector` y `geomap` (este último definido en el paquete `worldmap`).

Véanse también `draw` y `draw3d`.

Para utilizar este objeto, ejecútese primero `load("draw")`.

47.2.2 Funciones

draw (*gr2d*, ..., *gr3d*, ..., *options*, ...) [Función]

Representa gráficamente una serie de escenas; sus argumentos son objetos `gr2d` y/o `gr3d`, junto con algunas opciones, o listas de escenas y opciones. Por defecto, las escenas se representan en una columna.

La función `draw` acepta las siguientes opciones globales: `terminal`, `columns`, `dimensions`, `file_name` y `delay`.

Las funciones `draw2d` y `draw3d` son atajos a utilizar cuando se quiere representar una única escena en dos o tres dimensiones, respectivamente.

Véanse también `gr2d` y `gr3d`.

Para utilizar esta función, ejecútese primero `load("draw")`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) scene1: gr2d(title="Ellipse",
                   nticks=30,
                   parametric(2*cos(t),5*sin(t),t,0,2*%pi))$
(%i3) scene2: gr2d(title="Triangle",
                   polygon([4,5,7],[6,4,2]))$
(%i4) draw(scene1, scene2, columns = 2)$
```

Las dos sentencias gráficas siguientes son equivalentes:

```
(%i1) load("draw")$
(%i2) draw(gr3d(explicit(x^2+y^2,x,-1,1,y,-1,1)));
(%o2)                          [gr3d(explicit)]
(%i3) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1));
(%o3)                          [gr3d(explicit)]
```

Un fichero gif animado:

```
(%i1) load("draw")$
(%i2) draw(
        delay     = 100,
        file_name = "zzz",
        terminal  = 'animated_gif,
        gr2d(explicit(x^2,x,-1,1)),
        gr2d(explicit(x^3,x,-1,1)),
        gr2d(explicit(x^4,x,-1,1)));
End of animation sequence
(%o2)          [gr2d(explicit), gr2d(explicit), gr2d(explicit)]
```

Véanse también `gr2d`, `gr3d`, `draw2d` y `draw3d`.

draw2d (*option*, *graphic_object*, ...) [Función]

Esta función es un atajo para `draw(gr2d(`*options*`, ..., `*graphic_object*`, ...))`.

Puede utilizarse para representar una única escena en 2d.

Para utilizar esta función, ejecútese primero `load("draw")`.

Véanse también `draw` y `gr2d`.

`draw3d` (*option*, *graphic_object*, ...) [Función]

Esta función es un atajo para `draw(gr3d(`*options*`, ..., `*graphic_object*`, ...))`.

Puede utilizarse para representar una única escena en 3d.

Para utilizar esta función, ejecútese primero `load("draw")`.

Véanse también `draw` y `gr3d`.

`draw_file` (*Opción gráfica*, ..., *Opción gráfica*, ...) [Función]

Almacena el gráfico actual en un fichero. Las opciones gráficas que acepta son: `terminal`, `dimensions` y`file_name`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) /* dibujo en pantalla */
      draw(gr3d(explicit(x^2+y^2,x,-1,1,y,-1,1)))$
(%i3) /* same plot in eps format */
      draw_file(terminal  = eps,
                dimensions = [5,5]) $
```

`multiplot_mode` (*term*) [Función]

Esta función permite a Maxima trabajar en modo de gráficos múltiples en una sola ventana del terminal *term*; argumentos válidos para esta función son `screen`, `wxt`, `aquaterm` y `none`.

Cuando el modo de gráficos múltiples está activo, cada llamada a `draw` envía un nuevo gr'afico a la misma ventana, sin borrar los anteriores. Para desactivar el modo de gráficos múltiples escríbase `multiplot_mode(none)`.

Cuando el modo de gráficos múltiples está activo, la opción global `terminal` se bloquea; para desbloquearla y cambiar de terminal es necesario desactivar previamente el modo de gráficos múltiples.

Este modo de trabajo no funciona en plataformas Windows.

Ejemplo:

```
(%i1) load("draw")$
(%i2) set_draw_defaults(
         xrange = [-1,1],
         yrange = [-1,1],
         grid   = true,
         title  = "Step by step plot" )$
(%i3) multiplot_mode(screen)$
(%i4) draw2d(color=blue,  explicit(x^2,x,-1,1))$
(%i5) draw2d(color=red,   explicit(x^3,x,-1,1))$
(%i6) draw2d(color=brown, explicit(x^4,x,-1,1))$
(%i7) multiplot_mode(none)$
```

`set_draw_defaults` (*Opción gráfica*, ..., *Opción gráfica*, ...) [Función]

Establece las opciones gráficas de usuario. Esta función es útil para dibujar varios gráficos con las mismas opciones. Llamando a la función sin argumentos se borran las opciones de usuario por defecto.

Ejemplo:

```
(%i1) load("draw")$
(%i2) set_draw_defaults(
         xrange = [-10,10],
         yrange = [-2, 2],
         color  = blue,
         grid   = true)$
(%i3) /* dibujo con opciones de usuario */
      draw2d(explicit(((1+x)**2/(1+x*x))-1,x,-10,10))$
(%i4) set_draw_defaults()$
(%i5) /* dibujo con opciones por defecto */
      draw2d(explicit(((1+x)**2/(1+x*x))-1,x,-10,10))$
```

Para utilizar esta función, ejecútese primero `load("draw")`.

47.2.3 Opciones gráficas

`adapt_depth` [Opción gráfica]

Valor por defecto: 10

`adapt_depth` es el número máximo de particiones utilizadas por la rutina gráfica adaptativa.

Esta opción sólo es relevante para funciones de tipo `explicit` en 2d.

`allocation` [Opción gráfica]

Valor por defecto: `false`

Con la opción `allocation` es posible colocar a voluntad una escena en la ventana de salida, lo cual resulta de utilidad en el caso de gráficos múltiples. Cuando la opción toma el valor `false`, la escena se coloca de forma automática, dependiendo del valor asignado a la opción `columns`. En cualquier otro caso, a `allocation` se le debe asignar una lista con dos pares de números; el primero se corresponde con la posición de la esquina inferior izquierda del gráfico y el segundo par hace referencia al ancho y alto de la escena. Todas las cantidades deben darse en coordenadas relativas, entre 0 y 1.

Ejemplos:

Gráficos internos.

```
(%i1) load("draw")$
(%i2) draw(
        gr2d(
          explicit(x^2,x,-1,1)),
        gr2d(
          allocation = [[1/4, 1/4],[1/2, 1/2]],
          explicit(x^3,x,-1,1),
          grid = true) ) $
```

Multiplot con dimensiones establecidas por el usuario.

```
(%i1) load("draw")$
(%i2) draw(
        terminal = wxt,
```

```
gr2d(
  allocation = [[0, 0],[1, 1/4]],
  explicit(x^2,x,-1,1)),
gr3d(
  allocation = [[0, 1/4],[1, 3/4]],
  explicit(x^2+y^2,x,-1,1,y,-1,1) ))$
```

Véase también la opción `columns`.

`axis_3d` [Opción gráfica]

Valor por defecto: `true`

Cuando `axis_3d` vale `true`, los ejes x, y y z permanecen visibles en las escenas 3d.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(axis_3d = false,
             explicit(sin(x^2+y^2),x,-2,2,y,-2,2) )$
```

Véanse también `axis_bottom`, `axis_left`, `axis_top` y `axis_right` for axis in 2d.

`axis_bottom` [Opción gráfica]

Valor por defecto: `true`

Cuando `axis_bottom` vale `true`, el eje inferior permanece visible en las escenas 2d.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(axis_bottom = false,
             explicit(x^3,x,-1,1))$
```

Véanse también `axis_left`, `axis_top`, `axis_right` y `axis_3d`.

`axis_left` [Opción gráfica]

Valor por defecto: `true`

Cuando `axis_left` vale `true`, el eje izquierdo permanece visible en las escenas 2d.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(axis_left = false,
             explicit(x^3,x,-1,1))$
```

Véanse también `axis_bottom`, `axis_top`, `axis_right` y `axis_3d`.

`axis_right` [Opción gráfica]

Valor por defecto: `true`

Cuando `axis_right` vale `true`, el eje derecho permanece visible en las escenas 2d.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(axis_right = false,
             explicit(x^3,x,-1,1))$
```

Véanse también `axis_bottom`, `axis_left`, `axis_top` y `axis_3d`.

`axis_top` [Opción gráfica]

Valor por defecto: `true`

Cuando `axis_top` vale `true`, el eje superior permanece visible en las escenas 2d.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(axis_top = false,
             explicit(x^3,x,-1,1))$
```

Véanse también `axis_bottom`, `axis_left`, `axis_right` y `axis_3d`.

`background_color` [Opción gráfica]

Valor por defecto: `white`

Establece el color de fondo en los terminales `gif`, `png`, `jpg` y `gif`. El color de fondo por defecto es blanco.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Esta opción no es compatible con los terminales `epslatex` y `epslatex_standalone`.

Véase también `color`.

`border` [Opción gráfica]

Valor por defecto: `true`

Cuando `border` vale `true`, los bordes de los polígonos se dibujan de acuerdo con `line_type` y `line_width`.

Esta opción afecta a los siguientes objetos gráficos:

- gr2d: `polygon`, `rectangle` y `ellipse`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(color       = brown,
             line_width  = 8,
             polygon([[3,2],[7,2],[5,5]]),
             border      = false,
             fill_color  = blue,
             polygon([[5,2],[9,2],[7,5]]) )$
```

`capping` [Opción gráfica]

Valor por defecto: `[false, false]`

Una lista de dos elementos, `true` y `false`, indicando si los extremos de un objeto gráfico `tube` permanece abiertos o si deben ser cerrados. Por defecto, ambos extremos se dejan abiertos.

La asignación `capping = false` equivale a `capping = [false, false]` y `capping = true` equivale a `capping = [true, true]`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(
        capping = [false, true],
        tube(0, 0, a, 1,
             a, 0, 8) )$
```

`cbrange` [Opción gráfica]

Valor por defecto: `auto`

Cuando `cbrange` vale `auto`, el rango de los valores que se colorean cuando `enhanced3d` es diferente de `false` se calcula automáticamente. Valores fuera del rango utilizan el color del valor extremo más cercano.

Cuando `enhanced3d` o `colorbox` vale `false`, la opción `cbrange` no tiene efecto alguno.

Si el usuario quiere especificar un intervalo para los valores a colorear, éste debe expresarse como una lista de Maxima, como en `cbrange=[-2, 3]`.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d (
        enhanced3d     = true,
        color          = green,
        cbrange = [-3,10],
        explicit(x^2+y^2, x,-2,2,y,-2,2)) $
```

Véanse también `enhanced3d` y `cbtics`.

`cbtics` [Opción gráfica]

Valor por defecto: `auto`

Esta opción gráfica controla la forma en la que se dibujarán las marcas en la escala de color cuando la opción `enhanced3d` sea diferente de `false`.

Cuando `enhanced3d` o `colorbox` vale `false`, la opción `cbtics` no tiene efecto alguno.

Véase `xtics` para una descripción completa.

Ejemplo :

```
(%i1) load("draw")$
(%i2) draw3d (
        enhanced3d = true,
        color      = green,
```

```
                cbtics  = {["High",10],["Medium",05],["Low",0]},
                cbrange = [0, 10],
                explicit(x^2+y^2, x,-2,2,y,-2,2)) $
```

See also `enhanced3d`, `colorbox` and `cbrange`.

`color` [Opción gráfica]

Valor por defecto: `blue`

`color` especifica el color para dibujar líneas, puntos, bordes de polígonos y etiquetas.

Los colores se pueden dar a partir de sus nombres o en código hexadecimal *rgb*.

Los nombres de colores disponibles son: `white`, `black`, `gray0`, `grey0`, `gray10`, `grey10`, `gray20`, `grey20`, `gray30`, `grey30`, `gray40`, `grey40`, `gray50`, `grey50`, `gray60`, `grey60`, `gray70`, `grey70`, `gray80`, `grey80`, `gray90`, `grey90`, `gray100`, `grey100`, `gray`, `grey`, `light_gray`, `light_grey`, `dark_gray`, `dark_grey`, `red`, `light_red`, `dark_red`, `yellow`, `light_yellow`, `dark_yellow`, `green`, `light_green`, `dark_green`, `spring_green`, `forest_green`, `sea_green`, `blue`, `light_blue`, `dark_blue`, `midnight_blue`, `navy`, `medium_blue`, `royalblue`, `skyblue`, `cyan`, `light_cyan`, `dark_cyan`, `magenta`, `light_magenta`, `dark_magenta`, `turquoise`, `light_turquoise`, `dark_turquoise`, `pink`, `light_pink`, `dark_pink`, `coral`, `light_coral`, `orange_red`, `salmon`, `light_salmon`, `dark_salmon`, `aquamarine`, `khaki`, `dark_khaki`, `goldenrod`, `light_goldenrod`, `dark_goldenrod`, `gold`, `beige`, `brown`, `orange`, `dark_orange`, `violet`, `dark_violet`, `plum` y `purple`.

Las componentes cromáticas en código hexadecimal se introducen en el formato `"#rrggbb"`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^2,x,-1,1), /* default is black */
             color = red,
             explicit(0.5 + x^2,x,-1,1),
             color = blue,
             explicit(1 + x^2,x,-1,1),
             color = light_blue,
             explicit(1.5 + x^2,x,-1,1),
             color = "#23ab0f",
             label(["This is a label",0,1.2])  )$
```

Véase también `fill_color`.

`colorbox` [Opción gráfica]

Valor por defecto: `true`

Cuando `colorbox` vale `true`, se dibuja una escala de colores sin título al lado de los objetos `image` en 2D o de objetos coloreados en 3D. Cuando `colorbox` vale `false`, no se presenta la escala de colores. Cuando `colorbox` es una cadena de caracteres, se mostrará la escala de colores con un título.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

Escala de colores e imágenes.

```
(%i1) load("draw")$
(%i2) im: apply('matrix,
                  makelist(makelist(random(200),i,1,30),i,1,30))$
(%i3) draw2d(image(im,0,0,30,30))$
(%i4) draw2d(colorbox = false, image(im,0,0,30,30))$
```

Escala de colores y objeto 3D coloreado.

```
(%i1) load("draw")$
(%i2) draw3d(
        colorbox   = "Magnitude",
        enhanced3d = true,
        explicit(x^2+y^2,x,-1,1,y,-1,1))$
```

Véase también `palette`.

`columns` [Opción gráfica]

Valor por defecto: 1

`columns` es el número de columnas en gráficos múltiples.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia. También puede usarse como argumento de la función `draw`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) scene1: gr2d(title="Ellipse",
                   nticks=30,
                   parametric(2*cos(t),5*sin(t),t,0,2*%pi))$
(%i3) scene2: gr2d(title="Triangle",
                   polygon([4,5,7],[6,4,2]))$
(%i4) draw(scene1, scene2, columns = 2)$
```

`contour` [Opción gráfica]

Valor por defecto: `none`

La opción `contour` permite al usuario decidir dónde colocar las líneas de nivel. Valores posibles son:

- `none`: no se dibujan líneas de nivel.
- `base`: las líneas de nivel se proyectan sobre el plano xy.
- `surface`: las líneas de nivel se dibujan sobre la propia superficie.
- `both`: se dibujan dos conjuntos de líneas de nivel: sobre la superficie y las que se proyectan sobre el plano xy.
- `map`: las líneas de nivel se proyectan sobre el plano xy y el punto de vista del observador se coloca perpendicularmente a él.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
```

```
(%i2) draw3d(explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
             contour_levels = 15,
             contour        = both,
             surface_hide   = true) $
```

Véase también `contour_levels`.

`contour_levels` [Opción gráfica]

Valor por defecto: 5

Esta opción gráfica controla cómo se dibujarán las líneas de nivel. A `contour_levels` se le puede asignar un número entero positivo, una lista de tres números o un conjunto numérico arbitrario:

- Si a `contour_levels` se le asigna un entero positivo n, entonces se dibujarán n líneas de nivel a intervalos iguales. Por defecto, se dibujaán cinco isolíneas.
- Si a `contour_levels` se le asigna una lista de tres números de la forma `[inf,p,sup]`, las isolíneas se dibujarán desde `inf` hasta `sup` en pasos de amplitud `p`.
- Si a `contour_levels` se le asigna un conjunto de números de la forma `{n1, n2, ...}`, entonces se dibujarán las isolíneas correspondientes a los niveles `n1`, `n2`, ...

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplos:

Diez isolíneas igualmente espaciadas. El número real puede ajustarse a fin de poder conseguir etiquetas más sencillas.

```
(%i1) load("draw")$
(%i2) draw3d(color = green,
             explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
             contour_levels = 10,
             contour        = both,
             surface_hide   = true) $
```

Desde -8 hasta 8 en pasos de amplitud 4.

```
(%i1) load("draw")$
(%i2) draw3d(color = green,
             explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
             contour_levels = [-8,4,8],
             contour        = both,
             surface_hide   = true) $
```

Líneas correspondientes a los niveles -7, -6, 0.8 y 5.

```
(%i1) load("draw")$
(%i2) draw3d(color = green,
             explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
             contour_levels = {-7, -6, 0.8, 5},
             contour        = both,
             surface_hide   = true) $
```

Véase también `contour`.

`data_file_name` [Opción gráfica]

Valor por defecto: `"data.gnuplot"`

`data_file_name` es el nombre del fichero que almacena la información numérica que necesita Gnuplot para crear el gráfico solicitado.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia. También puede usarse como argumento de la función `draw`.

Véase ejemplo en `gnuplot_file_name`.

`delay` [Opción gráfica]

Valor por defecto: 5

Este es el retraso en centésimas de segundo entre imágenes en los ficheros gif animados.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia. También puede usarse como argumento de la función `draw`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw(
        delay     = 100,
        file_name = "zzz",
        terminal  = 'animated_gif,
        gr2d(explicit(x^2,x,-1,1)),
        gr2d(explicit(x^3,x,-1,1)),
        gr2d(explicit(x^4,x,-1,1)));
End of animation sequence
(%o2)          [gr2d(explicit), gr2d(explicit), gr2d(explicit)]
```

La opciób `delay` sólo se activa en caso de gifs animados; se ignora en cualquier otro caso.

Véanse también `terminal`, `dimensions`.

`dimensions` [Opción gráfica]

Valor por defecto: `[600,500]`

Dimensiones del terminal de salida. Su valor es una lista formada por el ancho y el alto. El significado de estos dos números depende del terminal con el que se esté trabajando.

Con los terminales `gif`, `animated_gif`, `png`, `jpg`, `svg`, `screen`, `wxt` y `aquaterm`, los enteros representan números de puntos en cada dirección. Si no son enteros se redondean.

Con los terminales `eps`, `eps_color`, `pdf` y `pdfcairo`, los números representan centésimas de cm, lo que significa que, por defecto, las imágenes en estos formatos tienen 6 cm de ancho por 5 cm de alto.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia. También puede usarse como argumento de la función `draw`.

Ejemplos:

La opción `dimensions` aplicada a un fichero de salida y al lienzo wxt.

```
(%i1) load("draw")$
```

```
(%i2) draw2d(
        dimensions = [300,300],
        terminal   = 'png,
        explicit(x^4,x,-1,1)) $
(%i3) draw2d(
        dimensions = [300,300],
        terminal   = 'wxt,
        explicit(x^4,x,-1,1)) $
```

La opción `dimensions` aplicada a una salida eps. En este caso queremos un fichero eps con dimensiones A4.

```
(%i1) load("draw")$
(%i2) A4portrait: 100*[21, 29.7]$
(%i3) draw3d(
        dimensions = A4portrait,
        terminal   = 'eps,
        explicit(x^2-y^2,x,-2,2,y,-2,2)) $
```

`draw_realpart` [Opción gráfica]

Valor por defecto: `true`

Cuando vale `true`, las funciones a dibujar se consideran funciones complejas cuyas partes reales se deben dibujar; cuando la opción vale `false`, no se dibujará nada en caso de que la función no devuelve valores reales.

Esta opción afecta a los objetos `explicit` y `parametric` en 2D y 3D, y al objeto `parametric_surface`.

Ejemplo:

La opción `draw_realpart` afecta a los objetos `explicit` y `parametric`.

```
(%i1) load("draw")$
(%i2) draw2d(
        draw_realpart = false,
        explicit(sqrt(x^2  - 4*x) - x, x, -1, 5),
        color         = red,
        draw_realpart = true,
        parametric(x,sqrt(x^2  - 4*x) - x + 1, x, -1, 5) );
```

`enhanced3d` [Opción gráfica]

Valor por defecto: `none`

Cuando `enhanced3d` vale `none`, las superficies no se colorean en escenas 3D. Para obtener una superficie coloreada se debe asignar una lista a la opción `enhanced3d`, en la que el primer elemento es una expresión y el resto son los nombres de las variables o parámetros utilizados en la expresión. Una lista tal como `[f(x,y,z), x, y, z]` significa que al punto `[x,y,z]` de la superficie se le asigna el número `f(x,y,z)`, el cual será coloreado de acuerdo con el valor actual de `palette`. Para aquellos objetos gráficos 3D definidos en términos de parámetros, es posible definir el número de color en términos de dichos parámetros, como en `[f(u), u]`, para los objetos `parametric` y `tube`, o `[f(u,v), u, v]`, para el objeto `parametric_surface`. Mientras que todos los objetos 3D admiten el modelo basado en coordenadas absolutas, `[f(x,y,z), x,`

`y, z]`, solamente dos de ellos, esto es `explicit` y `elevation_grid`, aceptan también el modelo basado en las coordenadas `[x,y]`, `[f(x,y), x, y]`. El objeto 3D `implicit` acepta solamente el modelo `[f(x,y,z), x, y, z]`. El objeto `points` también acepta el modelo `[f(x,y,z), x, y, z]`, pero cuando los puntos tienen naturaleza cronológica también admite el modelo `[f(k), k]`, siendo `k` un parámetro de orden.

Cuando a `enhanced3d` se le asigna algo diferente de `none`, se ignoran las opciones `color` y `surface_hide`.

Los nombres de las variables definidas en las listas pueden ser diferentes de aquellas utilizadas en las definiciones de los objetos gráficos.

A fin de mantener compatibilidad con versiones anteriores, `enhanced3d = false` es equivalente a `enhanced3d = none` y `enhanced3d = true` es equivalente a `enhanced3d = [z, x, y, z]`. Si a `enhanced3d` se le asigna una expresión, sus variables deben ser las mismas utilizadas en la definición de la superficie. Esto no es necesario cuando se utilizan listas.

Sobre la definición de paletas, véase `palette`.

Ejemplos:

Objeto `explicit` con coloreado definido por el modelo `[f(x,y,z), x, y, z]`.

```
(%i1) load("draw")$
(%i2) draw3d(
         enhanced3d = [x-z/10,x,y,z],
         palette    = gray,
         explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3))$
```

Objeto `explicit` con coloreado definido por el modelo `[f(x,y), x, y]`. Los nombres de las variables definidas en las listas pueden ser diferentes de aquellas utilizadas en las definiciones de los objetos gráficos 3D; en este caso, `r` corresponde a `x` y `s` a `y`.

```
(%i1) load("draw")$
(%i2) draw3d(
         enhanced3d = [sin(r*s),r,s],
         explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3))$
```

Objeto `parametric` con coloreado definido por el modelo `[f(x,y,z), x, y, z]`.

```
(%i1) load("draw")$
(%i2) draw3d(
         nticks = 100,
         line_width = 2,
         enhanced3d = [if y>= 0 then 1 else 0, x, y, z],
         parametric(sin(u)^2,cos(u),u,u,0,4*%pi)) $
```

Objeto `parametric` con coloreado definido por el modelo `[f(u), u]`. En este caso, `(u-1)^2` es una simplificación de `[(u-1)^2,u]`.

```
(%i1) load("draw")$
(%i2) draw3d(
         nticks = 60,
         line_width = 3,
         enhanced3d = (u-1)^2,
         parametric(cos(5*u)^2,sin(7*u),u-2,u,0,2))$
```

Objeto `elevation_grid` con coloreado definido por el modelo `[f(x,y), x, y]`.

```
(%i1) load("draw")$
(%i2) m: apply(
          matrix,
          makelist(makelist(cos(i^2/80-k/30),k,1,30),i,1,20)) $
(%i3) draw3d(
         enhanced3d = [cos(x*y*10),x,y],
         elevation_grid(m,-1,-1,2,2),
         xlabel = "x",
         ylabel = "y");
```

Objeto `tube` con coloreado definido por el modelo `[f(x,y,z), x, y, z]`.

```
(%i1) load("draw")$
(%i2) draw3d(
         enhanced3d = [cos(x-y),x,y,z],
         palette = gray,
         xu_grid = 50,
         tube(cos(a), a, 0, 1, a, 0, 4*%pi) )$
```

Objeto `tube` con coloreado definido por el modelo `[f(u), u]`. En este caso, `enhanced3d = -a` puede ser una simplificación de `enhanced3d = [-foo,foo]`.

```
(%i1) load("draw")$
(%i2) draw3d(
         capping = [true, false],
         palette = [26,15,-2],
         enhanced3d = [-foo, foo],
         tube(a, a, a^2, 1, a, -2, 2) )$
```

Objetos `implicit` y `points` con coloreado definido por el modelo `[f(x,y,z), x, y, z]`.

```
(%i1) load("draw")$
(%i2) draw3d(
         enhanced3d = [x-y,x,y,z],
         implicit((x^2+y^2+z^2-1)*(x^2+(y-1.5)^2+z^2-0.5)=0.015,
                  x,-1,1,y,-1.2,2.3,z,-1,1)) $
(%i3) m: makelist([random(1.0),random(1.0),random(1.0)],k,1,2000)$
(%i4) draw3d(
         point_type = filled_circle,
         point_size = 2,
         enhanced3d = [u+v-w,u,v,w],
         points(m) ) $
```

cuando los puntos tienen naturaleza cronológica también se admite el modelo `[f(k), k]`, siendo `k` un parámetro de orden.

```
(%i1) load("draw")$
(%i2) m:makelist([random(1.0), random(1.0), random(1.0)],k,1,5)$
(%i3) draw3d(
         enhanced3d = [sin(j), j],
         point_size = 3,
```

```
                point_type = filled_circle,
                points_joined = true,
                points(m)) $
```

`error_type` [Opción gráfica]

Valor por defecto: `y`

Dependiendo de su valor, el cual puede ser `x`, `y` o `xy`, el objeto gráfico `errors` dibujará puntos con barras de error horizontales, verticales, o ambas. Si `error_type=boxes`, se dibujarán cajas en lugar de cruces.

Véase también `errors`.

`file_name` [Opción gráfica]

Valor por defecto: `"maxima_out"`

`file_name` es el nombre del fichero en el que los terminales `png`, `jpg`, `gif`, `eps`, `eps_color`, `pdf`, `pdfcairo` y `svg` guardarán el gráfico.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia. También puede usarse como argumento de la función `draw`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(file_name = "myfile",
             explicit(x^2,x,-1,1),
             terminal  = 'png)$
```

Véanse también `terminal`, `dimensions`.

`fill_color` [Opción gráfica]

Valor por defecto: `"red"`

`fill_color` especifica el color para rellenar polígonos y funciones explícitas bidimensionales.

Véase `color` para más información sobre cómo definir colores.

`fill_density` [Opción gráfica]

Valor por defecto: 0

`fill_density` es un número entre 0 y 1 que especifica la intensidad del color de relleno (dado por `fill_color`) en los objetos `bars`.

Véase `bars` para ejemplos.

`filled_func` [Opción gráfica]

Valor por defecto: `false`

La opción `filled_func` establece cómo se van a rellenar las regiones limitadas por funciones. Si `filled_func` vale `true`, la región limitada por la función definida en el objeto `explicit` y el borde inferior del la ventana gráfica se rellena con `fill_color`. Si `filled_func` guarda la expresión de una función, entonces la región limitada por esta función y la definida en el objeto `explicit` será la que se rellene. Por defecto, las funciones explícitas no se rellenan.

Un caso de especial utilidad es `filled_func=0`, con lo que se sombrea la región limitada por el eje horizontal y la función explícita.

Esta opción sólo afecta al objeto gráfico bidimensional `explicit`.

Ejemplo:

Región limitada por un objeto `explicit` y el borde inferior de la ventana gráfica.

```
(%i1) load("draw")$
(%i2) draw2d(fill_color  = red,
             filled_func = true,
             explicit(sin(x),x,0,10) )$
```

Región limitada por un objeto `explicit` y la función definida en la opción `filled_func`. Nótese que la variable en `filled_func` debe ser la misma que la utilizada en `explicit`.

```
(%i1) load("draw")$
(%i2) draw2d(fill_color  = grey,
             filled_func = sin(x),
             explicit(-sin(x),x,0,%pi));
```

Véanse también `fill_color` y `explicit`.

`font` [Opción gráfica]

Valor por defecto: `""` (cadena vacía)

Esta opción permite seleccionar el tipo de fuente a utilizar por el terminal. Sólo se puede utilizar un tipo de fuente y tamaño por gráfico.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Véase también `font_size`.

Gnuplot no puede gestionar por sí mismo las fuentes, dejando esta tarea a las librerías que dan soporte a los diferentes terminales, cada uno con su propia manera de controlar la tipografía. A continuación un breve resumen:

- *x11*: Utiliza el mecanismo habitual para suministrar las fuentes en x11.

  Ejemplo:

  ```
  (%i1) load("draw")$
  (%i2) draw2d(font      = "Arial",
               font_size = 20,
               label(["Arial font, size 20",1,1]))$
  ```

- *windows*: El terminal de windows no permite cambiar fuentes desde dentro del gráfico. Una vez se ha creado el gráfico, se pueden cambiar las fuentes haciendo clic derecho en el menú de la ventana gráfica.

- *png, jpeg, gif*: La librería *libgd* utiliza la ruta a las fuentes almacenada en la variable de entorno `GDFONTPATH`; en tal caso sólo es necesario darle a la opción `font` el nombre de la fuente. También es posible darle la ruta completa al fichero de la fuente.

  Ejemplos:

  A la opción `font` se le puede dar la ruta completa al fichero de la fuente:

  ```
  (%i1) load("draw")$
  (%i2) path: "/usr/share/fonts/truetype/freefont/" $
  ```

```
(%i3) file: "FreeSerifBoldItalic.ttf" $
(%i4) draw2d(
        font      = concat(path, file),
        font_size = 20,
        color     = red,
        label(["FreeSerifBoldItalic font, size 20",1,1]),
        terminal  = png)$
```

Si la variable de entorno `GDFONTPATH` almacena la ruta a la carpeta donde se alojan las fuentes, es posible darle a la opción `font` sólo el nombre de la fuente:

```
(%i1) load("draw")$
(%i2) draw2d(
        font      = "FreeSerifBoldItalic",
        font_size = 20,
        color     = red,
        label(["FreeSerifBoldItalic font, size 20",1,1]),
        terminal  = png)$
```

- *Postscript*: Las fuentes estándar de Postscript son: `"Times-Roman"`, `"Times-Italic"`, `"Times-Bold"`, `"Times-BoldItalic"`, `"Helvetica"`, `"Helvetica-Oblique"`, `"Helvetica-Bold"`, `"Helvetic-BoldOblique"`, `"Courier"`, `"Courier-Oblique"`, `"Courier-Bold"` y `"Courier-BoldOblique"`.

  Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(
        font      = "Courier-Oblique",
        font_size = 15,
        label(["Courier-Oblique font, size 15",1,1]),
        terminal = eps)$
```

- *pdf*: Utiliza las mismas fuentes que *Postscript*.
- *pdfcairo*: Utiliza las mismas fuentes que *wxt*.
- *wxt*: La librería *pango* encuentra las fuentes por medio de la utilidad `fontconfig`.
- *aqua*: La fuente por defecto es `"Times-Roman"`.

La documentación de gnuplot es una importante fuente de información sobre terminales y fuentes.

`font_size` [Opción gráfica]

Valor por defecto: 10

Esta opción permite seleccionar el tamaño de la fuente a utilizar por el terminal. Sólo se puede utilizar un tipo de fuente y tamaño por gráfico. `font_size` sólo se activa cuando la opción `font` tiene un valor diferente de la cadena vacía.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Véase también `font`.

`gnuplot_file_name` [Opción gráfica]

Valor por defecto: `"maxout.gnuplot"`

`gnuplot_file_name` es el nombre del fichero que almacena las instrucciones a ser procesadas por Gnuplot.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia. También puede usarse como argumento de la función `draw`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(
       file_name = "my_file",
       gnuplot_file_name = "my_commands_for_gnuplot",
       data_file_name    = "my_data_for_gnuplot",
       terminal          = png,
       explicit(x^2,x,-1,1)) $
```

Véase también `data_file_name`.

`grid` [Opción gráfica]

Valor por defecto: `false`

Cuando

Cuando `grid` toma un valor distinto de `false`, se dibujará una rejilla sobre el plano *xy*. Si a `grid` se le asigna el valor `true`, se dibujará una línea de la rejilla por cada marca que haya sobre los ejes. Si a `grid` se le asigna la lista `[nx,ny]`, con *[nx,ny]* > *[0,0]*, se dibujarán `nx` líneas por cada marca del eje-*x* y *ny* líneas por cada marca del eje-*y*.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(grid = true,
             explicit(exp(u),u,-2,2))$
```

`head_angle` [Opción gráfica]

Valor por defecto: 45

`head_angle` indica el ángulo, en grados, entre la flecha y el segmento del vector.

Esta opción sólo es relevante para objetos de tipo `vector`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(xrange      = [0,10],
             yrange      = [0,9],
             head_length = 0.7,
             head_angle  = 10,
             vector([1,1],[0,6]),
             head_angle  = 20,
             vector([2,1],[0,6]),
             head_angle  = 30,
             vector([3,1],[0,6]),
             head_angle  = 40,
```

```
                vector([4,1],[0,6]),
                head_angle  = 60,
                vector([5,1],[0,6]),
                head_angle  = 90,
                vector([6,1],[0,6]),
                head_angle  = 120,
                vector([7,1],[0,6]),
                head_angle  = 160,
                vector([8,1],[0,6]),
                head_angle  = 180,
                vector([9,1],[0,6]) )$
```

Véanse también `head_both`, `head_length` y `head_type`.

`head_both` [Opción gráfica]

Valor por defecto: `false`

Cuando `head_both` vale `true`, los vectores se dibujan bidireccionales. Si vale `false`, se dibujan unidireccionales.

Esta opción sólo es relevante para objetos de tipo `vector`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(xrange      = [0,8],
             yrange      = [0,8],
             head_length = 0.7,
             vector([1,1],[6,0]),
             head_both   = true,
             vector([1,7],[6,0]) )$
```

Véanse también `head_length`, `head_angle` y `head_type`.

`head_length` [Opción gráfica]

Valor por defecto: 2

`head_length` indica, en las unidades del eje x, la longitud de las flechas de los vectores.

Esta opción sólo es relevante para objetos de tipo `vector`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(xrange      = [0,12],
             yrange      = [0,8],
             vector([0,1],[5,5]),
             head_length = 1,
             vector([2,1],[5,5]),
             head_length = 0.5,
             vector([4,1],[5,5]),
             head_length = 0.25,
             vector([6,1],[5,5]))$
```

Véanse también `head_both`, `head_angle` y `head_type`.

`head_type` [Opción gráfica]

Valor por defecto: `filled`

`head_type` se utiliza para especificar cómo se habrán de dibujar las flechas de los vectores. Los valores posibles para esta opción son: `filled` (flechas cerradas y rellenas), `empty` (flechas cerradas pero no rellenas) y `nofilled` (flechas abiertas).

Esta opción sólo es relevante para objetos de tipo `vector`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(xrange      = [0,12],
             yrange      = [0,10],
             head_length = 1,
             vector([0,1],[5,5]), /* default type */
             head_type = 'empty,
             vector([3,1],[5,5]),
             head_type = 'nofilled,
             vector([6,1],[5,5]))$
```

Véanse también `head_both`, `head_angle` y `head_length`.

`interpolate_color` [Opción gráfica]

Valor por defecto: `false`

Esta opción solo es relevante si `enhanced3d` tiene un valor diferente de `false`.

Si `interpolate_color` vale `false`, las superficies se colorean con cuadriláteros homogéneos. Si vale `true`, las transiciones de colores se suavizan por interpolación.

La opción `interpolate_color` también acepta una lista de dos números, `[m,n]`. Para *m* y *n* positivos, cada cuadrilátero o triángulo se interpola *m* y *n* veces en la dirección respectiva. Para *m* y *n* negativos, la frecuencia de interpolación se elige de forma que se dibujen al menos *abs(m)* y *abs(n)* puntos, pudiéndose considerar esto como una función especial de enrejado. Con valores nulos, esto es `interpolate_color=[0,0]`, se seleccionará un número óptimo de puntos interpolados en la superficie.

Además, `interpolate_color=true` es equivalente a `interpolate_color=[0,0]`.

La interpolación de colores en superficies paramétricas puede dar resultados imprevisibles.

Ejemplos:

Interpolación de color con funciones explícitas.

```
(%i1) load("draw")$
(%i2) draw3d(
        enhanced3d   = sin(x*y),
        explicit(20*exp(-x^2-y^2)-10, x ,-3, 3, y, -3, 3)) $
(%i3) draw3d(
        interpolate_color = true,
        enhanced3d   = sin(x*y),
        explicit(20*exp(-x^2-y^2)-10, x ,-3, 3, y, -3, 3)) $
(%i4) draw3d(
        interpolate_color = [-10,0],
```

```
        enhanced3d   = sin(x*y),
        explicit(20*exp(-x^2-y^2)-10, x ,-3, 3, y, -3, 3)) $
```

Interpolación de color con el objeto `mesh`.

```
(%i1) load("draw")$
(%i2) draw3d(
        enhanced3d = true,
        mesh([[1,1,3],   [7,3,1],[12,-2,4],[15,0,5]],
             [[2,7,8],   [4,3,1],[10,5,8], [12,7,1]],
             [[-2,11,10],[6,9,5],[6,15,1], [20,15,2]])) $
(%i3) draw3d(
        enhanced3d        = true,
        interpolate_color = true,
        mesh([[1,1,3],   [7,3,1],[12,-2,4],[15,0,5]],
             [[2,7,8],   [4,3,1],[10,5,8], [12,7,1]],
             [[-2,11,10],[6,9,5],[6,15,1], [20,15,2]])) $
(%i4) draw3d(
        enhanced3d        = true,
        interpolate_color = true,
        view=map,
        mesh([[1,1,3],   [7,3,1],[12,-2,4],[15,0,5]],
             [[2,7,8],   [4,3,1],[10,5,8], [12,7,1]],
             [[-2,11,10],[6,9,5],[6,15,1], [20,15,2]])) $
```

Véase también `enhanced3d`.

`ip_grid` [Opción gráfica]

Valor por defecto: `[50, 50]`

`ip_grid` establece la rejilla del primer muestreo para los gráficos de funciones implícitas.

Esta opción sólo es relevante para funciones de tipo `implicit`.

`ip_grid_in` [Opción gráfica]

Valor por defecto: `[5, 5]`

`ip_grid_in` establece la rejilla del segundo muestreo para los gráficos de funciones implícitas.

Esta opción sólo es relevante para funciones de tipo `implicit`.

`key` [Opción gráfica]

Valor por defecto: `""` (cadena vacía)

`key` es la clave de una función en la leyenda. Si `key` es una cadena vacía, las funciones no tendrán clave asociada en la leyenda.

Esta opción afecta a los siguientes objetos gráficos:

- `gr2d`: `points`, `polygon`, `rectangle`, `ellipse`, `vector`, `explicit`, `implicit`, `parametric` y `polar`.
- `gr3d`: `points`, `explicit`, `parametric`, y `parametric_surface`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(key   = "Sinus",
             explicit(sin(x),x,0,10),
             key   = "Cosinus",
             color = red,
             explicit(cos(x),x,0,10) )$
```

`key_pos` [Opción gráfica]

Valor por defecto: `""` (cadena vacía)

La opción `key_pos` establece en qué posición se colocará la leyenda. Si `key` es una cadena vacía, entonces se utilizará por defecto la posición `"top_right"`. Los valores disponibles para esta opción son: `top_left`, `top_center`, `top_right`, `center_left`, `center`, `center_right`, `bottom_left`, `bottom_center` y `bottom_right`.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplos:

```
(%i1) load("draw")$
(%i2) draw2d(
        key_pos = top_left,
        key     = "x",
        explicit(x,  x,0,10),
        color   = red,
        key     = "x squared",
        explicit(x^2,x,0,10))$
(%i3) draw3d(
        key_pos = center,
        key     = "x",
        explicit(x+y,x,0,10,y,0,10),
        color= red,
        key     = "x squared",
        explicit(x^2+y^2,x,0,10,y,0,10))$
```

`label_alignment` [Opción gráfica]

Valor por defecto: `center`

`label_alignment` se utiliza para especificar dónde se escribirán las etiquetas con respecto a las coordenadas de referencia. Los valores posibles para esta opción son: `center`, `left` y `right`.

Esta opción sólo es relevante para objetos de tipo `label`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(xrange          = [0,10],
             yrange          = [0,10],
             points_joined   = true,
             points([[5,0],[5,10]]),
```

```
                color           = blue,
                label(["Centered alignment (default)",5,2]),
                label_alignment = 'left,
                label(["Left alignment",5,5]),
                label_alignment = 'right,
                label(["Right alignment",5,8]))$
```

Véanse también `label_orientation` y `color`.

`label_orientation` [Opción gráfica]

Valor por defecto: `horizontal`

`label_orientation` se utiliza para especificar la orientación de las etiquetas. Los valores posibles para esta opción son: `horizontal` y `vertical`.

Esta opción sólo es relevante para objetos de tipo `label`.

Ejemplo:

En este ejemplo, el punto ficticio que se añade sirve para obtener la imagen, ya que el paquete `draw` necesita siempre de datos para construir la escena.

```
(%i1) load("draw")$
(%i2) draw2d(xrange     = [0,10],
             yrange     = [0,10],
             point_size = 0,
             points([[5,5]]),
             color      = navy,
             label(["Horizontal orientation (default)",5,2]),
             label_orientation = 'vertical,
             color             = "#654321",
             label(["Vertical orientation",1,5]))$
```

Véanse también `label_alignment` y `color`.

`line_type` [Opción gráfica]

Valor por defecto: `solid`

`line_type` indica cómo se van a dibujar las líneas; valores posibles son `solid` y `dots`, que están disponibles en todos los terminales, y `dashes`, `short_dashes`, `short_long_dashes`, `short_short_long_dashes` y `dot_dash`, que no esán disponibles en los terminales `png`, `jpg` y `gif`.

Esta opción afecta a los siguientes objetos gráficos:

- `gr2d`: `points`, `polygon`, `rectangle`, `ellipse`, `vector`, `explicit`, `implicit`, `parametric` y `polar`.
- `gr3d`: `points`, `explicit`, `parametric` y `parametric_surface`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(line_type = dots,
             explicit(1 + x^2,x,-1,1),
             line_type = solid, /* default */
             explicit(2 + x^2,x,-1,1))$
```

Véase también `line_width`.

`line_width` [Opción gráfica]

Valor por defecto: 1

`line_width` es el ancho de las líneas a dibujar. Su valor debe ser un número positivo.

Esta opción afecta a los siguientes objetos gráficos:

- gr2d: `points`, `polygon`, `rectangle`, `ellipse`, `vector`, `explicit`, `implicit`, `parametric` y `polar`.
- gr3d: `points` y `parametric`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^2,x,-1,1), /* default width */
             line_width = 5.5,
             explicit(1 + x^2,x,-1,1),
             line_width = 10,
             explicit(2 + x^2,x,-1,1))$
```

Véase también `line_type`.

`logcb` [Opción gráfica]

Valor por defecto: `false`

Cuando `logcb` vale `true`, la escala de colores se dibuja logarítmicamente.

Cuando `enhanced3d` o `colorbox` vale `false`, la opción `logcb` no tiene efecto alguno.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d (
        enhanced3d = true,
        color      = green,
        logcb = true,
        logz  = true,
        palette = [-15,24,-9],
        explicit(exp(x^2-y^2), x,-2,2,y,-2,2)) $
```

Véanse también `enhanced3d`, `colorbox` y `cbrange`.

`logx` [Opción gráfica]

Valor por defecto: `false`

Cuando `logx` vale `true`, el eje x se dibuja en la escala logarítmica.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(log(x),x,0.01,5),
             logx = true)$
```

Véanse también `logy`, `logx_secondary`, `logy_secondary` y `logz`.

`logx_secondary` [Opción gráfica]

Valor por defecto: `false`

Si `logx_secondary` vale `true`, el eje secundario de x se dibuja en la escala logarítmica.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(
        grid = true,
        key="x^2, linear scale",
        color=red,
        explicit(x^2,x,1,100),
        xaxis_secondary = true,
        xtics_secondary = true,
        logx_secondary  = true,
        key = "x^2, logarithmic x scale",
        color = blue,
        explicit(x^2,x,1,100) )$
```

Véanse también `logx`, `logy`, `logy_secondary` y `logz`.

`logy` [Opción gráfica]

Valor por defecto: `false`

Cuando `logy` vale `true`, el eje y se dibuja en la escala logarítmica.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(logy = true,
             explicit(exp(x),x,0,5))$
```

Véanse también `logx`, `logx_secondary`, `logy_secondary` y `logz`.

`logy_secondary` [Opción gráfica]

Valor por defecto: `false`

Si `logy_secondary` vale `true`, el eje secundario de y se dibuja en la escala logarítmica.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(
        grid = true,
        key="x^2, linear scale",
        color=red,
        explicit(x^2,x,1,100),
        yaxis_secondary = true,
```

```
                ytics_secondary = true,
                logy_secondary  = true,
                key = "x^2, logarithmic y scale",
                color = blue,
                explicit(x^2,x,1,100) )$
```

Véanse también `logx`, `logy`, `logx_secondary` y `logz`.

`logz` [Opción gráfica]

Valor por defecto: `false`

Cuando `logz` vale `true`, el eje z se dibuja en la escala logarítmica.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(logz = true,
             explicit(exp(u^2+v^2),u,-2,2,v,-2,2))$
```

Véanse también `logx` and `logy`.

`nticks` [Opción gráfica]

Valor por defecto: 29

En 2d, `nticks` es el número de puntos a utilizar por el programa adaptativo que genera las funciones explícitas. También es el número de puntos que se representan en las curvas paramétricas y polares.

Esta opción afecta a los siguientes objetos gráficos:

- `gr2d`: `ellipse`, `explicit`, `parametric` y `polar`.
- `gr3d`: `parametric`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(transparent = true,
             ellipse(0,0,4,2,0,180),
             nticks = 5,
             ellipse(0,0,4,2,180,180) )$
```

`palette` [Opción gráfica]

Valor por defecto: `color`

`palette` indica cómo transformar niveles de gris en componentes cromáticas. Trabaja conjuntamente con la opción `enhanced3d` en gráficos 3D, la cual asocia cada punto de una superficie con un número real o nivel de gris. También trabaja con imágenes grises. Con `palette`, estos niveles se transforman en colores.

Hay dos formas de definir estas transformaciones.

En primer lugar, `palette` puede ser un vector de longitud tres con sus componentes tomando valores enteros en el rango desde -36 a +36; cada valor es un índice para seleccionar una fórmula que transforma los niveles numéricos en las componentes cromáticas rojo, verde y azul:

```
 0: 0              1: 0.5          2: 1
```

```
 3: x               4: x^2           5: x^3
 6: x^4             7: sqrt(x)       8: sqrt(sqrt(x))
 9: sin(90x)       10: cos(90x)     11: |x-0.5|
12: (2x-1)^2       13: sin(180x)    14: |cos(180x)|
15: sin(360x)      16: cos(360x)    17: |sin(360x)|
18: |cos(360x)|    19: |sin(720x)|  20: |cos(720x)|
21: 3x             22: 3x-1         23: 3x-2
24: |3x-1|         25: |3x-2|       26: (3x-1)/2
27: (3x-2)/2       28: |(3x-1)/2|   29: |(3x-2)/2|
30: x/0.32-0.78125 31: 2*x-0.84     32: 4x;1;-2x+1.84;x/0.08-11.5
33: |2*x - 0.5|    34: 2*x          35: 2*x - 0.5
36: 2*x - 1
```

los números negativos se interpretan como colores invertidos de las componentes cromáticas. `palette = gray` y `palette = color` son atajos para `palette = [3,3,3]` y `palette = [7,5,15]`, respectivamente.

En segundo lugar, `palette` puede ser una paleta de colores definida por el usuario. En este caso, el formato para crear una paleta de longitud `n` es `palette=[color_1, color_2, ..., color_n]`, donde `color_i` es un color correctamente definido (véase la opción `color`), de tal manera que `color_1` se asigna al valor más bajo del nivel y `color_n` al más alto. El resto de colores se interpolan.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplos:

Trabaja conjuntamente con la opción `enhanced3d` en gráficos 3D.

```
(%i1) load("draw")$
(%i2) draw3d(
        enhanced3d = [z-x+2*y,x,y,z],
        palette = [32, -8, 17],
        explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3))$
```

También trabaja con imágenes grises.

```
(%i1) load("draw")$
(%i2) im: apply(
           'matrix,
            makelist(makelist(random(200),i,1,30),i,1,30))$
(%i3) /* palette = color, default */
      draw2d(image(im,0,0,30,30))$
(%i4) draw2d(palette = gray, image(im,0,0,30,30))$
(%i5) draw2d(palette = [15,20,-4],
             colorbox=false,
             image(im,0,0,30,30))$
```

`palette` puede ser una paleta de colores definida por el usuario. En este ejemplo, valores bajos de `x` se colorean en rojo y altos en amarillo.

```
(%i1) load("draw")$
(%i2) draw3d(
         palette = [red, blue, yellow],
```

```
                enhanced3d = x,
                explicit(x^2+y^2,x,-1,1,y,-1,1)) $
```

Véase también `colorbox` y `enhanced3d`.

`point_size` [Opción gráfica]

Valor por defecto: 1

`point_size` establece el tamaño de los puntos dibujados. Debe ser un número no negativo.

Esta opción no tiene efecto alguno cuando a la opción gráfica `point_type` se le ha dado el valor `dot`.

Esta opción afecta a los siguientes objetos gráficos:

- `gr2d`: `points`.
- `gr3d`: `points`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(
        points(makelist([random(20),random(50)],k,1,10)),
        point_size = 5,
        points(makelist(k,k,1,20),makelist(random(30),k,1,20)))$
```

`point_type` [Opción gráfica]

Valor por defecto: 1

`point_type` indica cómo se van a dibujar los puntos aislados. Los valores para esta opción pueden ser índices enteros mayores o iguales que -1, o también nombres de estilos: `$none` (-1), `dot` (0), `plus` (1), `multiply` (2), `asterisk` (3), `square` (4), `filled_square` (5), `circle` (6), `filled_circle` (7), `up_triangle` (8), `filled_up_triangle` (9), `down_triangle` (10), `filled_down_triangle` (11), `diamant` (12) y `filled_diamant` (13).

Esta opción afecta a los siguientes objetos gráficos:

- `gr2d`: `points`.
- `gr3d`: `points`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(xrange = [0,10],
             yrange = [0,10],
             point_size = 3,
             point_type = diamant,
             points([[1,1],[5,1],[9,1]]),
             point_type = filled_down_triangle,
             points([[1,2],[5,2],[9,2]]),
             point_type = asterisk,
             points([[1,3],[5,3],[9,3]]),
             point_type = filled_diamant,
             points([[1,4],[5,4],[9,4]]),
```

```
                    point_type = 5,
                    points([[1,5],[5,5],[9,5]]),
                    point_type = 6,
                    points([[1,6],[5,6],[9,6]]),
                    point_type = filled_circle,
                    points([[1,7],[5,7],[9,7]]),
                    point_type = 8,
                    points([[1,8],[5,8],[9,8]]),
                    point_type = filled_diamant,
                    points([[1,9],[5,9],[9,9]]) )$
```

`points_joined` [Opción gráfica]

Valor por defecto: `false`

Cuando `points_joined` vale `true`, los puntos se unen con segmentos; si vale `false`, se dibujarán puntos aislados. Un tercer valor posible para esta opción gráfica es `impulses`; en tal caso, se dibujarán segmentos verticales desde los puntos hasta el eje-x (2D) o hasta el plano-xy (3D).

Esta opción afecta a los siguientes objetos gráficos:

- `gr2d`: `points`.
- `gr3d`: `points`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(xrange        = [0,10],
             yrange        = [0,4],
             point_size    = 3,
             point_type    = up_triangle,
             color         = blue,
             points([[1,1],[5,1],[9,1]]),
             points_joined = true,
             point_type    = square,
             line_type     = dots,
             points([[1,2],[5,2],[9,2]]),
             point_type    = circle,
             color         = red,
             line_width    = 7,
             points([[1,3],[5,3],[9,3]]) )$
```

`proportional_axes` [Opción gráfica]

Valor por defecto: `none`

Cuando `proportional_axes` es igual a `xy` o `xy`, una escena 2D o 3D se dibujará con los ejes proporcionales a sus longitudes relativas.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Esta opción sólo funciona con Gnuplot versión 4.2.6 o superior.

Ejemplos:

Gráfico en 2D.

```
(%i1) load("draw")$
(%i2) draw2d(
        ellipse(0,0,1,1,0,360),
        transparent=true,
        color = blue,
        line_width = 4,
        ellipse(0,0,2,1/2,0,360),
        proportional_axes = xy) $
```

Multiplot.

```
(%i1) load("draw")$
(%i2) draw(
        terminal = wxt,
        gr2d(proportional_axes = xy,
             explicit(x^2,x,0,1)),
        gr2d(explicit(x^2,x,0,1),
             xrange = [0,1],
             yrange = [0,2],
             proportional_axes=xy),
        gr2d(explicit(x^2,x,0,1)))$
```

`surface_hide` [Opción gráfica]

Valor por defecto: `false`

Cuando `surface_hide` vale `true`, las partes ocultas no se muestran en las superficies de las escenas 3d.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw(columns=2,
           gr3d(explicit(exp(sin(x)+cos(x^2)),x,-3,3,y,-3,3)),
           gr3d(surface_hide = true,
                explicit(exp(sin(x)+cos(x^2)),x,-3,3,y,-3,3)) )$
```

`terminal` [Opción gráfica]

Valor por defecto: `screen`

Selecciona el terminal a utilizar por Gnuplot; valores posibles son: `screen` (por defecto), `png`, `pngcairo`, `jpg`, `gif`, `eps`, `eps_color`, `epslatex`, `epslatex_standalone`, `svg`, `dumb`, `dumb_file`, `pdf`, `pdfcairo`, `wxt`, `animated_gif`, `multipage_pdfcairo`, `multipage_pdf`, `multipage_eps`, `multipage_eps_color` y `aquaterm`.

Los terminales `screen`, `wxt` y `aquaterm` también se pueden definir como una lista de dos elementos: el propio nombre del terminal y un número entero no negativo. De esta forma se pueden abrir varias ventanas al mismo tiempo, cada una de ellas con su número correspondiente. Esta modalidad no funciona en plataformas Windows.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia. También puede usarse como argumento de la función `draw`.

pdfcairo necesita Gnuplot 4.3. `pdf` necesita que Gnuplot haya sido compilado con la opción `--enable-pdf` y libpdf debe estar instalado (http://www.pdflib.com/en/download/pdflib-family/pdflib-lite/).

Ejemplos:

```
(%i1) load("draw")$
(%i2) /* screen terminal (default) */
      draw2d(explicit(x^2,x,-1,1))$
(%i3) /* png file */
      draw2d(terminal  = 'png,
             explicit(x^2,x,-1,1))$
(%i4) /* jpg file */
      draw2d(terminal   = 'jpg,
             dimensions = [300,300],
             explicit(x^2,x,-1,1))$
(%i5) /* eps file */
      draw2d(file_name = "myfile",
             explicit(x^2,x,-1,1),
             terminal  = 'eps)$
(%i6) /* pdf file */
      draw2d(file_name = "mypdf",
             dimensions = 100*[12.0,8.0],
             explicit(x^2,x,-1,1),
             terminal  = 'pdf)$
(%i7) /* wxwidgets window */
      draw2d(explicit(x^2,x,-1,1),
             terminal  = 'wxt)$
```

Ventanas múltiples.

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^5,x,-2,2), terminal=[screen, 3])$
(%i3) draw2d(explicit(x^2,x,-2,2), terminal=[screen, 0])$
```

Un fichero gif animado.

```
(%i1) load("draw")$
(%i2) draw(
        delay     = 100,
        file_name = "zzz",
        terminal  = 'animated_gif,
        gr2d(explicit(x^2,x,-1,1)),
        gr2d(explicit(x^3,x,-1,1)),
        gr2d(explicit(x^4,x,-1,1)));
End of animation sequence
(%o2)          [gr2d(explicit), gr2d(explicit), gr2d(explicit)]
```

La opción `delay` sólo se activa en caso de gifs animados; se ignora en cualquier otro caso.

Salida multipágina en formato eps.

```
(%i1) load("draw")$
```

```
(%i2) draw(
        file_name = "parabol",
        terminal  = multipage_eps,
        dimensions = 100*[10,10],
        gr2d(explicit(x^2,x,-1,1)),
        gr3d(explicit(x^2+y^2,x,-1,1,y,-1,1))) $
```

Véanse también `file_name`, `pic_width`, `pic_height` y `delay`.

`title` [Opción gráfica]

Valor por defecto: `""` (cadena vacía)

La opción `title` almacena una cadena con el título de la escena. Por defecto, no se escribe título alguno.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(exp(u),u,-2,2),
             title = "Exponential function")$
```

`transform` [Opción gráfica]

Valor por defecto: `none`

Si `transform` vale `none`, el espacio no sufre transformación alguna y los objetos gráficos se representan tal cual se definen. Si es necesario transformar el espacio, se debe asignar una lista a la opción `transform`. En caso de una escena 2D, la lista toma la forma `[f1(x,y), f2(x,y), x, y]`. En caso de una escena 3D, la lista debe ser de la forma `[f1(x,y,z), f2(x,y,z), f3(x,y,z), x, y, z]`.

Los nombres de las variables definidas en las listas pueden ser diferentes de aquellas utilizadas en las definiciones de los objetos gráficos.

Ejemplos:

Rotación en 2D.

```
(%i1) load("draw")$
(%i2) th : %pi / 4$
(%i3) draw2d(
        color = "#e245f0",
        proportional_axes = 'xy,
        line_width = 8,
        triangle([3,2],[7,2],[5,5]),
        border     = false,
        fill_color = yellow,
        transform  = [cos(th)*x - sin(th)*y,
                      sin(th)*x + cos(th)*y, x, y],
        triangle([3,2],[7,2],[5,5]) )$
```

Traslación en 3D.

```
(%i1) load("draw")$
(%i2) draw3d(
```

```
                color     = "#a02c00",
                explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3),
                transform = [x+10,y+10,z+10,x,y,z],
                color     = blue,
                explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3) )$
```

`transparent` [Opción gráfica]

Valor por defecto: `false`

Cuando `transparent` vale `false`, las regiones internas de los polígonos se rellenan de acuerdo con `fill_color`.

Esta opción afecta a los siguientes objetos gráficos:

- `gr2d`: `polygon`, `rectangle` y `ellipse`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(polygon([[3,2],[7,2],[5,5]]),
             transparent = true,
             color       = blue,
             polygon([[5,2],[9,2],[7,5]]) )$
```

`unit_vectors` [Opción gráfica]

Valor por defecto: `false`

Cuando `unit_vectors` vale `true`, los vectores se dibujan con módulo unidad. Esta opción es útil para representar campos vectoriales. Cuando `unit_vectors` vale `false`, los vectores se dibujan con su longitud original.

Esta opción sólo es relevante para objetos de tipo `vector`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(xrange      = [-1,6],
             yrange      = [-1,6],
             head_length = 0.1,
             vector([0,0],[5,2]),
             unit_vectors = true,
             color        = red,
             vector([0,3],[5,2]))$
```

`user_preamble` [Opción gráfica]

Valor por defecto: `""` (cadena vacía)

Usuarios expertos en Gnuplot pueden hacer uso de esta opción para afinar el comportamiento de Gnuplot escribiendo código que será enviado justo antes de la instrucción `plot` o `splot`.

El valor dado a esta opción debe ser una cadena alfanumérica o una lista de cadenas (una por línea).

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

Se le indica a Gnuplot que dibuje los ejes encima de todos los demás objetos,

```
(%i1) load("draw")$
(%i2) draw2d(
        xaxis =true, xaxis_type=solid,
        yaxis =true, yaxis_type=solid,
        user_preamble="set grid front",
        region(x^2+y^2<1 ,x,-1.5,1.5,y,-1.5,1.5))$
```

`view` [Opción gráfica]

Valor por defecto: `[60,30]`

Un par de ángulos, medidos en grados, indicando la direcciÃ³n del observador en una escena 3D. El primer ángulo es la rotación vertical alrededor del eje x, dentro del intervalo $[0, 360]$. El segundo es la rotación horizontal alrededor del eje z, dentro del intervalo $[0, 360]$.

Dándole a la opción `view` el valor `map`, la dirección del observador se sitúa perpendicularmente al plano-xy.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(view = [170, 50],
             enhanced3d = true,
             explicit(sin(x^2+y^2),x,-2,2,y,-2,2) )$
(%i3) draw3d(view = map,
             enhanced3d = true,
             explicit(sin(x^2+y^2),x,-2,2,y,-2,2) )$
```

`wired_surface` [Opción gráfica]

Valor por defecto: `false`

Indica si las superficies en 3D en modo `enhanced3d` deben mostrar o no la malla que unen los puntos.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(
        enhanced3d    = [sin(x),x,y],
        wired_surface = true,
        explicit(x^2+y^2,x,-1,1,y,-1,1)) $
```

`x_voxel` [Opción gráfica]

Valor por defecto: 10

`x_voxel` es el número de voxels en la dirección x a utilizar por el algoritmo *marching cubes* implementado por el objeto `implicit` tridimensional. También se utiliza como opción del objeto gráfico `region`.

`xaxis` [Opción gráfica]

Valor por defecto: `false`

Si `xaxis` vale `true`, se dibujará el eje x.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             xaxis       = true,
             xaxis_color = blue)$
```

Véanse también `xaxis_width`, `xaxis_type` y `xaxis_color`.

`xaxis_color` [Opción gráfica]

Valor por defecto: `"black"`

`xaxis_color` especifica el color para el eje x. Véase `color` para ver cómo se definen los colores.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             xaxis       = true,
             xaxis_color = red)$
```

Véanse también `xaxis`, `xaxis_width` y `xaxis_type`.

`xaxis_secondary` [Opción gráfica]

Valor por defecto: `false`

Si `xaxis_secondary` vale `true`, los valores de las funciones se pueden representar respecto del eje x secundario, el cual se dibuja en la parte superior de la escena.

Nótese que esta es una opción gráfica local que sólo afecta a objetos 2d.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(
         key   = "Bottom x-axis",
         explicit(x+1,x,1,2),
         color = red,
         key   = "Above x-axis",
         xtics_secondary = true,
         xaxis_secondary = true,
         explicit(x^2,x,-1,1)) $
```

Véanse también `xrange_secondary`, `xtics_secondary`, `xtics_rotate_secondary`, `xtics_axis_secondary` y `xaxis_secondary`.

`xaxis_type` [Opción gráfica]

Valor por defecto: `dots`

`xaxis_type` indica cómo se debe dibujar el eje *x*; valores admisibles son `solid` y `dots`.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             xaxis       = true,
             xaxis_type  = solid)$
```

Véanse también `xaxis`, `xaxis_width` y `xaxis_color`.

`xaxis_width` [Opción gráfica]

Valor por defecto: 1

`xaxis_width` es el ancho del eje *x*. Su valor debe ser un número positivo.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             xaxis       = true,
             xaxis_width = 3)$
```

Véanse también `xaxis`, `xaxis_type` y `xaxis_color`.

`xlabel` [Opción gráfica]

Valor por defecto: `""`

La opción `xlabel` almacena una cadena con la etiqueta del eje *x*. Por defecto, el eje tiene etiqueta `"x"`.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(xlabel = "Time",
             explicit(exp(u),u,-2,2),
             ylabel = "Population")$
```

Véanse también `xlabel_secondary`, `ylabel`, `ylabel_secondary` y `zlabel`.

`xlabel_secondary` [Opción gráfica]

Valor por defecto: `""` (cadena vacía)

La opción `xlabel_secondary` almacena una cadena con la etiqueta del eje *x* secundario. Por defecto, el eje no tiene etiqueta.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(
         xaxis_secondary=true,yaxis_secondary=true,
         xtics_secondary=true,ytics_secondary=true,
         xlabel_secondary="t[s]",
         ylabel_secondary="U[V]",
         explicit(sin(t),t,0,10) )$
```

Véanse también `xlabel`, `ylabel`, `ylabel_secondary` y `zlabel`.

`xrange` [Opción gráfica]

Valor por defecto: `auto`

Cuando `xrange` vale `auto`, el rango de la coordenada x se calcula de forma automática.

Si el usuario quiere especificar un intervalo para x, éste debe expresarse como una lista de Maxima, como en `xrange=[-2, 3]`.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(xrange = [-3,5],
             explicit(x^2,x,-1,1))$
```

Véanse también `yrange` y `zrange`.

`xrange_secondary` [Opción gráfica]

Valor por defecto: `auto`

Cuando `xrange_secondary` vale `auto`, el rango del eje x secundario se calcula de forma automática.

Si el usuario quiere especificar un intervalo para el eje x secundario, éste debe expresarse como una lista de Maxima, como en `xrange_secondary=[-2, 3]`.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Véanse también `xrange`, `yrange`, `zrange` y `yrange_secondary`.

`xtics` [Opción gráfica]

Valor por defecto: `true`

Esta opción gráfica controla la forma en la que se dibujarán las marcas del eje x.

- Cuando a `xtics` se le da el valor *true*, las marcas se dibujarán de forma automática.
- Cuando a `xtics` se le da el valor *false*, no habrá marcas en los ejes.
- Cuando a `xtics` se le da un valor numérico positivo, se interpretará como la distancia entre dos marcas consecutivas.
- Cuando a `xtics` se le da una lista de longitud tres de la forma `[start,incr,end]`, las marcas se dibujarán desde `start` hasta `end` a intervalos de longitud `incr`.

- Cuando a `xtics` se le da un conjunto de números de la forma `{n1, n2, ...}`, las marcas se dibujarán exactamente en los valores `n1`, `n2`, ...
- Cuando a `xtics` se le da un conjunto de pares de la forma `{["label1", n1], ["label2", n2], ...}`, las marcas correspondientes a los valores `n1`, `n2`, ... se etiquetarán con `"label1"`, `"label2"`, ..., respectivamente.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplos:

Marcas desactivadas.

```
(%i1) load("draw")$
(%i2) draw2d(xtics = 'false,
             explicit(x^3,x,-1,1)  )$
```

Marcas cada 1/4 unidades.

```
(%i1) load("draw")$
(%i2) draw2d(xtics = 1/4,
             explicit(x^3,x,-1,1)  )$
```

Marcas desde -3/4 hasta 3/4 en saltos de 1/8.

```
(%i1) load("draw")$
(%i2) draw2d(xtics = [-3/4,1/8,3/4],
             explicit(x^3,x,-1,1)  )$
```

Marcas en los puntos -1/2, -1/4 y 3/4.

```
(%i1) load("draw")$
(%i2) draw2d(xtics = {-1/2,-1/4,3/4},
             explicit(x^3,x,-1,1)  )$
```

Marcas etiquetadas.

```
(%i1) load("draw")$
(%i2) draw2d(xtics = {["High",0.75],["Medium",0],["Low",-0.75]},
             explicit(x^3,x,-1,1)  )$
```

`xtics_axis` [Opción gráfica]

Valor por defecto: `false`

Si `xtics_axis` vale `true`, las marcas y sus etiquetas se dibujan sobre el propio eje x, si vale `false` las marcas se colocan a lo largo del borde del gráfico.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

`xtics_rotate` [Opción gráfica]

Valor por defecto: `false`

Si `xtics_rotate` vale `true`, las marcas del eje x se giran 90 grados.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

`xtics_rotate_secondary` [Opción gráfica]

Valor por defecto: `false`

Si `xtics_rotate_secondary` vale `true`, las marcas del eje x secundario se giran 90 grados.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

`xtics_secondary` [Opción gráfica]

Valor por defecto: `auto`

Esta opción gráfica controla la forma en la que se dibujarán las marcas del eje x secundario.

Véase `xtics` para una descripción completa.

`xtics_secondary_axis` [Opción gráfica]

Valor por defecto: `false`

Si `xtics_secondary_axis` vale `true`, las marcas y sus etiquetas se dibujan sobre el propio eje x secundario, si vale `false` las marcas se colocan a lo largo del borde del gráfico.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

`xu_grid` [Opción gráfica]

Valor por defecto: 30

`xu_grid` es el número de coordenadas de la primera variable (`x` en superficies explcítas y `u` en las paramétricas) para formar la rejilla de puntos muestrales.

Esta opción afecta a los siguientes objetos gráficos:

- `gr3d`: `explicit` y `parametric_surface`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(xu_grid = 10,
             yv_grid = 50,
             explicit(x^2+y^2,x,-3,3,y,-3,3) )$
```

Véase también `yv_grid`.

`xy_file` [Opción gráfica]

Valor por defecto: `""` (cadena vacía)

`xy_file` es el nombre del fichero donde se almacenarán las coordenadas después de hacer clic con el botón del ratón en un punto de la imagen y pulsado la tecla 'x'. Por defecto, las coordenadas no se almacenan.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

`xyplane` [Graphic option]

Valor por defecto: `false`

Coloca el plano-xy en escenas 3D. Si `xyplane` vale `false`, el plano-xy se coloca automáticamente; en cambio, si toma un valor real, el plano-xy intersectará con el eje z a ese nivel. Esta opción no tiene efecto alguno en escenas 2D.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(xyplane = %e-2,
             explicit(x^2+y^2,x,-1,1,y,-1,1))$
```

`y_voxel` [Opción gráfica]

Valor por defecto: 10

`y_voxel` es el número de voxels en la dirección y a utilizar por el algoritmo *marching cubes* implementado por el objeto `implicit` tridimensional. También se utiliza como opción del objeto gráfico `region`.

`yaxis` [Opción gráfica]

Valor por defecto: `false`

Si `yaxis` vale `true`, se dibujará el eje y.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             yaxis       = true,
             yaxis_color = blue)$
```

Véanse también `yaxis_width`, `yaxis_type` y `yaxis_color`.

`yaxis_color` [Opción gráfica]

Valor por defecto: `"black"`

`yaxis_color` especifica el color para el eje y. Véase `color` para ver cómo se definen los colores.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             yaxis       = true,
             yaxis_color = red)$
```

Véanse también `yaxis`, `yaxis_width` y `yaxis_type`.

`yaxis_secondary` [Opción gráfica]

Valor por defecto: `false`

Si `yaxis_secondary` vale `true`, los valores de las funciones se pueden representar respecto del eje y secundario, el cual se dibuja al lado derecho de la escena.

Nótese que esta es una opción gráfica local que sólo afecta a objetos 2d.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(
         explicit(sin(x),x,0,10),
         yaxis_secondary = true,
         ytics_secondary = true,
         color = blue,
         explicit(100*sin(x+0.1)+2,x,0,10));
```

Véanse también `yrange_secondary`, `ytics_secondary`, `ytics_rotate_secondary` y `ytics_axis_secondary`.

`yaxis_type` [Opción gráfica]

Valor por defecto: `dots`

`yaxis_type` indica cómo se debe dibujar el eje y; valores admisibles son `solid` y `dots`.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             yaxis       = true,
             yaxis_type  = solid)$
```

Véanse también `yaxis`, `yaxis_width` y `yaxis_color`.

`yaxis_width` [Opción gráfica]

Valor por defecto: 1

`yaxis_width` es el ancho del eje y. Su valor debe ser un número positivo.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             yaxis       = true,
             yaxis_width = 3)$
```

Véanse también `yaxis`, `yaxis_type` y `yaxis_color`.

`ylabel` [Opción gráfica]

Valor por defecto: `""`

La opción `ylabel` almacena una cadena con la etiqueta del eje y. Por defecto, el eje tiene etiqueta `"y"`.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
```

```
(%i2) draw2d(xlabel = "Time",
             ylabel = "Population",
             explicit(exp(u),u,-2,2) )$
```

Véanse también `xlabel`, `xlabel_secondary`, `ylabel_secondary` y `zlabel`.

`ylabel_secondary` [Opción gráfica]

Valor por defecto: `""` (cadena vacía)

La opción `ylabel_secondary` almacena una cadena con la etiqueta del eje *y* secundario. Por defecto, el eje no tiene etiqueta.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(
        key="current",
        xlabel="t[s]",
        ylabel="I[A]",ylabel_secondary="P[W]",
        explicit(sin(t),t,0,10),
        yaxis_secondary=true,
        ytics_secondary=true,
        color=red,key="Power",
        explicit((sin(t))^2,t,0,10)
      )$
```

Véanse también `xlabel`, `xlabel_secondary`, `ylabel` and `zlabel`.

`yrange` [Opción gráfica]

Valor por defecto: `auto`

Cuando `yrange` vale `auto`, el rango de la coordenada *y* se calcula de forma automática.

Si el usuario quiere especificar un intervalo para *y*, éste debe expresarse como una lista de Maxima, como en `yrange=[-2, 3]`.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(yrange = [-2,3],
             explicit(x^2,x,-1,1),
             xrange = [-3,3])$
```

Véanse también `xrange` y `zrange`.

`yrange_secondary` [Opción gráfica]

Valor por defecto: `auto`

Cuando `yrange_secondary` vale `auto`, el rango del eje *y* secundario se calcula de forma automática.

Si el usuario quiere especificar un intervalo para el eje *y* secundario, éste debe expresarse como una lista de Maxima, como en `yrange_secondary=[-2, 3]`.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(
         explicit(sin(x),x,0,10),
         yaxis_secondary = true,
         ytics_secondary = true,
         yrange = [-3, 3],
         yrange_secondary = [-20, 20],
         color = blue,
         explicit(100*sin(x+0.1)+2,x,0,10)) $
```

Véanse también `xrange`, `yrange` y `zrange`.

`ytics` [Opción gráfica]

Valor por defecto: `true`

Esta opción gráfica controla la forma en la que se dibujarán las marcas del eje *y*.

Véase `xtics` para una descripción completa.

`ytics_axis` [Opción gráfica]

Valor por defecto: `false`

Si `ytics_axis` vale `true`, las marcas y sus etiquetas se dibujan sobre el propio eje *y*, si vale `false` las marcas se colocan a lo largo del borde del gráfico.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

`ytics_rotate` [Opción gráfica]

Valor por defecto: `false`

Si `ytics_rotate` vale `true`, las marcas del eje *y* se giran 90 grados.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

`ytics_rotate_secondary` [Opción gráfica]

Valor por defecto: `false`

Si `ytics_rotate_secondary` vale `true`, las marcas del eje *y* secundario se giran 90 grados.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

`ytics_secondary` [Opción gráfica]

Valor por defecto: `auto`

Esta opción gráfica controla la forma en la que se dibujarán las marcas del eje *y* secundario.

Véase `xtics` para una descripción completa.

`ytics_secondary_axis` [Opción gráfica]

Valor por defecto: `false`

Si `ytics_secondary_axis` vale `true`, las marcas y sus etiquetas se dibujan sobre el propio eje *y* secundario, si vale `false` las marcas se colocan a lo largo del borde del gráfico.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

`yv_grid` [Opción gráfica]

Valor por defecto: 30

`yv_grid` es el número de coordenadas de la segunda variable (`y` en superficies explcítas y `v` en las paramétricas) para formar la rejilla de puntos muestrales.

Esta opción afecta a los siguientes objetos gráficos:

- `gr3d`: `explicit` y `parametric_surface`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(xu_grid = 10,
             yv_grid = 50,
             explicit(x^2+y^2,x,-3,3,y,-3,3) )$
```

Véase también `xu_grid`.

`z_voxel` [Opción gráfica]

Valor por defecto: 10

`z_voxel` es el número de voxels en la dirección z a utilizar por el algoritmo *marching cubes* implementado por el objeto `implicit` tridimensional.

`zaxis` [Opción gráfica]

Valor por defecto: `false`

Si `zaxis` vale `true`, se dibujará el eje *z* en escenas 3D. Esta opción no tiene efecto alguno en escenas 2D.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
             zaxis       = true,
             zaxis_type  = solid,
             zaxis_color = blue)$
```

Véanse también `zaxis_width`, `zaxis_type` y `zaxis_color`.

`zaxis_color` [Opción gráfica]

Valor por defecto: `"black"`

`zaxis_color` especifica el color para el eje *z*. Véase `color` para ver cómo se definen los colores. Esta opción no tiene efecto alguno en escenas 2D.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
             zaxis       = true,
             zaxis_type  = solid,
             zaxis_color = red)$
```

Véanse también `zaxis`, `zaxis_width` y `zaxis_type`.

`zaxis_type` [Opción gráfica]

Valor por defecto: `dots`

`zaxis_type` indica cómo se debe dibujar el eje z; valores admisibles son `solid` y `dots`. Esta opción no tiene efecto alguno en escenas 2D.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
             zaxis       = true,
             zaxis_type  = solid)$
```

Véanse también `zaxis`, `zaxis_width` y `zaxis_color`.

`zaxis_width` [Opción gráfica]

Valor por defecto: 1

`zaxis_width` es el ancho del eje z. Su valor debe ser un número positivo. Esta opción no tiene efecto alguno en escenas 2D.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
             zaxis       = true,
             zaxis_type  = solid,
             zaxis_width = 3)$
```

Véanse también `zaxis`, `zaxis_type` y `zaxis_color`.

`zlabel` [Opción gráfica]

Valor por defecto: `""`

La opción `zlabel` almacena una cadena con la etiqueta del eje z. Por defecto, el eje tiene etiqueta `"z"`.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
```

```
(%i2) draw3d(zlabel = "Z variable",
             ylabel = "Y variable",
             explicit(sin(x^2+y^2),x,-2,2,y,-2,2),
             xlabel = "X variable" )$
```

Véanse también `xlabel` y `ylabel`.

`zrange` [Opción gráfica]

Valor por defecto: `auto`

Cuando `zrange` vale `auto`, el rango de la coordenada *z* se calcula de forma automática.

Si el usuario quiere especificar un intervalo para *z*, éste debe expresarse como una lista de Maxima, como en `zrange=[-2, 3]`.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(yrange = [-3,3],
             zrange = [-2,5],
             explicit(x^2+y^2,x,-1,1,y,-1,1),
             xrange = [-3,3])$
```

Véanse también `xrange` y `yrange`.

`ztics` [Opción gráfica]

Valor por defecto: `true`

Esta opción gráfica controla la forma en la que se dibujarán las marcas del eje *z*.

Véase `xtics` para una descripción completa.

`ztics_axis` [Opción gráfica]

Valor por defecto: `false`

Si `ztics_axis` vale `true`, las marcas y sus etiquetas se dibujan sobre el propio eje *z*, si vale `false` las marcas se colocan a lo largo del borde del gráfico.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

`ztics_rotate` [Opción gráfica]

Valor por defecto: `false`

Si `ztics_rotate` vale `true`, las marcas del eje *z* se giran 90 grados.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

47.2.4 Objetos gráficos

`bars` ([*x1*,*h1*,*w1*], [*x2*,*h2*,*w2*, ...]) [Objeto gráfico]

Dibuja barras verticales en 2D.

2D

`bars ([x1,h1,w1], [x2,h2,w2, ...])` dibuja barras centradas en los valores *x1*, *x2*, ... de alturas *h1*, *h2*, ... y anchos *w1*, *w2*, ...

Este objeto se ve afectado por las siguientes *opciones gráficas*: `key`, `fill_color`, `fill_density` y `line_width`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(
       key          = "Grupo A",
       fill_color   = blue,
       fill_density = 0.2,
       bars([0.8,5,0.4],[1.8,7,0.4],[2.8,-4,0.4]),
       key          = "Grupo B",
       fill_color   = red,
       fill_density = 0.6,
       line_width   = 4,
       bars([1.2,4,0.4],[2.2,-2,0.4],[3.2,5,0.4]),
       xaxis = true);
```

`cylindrical` (*radius*,*z*,*minz*,*maxz*,*azi*,*minazi*,*maxazi*) [Objeto gráfico]

Dibuja funciones 3D definidas en coordenadas cilíndricas.

3D

`cylindrical` (*radius*,*z*,*minz*,*maxz*,*azi*,*minazi*,*maxazi*) dibuja la función *radius*(*z*,*azi*) definida en coordenadas cilíndricas, con la variable z tomando valores desde *minz* hasta *maxz* y el *azimut azi* tomando valores desde *minazi* hasta *maxazi*.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `xu_grid`, `yv_grid`, `line_type`, `key`, `wired_surface`, `enhanced3d` y `color`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(cylindrical(1,z,-2,2,az,0,2*%pi))$
```

`elevation_grid` (*mat*,*x0*,*y0*,*width*,*height*) [Objeto gráfico]

Dibuja la matriz *mat* en 3D. Los valores z se toman de *mat*, las abscisas van desde *x0* hasta *x0* + *width* y las ordenadas desde *y0* hasta *y0* + *height*. El elemento $a(1,1)$ se proyecta sobre el punto $(x0, y0 + height)$, $a(1,n)$ sobre $(x0 + width, y0 + height)$, $a(m,1)$ sobre $(x0, y0)$ y $a(m,n)$ sobre $(x0 + width, y0)$.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `line_type`, `line_width`, `key`, `wired_surface`, `enhanced3d` y `color`.

En versiones antiguas de Maxima, `elevation_grid` se llamaba `mesh`. Véase también `mesh`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) m: apply(
            matrix,
            makelist(makelist(random(10.0),k,1,30),i,1,20)) $
(%i3) draw3d(
         color = blue,
```

```
                elevation_grid(m,0,0,3,2),
                xlabel = "x",
                ylabel = "y",
                surface_hide = true);
```

`ellipse` (*xc*, *yc*, *a*, *b*, *ang1*, *ang2*) [Objeto gráfico]

Dibuja elipses y círculos en 2D.

2D

`ellipse` (*xc*, *yc*, *a*, *b*, *ang1*, *ang2*) dibuja una elipse de centro [*xc*, *yc*] con semiejes horizontal y vertical *a* y *b*, respectivamente, comenzando en el ángulo *ang1* y trazando un arco de amplitud igual al ángulo *ang2*.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `nticks`, `transparent`, `fill_color`, `border`, `line_width`, `line_type`, `key` y `color`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(transparent = false,
             fill_color  = red,
             color       = gray30,
             transparent = false,
             line_width  = 5,
             ellipse(0,6,3,2,270,-270),
             /* center (x,y), a, b, start & end in degrees */
             transparent = true,
             color       = blue,
             line_width  = 3,
             ellipse(2.5,6,2,3,30,-90),
             xrange      = [-3,6],
             yrange      = [2,9] )$
```

`errors` ([*x1*,*x2*,...], [*y1*,*y2*,...]) [Objeto gráfico]

Dibuja puntos con barras de error asociadas horizontales, verticales o de ambos tipos, según sea el valor de la opción `error_type`.

2D

Si `error_type=x`, los argumentos a `errors` deben ser de la forma `[x,y,xdelta]` o `[x,y,xlow,xhigh]`. Si `error_type=y`, los argumentos deben ser del tipo `[x,y,ydelta]` o `[x,y,ylow,yhigh]`. Si `error_type=xy` o `error_type=boxes`, los argumentos deben ser de la forma `[x,y,xdelta,ydelta]` o `[x,y,xlow,xhigh,ylow,yhigh]`.

Véase también `error_type`.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `error_type`, `points_joined`, `line_width`, `key`, `line_type`, `color`, `fill_density`, `xaxis_secondary` y `yaxis_secondary`.

La opción `fill_density` solo es relevante cuando `error_type=boxes`.

Ejemplos:

Barras de error horizontales.

```
(%i1) load("draw")$
(%i2) draw2d(
        error_type = y,
        errors([[1,2,1], [3,5,3], [10,3,1], [17,6,2]]))$
```

Barras de error horizontales y verticales.

```
(%i1) load("draw")$
(%i2) draw2d(
        error_type = xy,
        points_joined = true,
        color = blue,
        errors([[1,2,1,2], [3,5,2,1], [10,3,1,1], [17,6,1/2,2]]));
```

`explicit` (*fcn*,*var*,*minval*,*maxval*) [Objeto gráfico]
`explicit` (*fcn*,*var1*,*minval1*,*maxval1*,*var2*,*minval2*,*maxval2*) [Objeto gráfico]

Dibuja funciones explícitas en 2D y 3D.

2D

`explicit` (*fcn*,*var*,*minval*,*maxval*) dibuja la función explícita *fcn*, con la variable *var* tomando valores desde *minval* hasta *maxval*.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `nticks`, `adapt_depth`, `draw_realpart`, `line_width`, `line_type`, `key`, `filled_func`, `fill_color` y `color`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(line_width = 3,
             color      = blue,
             explicit(x^2,x,-3,3) )$
(%i3) draw2d(fill_color  = brown,
             filled_func = true,
             explicit(x^2,x,-3,3) )$
```

3D

`explicit` (*fcn*,*var1*,*minval1*,*maxval1*,*var2*,*minval2*,*maxval2*) dibuja la función explícita *fcn*, con la variable `var1` tomando valores desde *minval1* hasta *maxval1* y la variable *var2* tomando valores desde *minval2* hasta *maxval2*.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `draw_realpart`, `xu_grid`, `yv_grid`, `line_type`, `line_width`, `key`, `wired_surface`, `enhanced3d` y `color`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(key   = "Gauss",
             color = "#a02c00",
             explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3),
             yv_grid     = 10,
             color = blue,
             key   = "Plane",
             explicit(x+y,x,-5,5,y,-5,5),
```

```
surface_hide = true)$
```

Véase también `filled_func` para el relleno de curvas.

`image (`*`im`*`,`*`x0`*`,`*`y0`*`,`*`width`*`,`*`height`*`)` [Objeto gráfico]

Reproduce una imagen en 2D.

2D

`image (`*`im`*`,`*`x0`*`,`*`y0`*`,`*`width`*`,`*`height`*`)`: dibuja la imagen *im* en la región rectangular desde el vértice `(`*`x0`*`,`*`y0`*`)` hasta el `(x0+`*`width`*`,y0+`*`height`*`)` del plano real. El argumento *im* debe ser una matriz de números reales, una matriz de vectores de longitud tres o un objeto de tipo `picture`.

Si *im* es una matriz de números reales, los valores de los píxeles se interpretan según indique la opción gráfica `palette`, que es un vector de longitud tres con sus componentes tomando valores enteros en el rango desde -36 a +36; cada valor es un índice para seleccionar una fórmula que transforma los niveles numéricos en las componentes cromáticas rojo, verde y azul:

```
 0: 0               1: 0.5           2: 1
 3: x               4: x^2           5: x^3
 6: x^4             7: sqrt(x)       8: sqrt(sqrt(x))
 9: sin(90x)       10: cos(90x)     11: |x-0.5|
12: (2x-1)^2       13: sin(180x)    14: |cos(180x)|
15: sin(360x)      16: cos(360x)    17: |sin(360x)|
18: |cos(360x)|    19: |sin(720x)|  20: |cos(720x)|
21: 3x             22: 3x-1         23: 3x-2
24: |3x-1|         25: |3x-2|       26: (3x-1)/2
27: (3x-2)/2       28: |(3x-1)/2|   29: |(3x-2)/2|
30: x/0.32-0.78125 31: 2*x-0.84
32: 4x;1;-2x+1.84;x/0.08-11.5
33: |2*x - 0.5|    34: 2*x          35: 2*x - 0.5
36: 2*x - 1
```

los números negativos se interpretan como colores invertidos de las componentes cromáticas.

`palette = gray` y `palette = color` son atajos para `palette = [3,3,3]` y `palette = [7,5,15]`, respectivamente.

Si *im* es una matriz de vectores de longitud tres, éstos se interpretarán como las componentes cromáticas rojo, verde y azul.

Ejemplos:

Si *im* es una matriz de números reales, los valores de los píxeles se interpretan según indique la opción gráfica `palette`.

```
(%i1) load("draw")$
(%i2) im: apply(
           'matrix,
            makelist(makelist(random(200),i,1,30),i,1,30))$
(%i3) /* palette = color, default */
      draw2d(image(im,0,0,30,30))$
(%i4) draw2d(palette = gray, image(im,0,0,30,30))$
```

```
(%i5) draw2d(palette = [15,20,-4],
             colorbox=false,
             image(im,0,0,30,30))$
```

Véase también `colorbox`.

Si *im* es una matriz de vectores de longitud tres, éstos se interpretarán como las componentes cromáticas rojo, verde y azul.

```
(%i1) load("draw")$
(%i2) im: apply(
            'matrix,
             makelist(
               makelist([random(300),
                         random(300),
                         random(300)],i,1,30),i,1,30))$
(%i3) draw2d(image(im,0,0,30,30))$
```

El paquete `draw` carga automáticamente el paquete `picture`. En este ejemplo, una imagen de niveles se define a mano, reproduciéndola a continuación.

```
(%i1) load("draw")$
(%i2) im: make_level_picture([45,87,2,134,204,16],3,2);
(%o2)       picture(level, 3, 2, {Array:  #(45 87 2 134 204 16)})
(%i3) /* default color palette */
      draw2d(image(im,0,0,30,30))$
(%i4) /* gray palette */
      draw2d(palette = gray,
             image(im,0,0,30,30))$
```

Se lee un fichero xpm y se reproduce.

```
(%i1) load("draw")$
(%i2) im: read_xpm("myfile.xpm")$
(%i3) draw2d(image(im,0,0,10,7))$
```

Véanse también `make_level_picture`, `make_rgb_picture` y `read_xpm`.

En http://www.telefonica.net/web2/biomates/maxima/gpdraw/image se encuentran ejemplos más elaborados.

`implicit` (*fcn*,*x*,*xmin*,*xmax*,*y*,*ymin*,*ymax*) [Objeto gráfico]
`implicit` (*fcn*,*x*,*xmin*,*xmax*,*y*,*ymin*,*ymax*,*z*,*zmin*,*zmax*) [Objeto gráfico]

Dibuja funciones implícitas en 2D y 3D.

2D

`implicit` (*fcn*,*x*,*xmin*,*xmax*,*y*,*ymin*,*ymax*) dibuja la función implícita *fcn*, con la variable `x` tomando valores desde *xmin* hasta *xmax*, y la variable *y* tomando valores desde *ymin* hasta *ymax*.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `ip_grid`, `ip_grid_in`, `line_width`, `line_type`, `key` y `color`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(terminal  = eps,
```

```
                grid      = true,
                line_type = solid,
                key       = "y^2=x^3-2*x+1",
                implicit(y^2=x^3-2*x+1, x, -4,4, y, -4,4),
                line_type = dots,
                key       = "x^3+y^3 = 3*x*y^2-x-1",
                implicit(x^3+y^3 = 3*x*y^2-x-1, x,-4,4, y,-4,4),
                title     = "Two implicit functions" )$
```

3D

`implicit (`*`fcn`*`,`*`x`*`,`*`xmin`*`,`*`xmax`*`, `*`y`*`,`*`ymin`*`,`*`ymax`*`, `*`z`*`,`*`zmin`*`,`*`zmax`*`)` dibuja la función implícita *fcn*, con la variable *x* tomando valores desde *xmin* hasta *xmax*, la variable *y* tomando valores desde *ymin* hasta *ymax* y la variable *z* tomando valores desde *zmin* hasta *zmax*. Este objeto está programado con el algoritmo *marching cubes*.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `x_voxel`, `y_voxel`, `z_voxel`, `line_width`, `line_type`, `key`, `wired_surface`, `enhanced3d` y `color`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(
        color=blue,
        implicit((x^2+y^2+z^2-1)*(x^2+(y-1.5)^2+z^2-0.5)=0.015,
                 x,-1,1,y,-1.2,2.3,z,-1,1),
        surface_hide=true);
```

`label (`[*`string`*`,`*`x`*`,`*`y`*`]`,...) [Objeto gráfico]
`label (`[*`string`*`,`*`x`*`,`*`y`*`,`*`z`*`]`,...) [Objeto gráfico]

Escribe etiquetas en 2D y 3D.

Las etiquetas coloreadas sólo trabajan con Gnuplot 4.3. Este es un fallo conocido del paquete `draw`.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `label_alignment`, `label_orientation` y `color`.

2D

`label([`*`string`*`,`*`x`*`,`*`y`*`])` escribe la cadena de caracteres *string* en el punto `[`*`x`*`,`*`y`*`]`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(yrange = [0.1,1.4],
             color = red,
             label(["Label in red",0,0.3]),
             color = "#0000ff",
             label(["Label in blue",0,0.6]),
             color = light_blue,
             label(["Label in light-blue",0,0.9],
                   ["Another light-blue",0,1.2])  )$
```

3D

`label([`*`string`*`,`*`x`*`,`*`y`*`,`*`z`*`])` escribe la cadena de caracteres *string* en el punto `[`*`x`*`,`*`y`*`,`*`z`*`]`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(explicit(exp(sin(x)+cos(x^2)),x,-3,3,y,-3,3),
             color = red,
             label(["UP 1",-2,0,3], ["UP 2",1.5,0,4]),
             color = blue,
             label(["DOWN 1",2,0,-3]) )$
```

`mesh` (*fila_1*,*fila_2*,...) [Objeto gráfico]

Dibuja un enrejado cuadrangular en 3D.

3D

El argumento *fila_i* es una lista de *n* puntos en 3D de la forma `[[x_i1,y_i1,z_i1], ...,[x_in,y_in,z_in]]`, siendo todas las filas de igual longitud. Todos estos puntos definen una superficie arbitraria en 3D. En cierto sentido, se trata de una generalización del objeto `elevation_grid`.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `line_type`, `line_width`, `color`, `key`, `wired_surface`, `enhanced3d` y `transform`.

Ejemplos:

Un sencillo ejemplo.

```
(%i1) load("draw")$
(%i2) draw3d(
         mesh([[1,1,3],   [7,3,1],[12,-2,4],[15,0,5]],
              [[2,7,8],   [4,3,1],[10,5,8], [12,7,1]],
              [[-2,11,10],[6,9,5],[6,15,1], [20,15,2]])) $
```

Dibujando un triángulo en 3D.

```
(%i1) load("draw")$
(%i2) draw3d(
        line_width = 2,
        mesh([[1,0,0],[0,1,0]],
             [[0,0,1],[0,0,1]])) $
```

Dos cuadriláteros.

```
(%i1) load("draw")$
(%i2) draw3d(
        surface_hide = true,
        line_width   = 3,
        color = red,
        mesh([[0,0,0], [0,1,0]],
             [[2,0,2], [2,2,2]]),
        color = blue,
        mesh([[0,0,2], [0,1,2]],
             [[2,0,4], [2,2,4]])) $
```

`parametric` (*xfun*,*yfun*,*par*,*parmin*,*parmax*) [Objeto gráfico]
`parametric` (*xfun*,*yfun*,*zfun*,*par*,*parmin*,*parmax*) [Objeto gráfico]

Dibuja funciones paramétricas en 2D y 3D.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `nticks`, `line_width`, `line_type`, `key`, `color` y `enhanced3d`.

2D

`parametric (xfun,yfun,par,parmin,parmax)` dibuja la función paramétrica `[xfun,yfun]`, con el parámetro *par* tomando valores desde *parmin* hasta *parmax*.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(exp(x),x,-1,3),
             color = red,
             key   = "This is the parametric one!!",
             parametric(2*cos(rrr),rrr^2,rrr,0,2*%pi))$
```

3D

`parametric (xfun,yfun,zfun,par,parmin,parmax)` dibuja la curva paramétrica `[xfun,yfun,zfun]`, con el parámetro *par* tomando valores desde *parmin* hasta *parmax*.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(explicit(exp(sin(x)+cos(x^2)),x,-3,3,y,-3,3),
             color = royalblue,
             parametric(cos(5*u)^2,sin(7*u),u-2,u,0,2),
             color      = turquoise,
             line_width = 2,
             parametric(t^2,sin(t),2+t,t,0,2),
             surface_hide = true,
             title = "Surface & curves" )$
```

`parametric_surface` [Objeto gráfico]
`(xfun,yfun,zfun,par1,par1min,par1max,par2,par2min,par2max)`

Dibuja superficies paramétricas en 3D.

3D

`parametric_surface (xfun,yfun,zfun,par1,par1min,par1max,par2,par2min,par2max)` dibuja la superficie paramétrica `[xfun,yfun,zfun]`, con el parámetro *par1* tomando valores desde *par1min* hasta *par1max* y el parámetro *par2* tomando valores desde *par2min* hasta *par2max*.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `draw_realpart`, `xu_grid`, `yv_grid`, `line_type`, `line_width`, `key`, `wired_surface`, `enhanced3d` y `color`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(title          = "Sea shell",
             xu_grid        = 100,
             yv_grid        = 25,
             view           = [100,20],
             surface_hide   = true,
             parametric_surface(0.5*u*cos(u)*(cos(v)+1),
```

```
                                    0.5*u*sin(u)*(cos(v)+1),
                                    u*sin(v) - ((u+3)/8*%pi)^2 - 20,
                                    u, 0, 13*%pi, v, -%pi, %pi) )$
```

`points` ([[*x1*,*y1*], [*x2*,*y2*],...]) [Objeto gráfico]
`points` ([*x1*,*x2*,...], [*y1*,*y2*,...]) [Objeto gráfico]
`points` ([*y1*,*y2*,...]) [Objeto gráfico]
`points` ([[*x1*,*y1*,*z1*], [*x2*,*y2*,*z2*],...]) [Objeto gráfico]
`points` ([*x1*,*x2*,...], [*y1*,*y2*,...], [*z1*,*z2*,...]) [Objeto gráfico]
`points` (*matrix*) [Objeto gráfico]
`points` (*1d_y_array*) [Objeto gráfico]
`points` (*1d_x_array*, *1d_y_array*) [Objeto gráfico]
`points` (*1d_x_array*, *1d_y_array*, *1d_z_array*) [Objeto gráfico]
`points` (*2d_xy_array*) [Objeto gráfico]
`points` (*2d_xyz_array*) [Objeto gráfico]

Dibuja puntos en 2D y 3D.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `point_size`, `point_type`, `points_joined`, `line_width`, `key`, `line_type` y `color`. En modo 3D también se ve afectado por `enhanced3d`.

2D

`points ([x1,y1], [x2,y2],...])` o `points ([x1,x2,...], [y1,y2,...])` dibuja los puntos `[x1,y1]`, `[x2,y2]`, etc. Si no se dan las abscisas, éstas se asignan automáticamente a enteros positivos consecutivos, de forma que `points([y1,y2,...])` dibuja los puntos `[1,y1]`, `[2,y2]`, etc. Si *matrix* es una matriz de dos columnas o de dos filas, `points (matrix)` dibuja los puntos asociados.

Si *1d_y_array* es un array lisp de números en 1D, `points (1d_y_array)` los dibujará asignando las abscisas a números enteros consecutivos. `points (1d_x_array, 1d_y_array)` dibuja los puntos cuyas coordenadas se toman de los dos arrays pasados como argumentos. Si *2d_xy_array* es un array lisp 2D de dos filas, o de dos columnas, `points (2d_xy_array)` dibuja los correspondientes puntos del plano.

Ejemplos:

Dos tipos de argumentos para `points`, una lista de pares ordenados y dos listas con las coordenadas separadas.

```
(%i1) load("draw")$
(%i2) draw2d(
        key = "Small points",
        points(makelist([random(20),random(50)],k,1,10)),
        point_type    = circle,
        point_size    = 3,
        points_joined = true,
        key           = "Great points",
        points(makelist(k,k,1,20),makelist(random(30),k,1,20)),
        point_type    = filled_down_triangle,
        key           = "Automatic abscissas",
        color         = red,
        points([2,12,8]))$
```

Dibujando impulsos.

```
(%i1) load("draw")$
(%i2) draw2d(
        points_joined = impulses,
        line_width    = 2,
        color         = red,
        points(makelist([random(20),random(50)],k,1,10)))$
```

Array con ordenadas.

```
(%i1) load("draw")$
(%i2) a: make_array (flonum, 100) $
(%i3) for i:0 thru 99 do a[i]: random(1.0) $
(%i4) draw2d(points(a)) $
```

Dos arrays con coordenadas separadas.

```
(%i1) load("draw")$
(%i2) x: make_array (flonum, 100) $
(%i3) y: make_array (fixnum, 100) $
(%i4) for i:0 thru 99 do (
        x[i]: float(i/100),
        y[i]: random(10) ) $
(%i5) draw2d(points(x, y)) $
```

Un array 2D de dos columnas.

```
(%i1) load("draw")$
(%i2) xy: make_array(flonum, 100, 2) $
(%i3) for i:0 thru 99 do (
        xy[i, 0]: float(i/100),
        xy[i, 1]: random(10) ) $
(%i4) draw2d(points(xy)) $
```

Dibujando un array rellenado con la función `read_array`.

```
(%i1) load("draw")$
(%i2) a: make_array(flonum,100) $
(%i3) read_array (file_search ("pidigits.data"), a) $
(%i4) draw2d(points(a)) $
```

3D

`points ([[`*x1*`,`*y1*`,`*z1*`], [`*x2*`,`*y2*`,`*z2*`],...])` o `points ([`*x1*`,`*x2*`,...], [`*y1*`,`*y2*`,...], [`*z1*`,`*z2*`,...])` dibuja los puntos `[`*x1*`,`*y1*`,`*z1*`]`, `[`*x2*`,`*y2*`,`*z2*`]`, etc. Si *matrix* es una matriz de tres columnas o de tres filas, `points (`*matrix*`)` dibuja los puntos asociados. Si *matrix* es una matriz columna o fila, las abscisas se asignan automáticamente.

En caso de que los argumentos sean arrays lisp, `points (`*1d_x_array*`, `*1d_y_array*`, `*1d_z_array*`)` toma las coordenadas de los tres arrays unidimensionales. Si *2d_xyz_array* es un array 2D de tres columnas, o de tres filas, entonces `points (`*2d_xyz_array*`)` dibuja los puntos correspondientes.

Ejemplos:

Una muestra tridimensional,

```
(%i1) load("draw")$
```

```
(%i2) load ("numericalio")$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) draw3d(title = "Daily average wind speeds",
             point_size = 2,
             points(args(submatrix (s2, 4, 5))) )$
```

Dos muestras tridimensionales,

```
(%i1) load("draw")$
(%i2) load ("numericalio")$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) draw3d(
         title = "Daily average wind speeds. Two data sets",
         point_size = 2,
         key        = "Sample from stations 1, 2 and 3",
         points(args(submatrix (s2, 4, 5))),
         point_type = 4,
         key        = "Sample from stations 1, 4 and 5",
         points(args(submatrix (s2, 2, 3))) )$
```

Arrays unidimensionales,

```
(%i1) load("draw")$
(%i2) x: make_array (fixnum, 10) $
(%i3) y: make_array (fixnum, 10) $
(%i4) z: make_array (fixnum, 10) $
(%i5) for i:0 thru 9 do (
        x[i]: random(10),
        y[i]: random(10),
        z[i]: random(10) ) $
(%i6) draw3d(points(x,y,z)) $
```

Array bidimensional coloreado,

```
(%i1) load("draw")$
(%i2) xyz: make_array(fixnum, 10, 3) $
(%i3) for i:0 thru 9 do (
        xyz[i, 0]: random(10),
        xyz[i, 1]: random(10),
        xyz[i, 2]: random(10) ) $
(%i4) draw3d(
         enhanced3d = true,
         points_joined = true,
         points(xyz)) $
```

Números de colores especificados explícitamente por el usuario.

```
(%i1) load("draw")$
(%i2) pts: makelist([t,t^2,cos(t)], t, 0, 15)$
(%i3) col_num: makelist(k, k, 1, length(pts))$
(%i4) draw3d(
        enhanced3d = ['part(col_num,k),k],
        point_size = 3,
```

```
                point_type = filled_circle,
                points(pts))$
```

polar (*radius*,*ang*,*minang*,*maxang*) [Objeto gráfico]

Dibuja funciones 2D definidas en coordenadas polares.

2D

polar (*radius*,*ang*,*minang*,*maxang*) dibuja la función *radius*(*ang*) definida en coordenadas polares, con la variable *ang* tomando valores desde *minang* hasta *maxang*.

Este objeto se ve afectado por las siguientes *opciones gráficas*: nticks, line_width, line_type, key y color.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(user_preamble = "set grid polar",
             nticks        = 200,
             xrange        = [-5,5],
             yrange        = [-5,5],
             color         = blue,
             line_width    = 3,
             title         = "Hyperbolic Spiral",
             polar(10/theta,theta,1,10*%pi) )$
```

polygon ([[*x1*,*y1*], [*x2*,*y2*],...]) [Objeto gráfico]
polygon ([*x1*,*x2*,...], [*y1*,*y2*,...]) [Objeto gráfico]

Dibuja polígonos en 2D.

2D

polygon ([[*x1*,*y1*], [*x2*,*y2*],...]) o polygon ([*x1*,*x2*,...], [*y1*,*y2*,...]): dibuja en el plano un polígono de vértices [*x1*,*y1*], [*x2*,*y2*], etc..

Este objeto se ve afectado por las siguientes *opciones gráficas*: transparent, fill_color, border, line_width, key, line_type y color.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(color      = "#e245f0",
             line_width = 8,
             polygon([[3,2],[7,2],[5,5]]),
             border      = false,
             fill_color  = yellow,
             polygon([[5,2],[9,2],[7,5]]) )$
```

quadrilateral (*point_1*, *point_2*, *point_3*, *point_4*) [Objeto gráfico]

Dibuja un cuadrilátero.

2D

quadrilateral ([*x1*,*y1*], [*x2*,*y2*], [*x3*,*y3*], [*x4*,*y4*]) dibuja un cuadrilátero de vértices [*x1*,*y1*], [*x2*,*y2*], [*x3*,*y3*] y [*x4*,*y4*].

Este objeto se ve afectado por las siguientes *opciones gráficas*: transparent, fill_color, border, line_width, key, xaxis_secondary, yaxis_secondary, line_type, transform y color.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(
        quadrilateral([1,1],[2,2],[3,-1],[2,-2]))$
```

3D

`quadrilateral` (`[`*x1*`,`*y1*`,`*z1*`]`, `[`*x2*`,`*y2*`,`*z2*`]`, `[`*x3*`,`*y3*`,`*z3*`]`, `[`*x4*`,`*y4*`,`*z4*`]`) dibuja un cuadrilátero de vértices `[`*x1*`,`*y1*`,`*z1*`]`, `[`*x2*`,`*y2*`,`*z2*`]`, `[`*x3*`,`*y3*`,`*z3*`]` y `[`*x4*`,`*y4*`,`*z4*`]`.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `line_type`, `line_width`, `color`, `key`, `enhanced3d` y `transform`.

`rectangle` (`[`*x1*`,`*y1*`]`, `[`*x2*`,`*y2*`]`) [Objeto gráfico]

Dibuja rectángulos en 2D.

2D

`rectangle (`[*x1*`,`*y1*`]`, `[`*x2*`,`*y2*`])` dibuja un rectángulo de vértices opuestos `[`*x1*`,`*y1*`]` y `[`*x2*`,`*y2*`]`.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `transparent`, `fill_color`, `border`, `line_width`, `key`, `line_type` y `color`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(fill_color  = red,
             line_width  = 6,
             line_type   = dots,
             transparent = false,
             fill_color  = blue,
             rectangle([-2,-2],[8,-1]), /* opposite vertices */
             transparent = true,
             line_type   = solid,
             line_width  = 1,
             rectangle([9,4],[2,-1.5]),
             xrange      = [-3,10],
             yrange      = [-3,4.5] )$
```

`region` (*expr*`,`*var1*`,`*minval1*`,`*maxval1*`,`*var2*`,`*minval2*`,`*maxval2*) [Objeto gráfico]

Dibuja una región del plano definida por desigualdades.

2D *expr* es una expresión formada por desigualdades y los operadores lógicos `and`, `or` y `not`. La región está acotada por el rectángulo definido por [*minval1*, *maxval1*] y [*minval2*, *maxval2*].

Este objeto se ve afectado por las siguientes *opciones gráficas*: `fill_color`, `key`, `x_voxel` y `y_voxel`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(
        x_voxel = 30,
        y_voxel = 30,
        region(x^2+y^2<1 and x^2+y^2 > 1/2,
               x, -1.5, 1.5, y, -1.5, 1.5));
```

spherical (*radius*,*azi*,*minazi*,*maxazi*,*zen*,*minzen*,*maxzen*) [Objeto gráfico]

Dibuja funciones 3D definidas en coordenadas esféricas.

3D

spherical (*radius*,*azi*,*minazi*,*maxazi*,*zen*,*minzen*,*maxzen*) dibuja la función *radius*(*azi*,*zen*) definida en coordenadas esféricas, con el *azimut azi* tomando valores desde *minazi* hasta *maxazi* y el *zenit zen* tomando valores desde *minzen* hasta *maxzen*.

Este objeto se ve afectado por las siguientes *opciones gráficas*: xu_grid, yv_grid, line_type, key, wired_surface, enhanced3d y color.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(spherical(1,a,0,2*%pi,z,0,%pi))$
```

triangle (*punto_1*, *punto_2*, *punto_3*) [Objeto gráfico]

Dibuja un triángulo.

2D

triangle ([*x1*,*y1*], [*x2*,*y2*], [*x3*,*y3*]) dibuja un triángulo de vértices [*x1*,*y1*], [*x2*,*y2*] y [*x3*,*y3*].

Este objeto se ve afectado por las siguientes *opciones gráficas*: transparent, fill_color, border, line_width, key, xaxis_secondary, yaxis_secondary, line_type, transform y color.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(
        triangle([1,1],[2,2],[3,-1]))$
```

3D

triangle ([*x1*,*y1*,*z1*], [*x2*,*y2*,*z2*], [*x3*,*y3*,*z3*]) dibuja un triángulo de vértices [*x1*,*y1*,*z1*], [*x2*,*y2*,*z2*] y [*x3*,*y3*,*z3*].

Este objeto se ve afectado por las siguientes *opciones gráficas*: line_type, line_width, color, key, enhanced3d y transform.

tube (*xfun*,*yfun*,*zfun*,*rfun*,*p*,*pmin*,*pmax*) [Objeto gráfico]

Dibuja un tubo en 3D de diámetro variable.

3D

[*xfun*,*yfun*,*zfun*] es la curva paramétrica de parámetro *p*, el cual toma valores entre *pmin* y *pmax*. Se colocan círculos de radio *rfun* con sus centros sobre la curva paramétrica y perpendiculares a ella.

Este objeto se ve afectado por las siguientes *opciones gráficas*: xu_grid, yv_grid, line_type, line_width, key, wired_surface, enhanced3d, color y capping.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(
        enhanced3d = true,
```

```
                xu_grid    = 50,
                tube(cos(a), a, 0, cos(a/10)^2,
                     a, 0, 4*%pi) )$
```

`vector` ([*x*,*y*], [*dx*,*dy*]) [Objeto gráfico]
`vector` ([*x*,*y*,*z*], [*dx*,*dy*,*dz*]) [Objeto gráfico]

Dibuja vectores en 2D y 3D.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `head_both`, `head_length`, `head_angle`, `head_type`, `line_width`, `line_type`, `key` y `color`.

2D

`vector ([x,y], [dx,dy])` dibuja el vector `[dx,dy]` con origen en `[x,y]`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw2d(xrange      = [0,12],
             yrange      = [0,10],
             head_length = 1,
             vector([0,1],[5,5]), /* default type */
             head_type = 'empty,
             vector([3,1],[5,5]),
             head_both = true,
             head_type = 'nofilled,
             line_type = dots,
             vector([6,1],[5,5]))$
```

3D

`vector([x,y,z], [dx,dy,dz])` dibuja el vector `[dx,dy,dz]` con origen en `[x,y,z]`.

Ejemplo:

```
(%i1) load("draw")$
(%i2) draw3d(color = cyan,
             vector([0,0,0],[1,1,1]/sqrt(3)),
             vector([0,0,0],[1,-1,0]/sqrt(2)),
             vector([0,0,0],[1,1,-2]/sqrt(6)) )$
```

47.3 Funciones y variables para picture

`get_pixel` (*pic*,*x*,*y*) [Función]

Devuelve el pixel de la imagen *pic*. Las coordenadas *x* e *y* van desde 0 hasta `ancho-1` y `alto-1`, respectivamente.

`make_level_picture` (*data*) [Función]
`make_level_picture` (*data*,*width*,*height*) [Función]

Devuelve un objeto `picture` consistente en una imagen de niveles. `make_level_picture (data)` construye el objeto `picture` a partir de la matriz *data*. `make_level_picture (data,width,height)` construye el objeto a partir de una lista de números, en cuyo caso deben indicarse el ancho *width* y la altura *height* en píxeles.

El objeto `picture` devuelto contiene los siguientes cuatro elemento:

1. el símbolo `level`

2. anchura de la imagen
3. altura de la imagen
4. un array de enteros con los valores de los píxeles entre 0 y 255. El argumento *data* debe contener sólo números entre 0 y 255; los cantidades negativas se transforman en ceros y las que son mayores de 255 se igualan a este número.

Ejemplo:

Imagen de niveles a partir de una matriz.

```
(%i1) load("draw")$
(%i2) make_level_picture(matrix([3,2,5],[7,-9,3000]));
(%o2)         picture(level, 3, 2, {Array:  #(3 2 5 7 0 255)})
```

Imagen de niveles a partir de una lista numérica.

```
(%i1) load("draw")$
(%i2) make_level_picture([-2,0,54,%pi],2,2);
(%o2)           picture(level, 2, 2, {Array:  #(0 0 54 3)})
```

`make_rgb_picture` (*redlevel*,*greenlevel*,*bluelevel*) [Función]

Devuelve un objeto *picture* conteniendo una imagen en color (RGB). Los tres argumentos deben ser imágenes de niveles, para el rojo (R), verde (G) y azul (B).

El objeto *picture* devuelto contiene los siguientes cuatro elemento:

1. el símbolo `rgb`
2. anchura de la imagen
3. altura de la imagen
4. un array de enteros de *3*ancho*alto* con los valores de los píxeles entre 0 y 255. Cada valor de pixel se representa en el array con tres números consecutivos (rojo, verde, azul).

Ejemplo:

```
(%i1) load("draw")$
(%i2) red: make_level_picture(matrix([3,2],[7,260]));
(%o2)           picture(level, 2, 2, {Array:  #(3 2 7 255)})
(%i3) green: make_level_picture(matrix([54,23],[73,-9]));
(%o3)           picture(level, 2, 2, {Array:  #(54 23 73 0)})
(%i4) blue: make_level_picture(matrix([123,82],[45,32.5698]));
(%o4)          picture(level, 2, 2, {Array:  #(123 82 45 33)})
(%i5) make_rgb_picture(red,green,blue);
(%o5) picture(rgb, 2, 2,
              {Array:  #(3 54 123 2 23 82 7 73 45 255 0 33)})
```

`negative_picture` (*pic*) [Función]

Devuelve el negativo de la imagen, sea ésta de tipo nivel (*level*) o color (*rgb*).

`picture_equalp` (*x*,*y*) [Función]

Devuelve `true` si los dos argumentos son imágenes idénticas, o `false` en caso contrario.

`picturep` (*x*) [Función]

Devuelve `true` si el argumento es una imagen bien formada, o `false` en caso contrario.

`read_xpm` (*xpm_file*) [Función]
Lee el fichero gráfico en formato xpm y devuelve un objeto `picture`.

`rgb2level` (*pic*) [Función]
Transforma una imagen en color *rgb* a otra de niveles *level* promediando los niveles.

`take_channel` (*im*,*color*) [Función]
Si el argumento *color* es `red`, `green` o `blue`, la función `take_channel` devuelve el canal de color correspondiente de la imagen *im*.

Ejemplo:

```
(%i1) load("draw")$
(%i2) red: make_level_picture(matrix([3,2],[7,260]));
(%o2)           picture(level, 2, 2, {Array:  #(3 2 7 255)})
(%i3) green: make_level_picture(matrix([54,23],[73,-9]));
(%o3)           picture(level, 2, 2, {Array:  #(54 23 73 0)})
(%i4) blue: make_level_picture(matrix([123,82],[45,32.5698]));
(%o4)          picture(level, 2, 2, {Array:  #(123 82 45 33)})
(%i5) make_rgb_picture(red,green,blue);
(%o5) picture(rgb, 2, 2,
              {Array:  #(3 54 123 2 23 82 7 73 45 255 0 33)})
(%i6) take_channel(%,'green);  /* simple quote!!! */
(%o6)           picture(level, 2, 2, {Array:  #(54 23 73 0)})
```

47.4 Funciones y variables para worldmap

Este paquete carga automáticamente el paquete `draw`.

47.4.1 Variables y Funciones

`boundaries_array` [Global variable]
Valor por defecto: `false`

`boundaries_array` es donde el objeto gráfico `geomap` lee las coordenadas de las líneas fronterizas.

Cada componente de `boundaries_array` es un array de números decimales en coma flotante representando las coordenadas que definen un segmento poligonal o línea fronteriza.

Véase también `geomap`.

`numbered_boundaries` (*nlist*) [Función]
Dibuja una lista de segmentos poligonales (líneas fronterizas), etiquetadas con sus números correspondientes (coordenadas de `boundaries_array`). Esta función es de mucha ayuda a la hora de definir nuevas entidades geográficas.

Ejemplo:

Mapa de Europa con las fronteras y costas etiquetadas con su componente numérica de `boundaries_array`.

```
(%i1) load("worldmap")$
(%i2) european_borders:
```

```
                region_boundaries(-31.81,74.92,49.84,32.06)$
(%i3) numbered_boundaries(european_borders)$
```

make_poly_continent (*continent_name*) [Función]
make_poly_continent (*country_list*) [Función]

Construye los polígonos necesarios para dibujar un continente o lista de países coloreados.

Ejemplo:

```
(%i1) load("worldmap")$
(%i2) /* A continent */
      make_poly_continent(Africa)$
(%i3) apply(draw2d, %)$
(%i4) /* A list of countries */
      make_poly_continent([Germany,Denmark,Poland])$
(%i5) apply(draw2d, %)$
```

make_poly_country (*country_name*) [Función]

Construye los polígonos necesarios para dibujar un país coloreado. En caso de contener islas, un país tendrá asociados varios polígonos.

Ejemplo:

```
(%i1) load("worldmap")$
(%i2) make_poly_country(India)$
(%i3) apply(draw2d, %)$
```

make_polygon (*nlist*) [Función]

Devuelve un objeto `polygon` a partie de una lista de líneas fronterizas y de costas. El argumento *nlist* debe ser una lista de componentes de `boundaries_array`.

Ejemplo:

La variable Bhutan (Bután) está definida con los números fronterizos 171, 173 y 1143, de manera que `make_polygon([171,173,1143])` concatena los arrays `boundaries_array[171]`, `boundaries_array[173]` y `boundaries_array[1143]` y devuelve un objeto `polygon` apto para ser dibujado por `draw`. A fin de evitar mensajes de errores, los arrays deben ser compatibles en el sentido de que dos de ellos consecutivos deben tener dos coordenadas comunes en los extremos. En este ejemplo, las dos primeras componentes de `boundaries_array[171]` son iguales a las dos últimas de `boundaries_array[173]`, y las dos primeras de `boundaries_array[173]` coinciden con las dos primeras de `boundaries_array[1143]`; en conclusión, los números de segmentos poligonales171, 173 y 1143 (en este orden) son compatibles y el polígono coloreado se podrá dibujar.

```
(%i1) load("worldmap")$
(%i2) Bhutan;
(%o2)                         [[171, 173, 1143]]
(%i3) boundaries_array[171];
(%o3) {Array:
       #(88.750549 27.14727 88.806351 27.25305 88.901367 27.282221
         88.917877 27.321039)}
```

```
(%i4) boundaries_array[173];
(%o4) {Array:
       #(91.659554 27.76511 91.6008 27.66666 91.598022 27.62499
         91.631348 27.536381 91.765533 27.45694 91.775253 27.4161
         92.007751 27.471939 92.11441 27.28583 92.015259 27.168051
         92.015533 27.08083 92.083313 27.02277 92.112183 26.920271
         92.069977 26.86194 91.997192 26.85194 91.915253 26.893881
         91.916924 26.85416 91.8358 26.863331 91.712479 26.799999
         91.542191 26.80444 91.492188 26.87472 91.418854 26.873329
         91.371353 26.800831 91.307457 26.778049 90.682457 26.77417
         90.392197 26.903601 90.344131 26.894159 90.143044 26.75333
         89.98996 26.73583 89.841919 26.70138 89.618301 26.72694
         89.636093 26.771111 89.360786 26.859989 89.22081 26.81472
         89.110237 26.829161 88.921631 26.98777 88.873016 26.95499
         88.867737 27.080549 88.843307 27.108601 88.750549
         27.14727)}
(%i5) boundaries_array[1143];
(%o5) {Array:
       #(91.659554 27.76511 91.666924 27.88888 91.65831 27.94805
         91.338028 28.05249 91.314972 28.096661 91.108856 27.971109
         91.015808 27.97777 90.896927 28.05055 90.382462 28.07972
         90.396088 28.23555 90.366074 28.257771 89.996353 28.32333
         89.83165 28.24888 89.58609 28.139999 89.35997 27.87166
         89.225517 27.795 89.125793 27.56749 88.971077 27.47361
         88.917877 27.321039)}
(%i6) Bhutan_polygon: make_polygon([171,173,1143])$
(%i7) draw2d(Bhutan_polygon)$
```

`region_boundaries` (*x1*,*y1*,*x2*,*y2*) [Función]

Detecta los segmentos poligonales almacenados en la variable global `boundaries_array` totalmente contenidos en el rectángulo de vértices (*x1*,*y1*) -superior izquierdo- y (*x2*,*y2*) -inferior derecho-.

Ejemplo:

Devuelve los números de los segmentos necesarios para dibujar el sur de Italia.

```
(%i1) load("worldmap")$
(%i2) region_boundaries(10.4,41.5,20.7,35.4);
(%o2)                [1846, 1863, 1864, 1881, 1888, 1894]
(%i3) draw2d(geomap(%))$
```

`region_boundaries_plus` (*x1*,*y1*,*x2*,*y2*) [Función]

Detecta los segmentos poligonales almacenados en la variable global `boundaries_array` con al menos un vértice dentro del rectángulo definido por los extremos (*x1*,*y1*) -superior izquierdo- y (*x2*,*y2*) -inferior derecho-.

Ejemplo:

```
(%i1) load("worldmap")$
(%i2) region_boundaries_plus(10.4,41.5,20.7,35.4);
```

```
(%o2) [1060, 1062, 1076, 1835, 1839, 1844, 1846, 1858,
       1861, 1863, 1864, 1871, 1881, 1888, 1894, 1897]
(%i3) draw2d(geomap(%))$
```

47.4.2 Objetos gráficos

`geomap` (*`numlist`*) [Objeto gráfico]
`geomap` (*`numlist`*,*`3Dprojection`*) [Objeto gráfico]

Dibuja mapas cartográficos en 2D y 3D.

2D

Esta función trabaja junto con la variable global `boundaries_array`.

El argumento *numlist* es una lista de números o de listas de números. Todos estos números deben ser enteros mayores o iguales que cero, representando las componentes del array global `boundaries_array`.

Cada componente de `boundaries_array` es un array de decimales en coma flotante, las coordenadas de un segmento poligonal o línea fronteriza.

`geomap (`*`numlist`*`)` toma los enteros de sus argumentos y dibuja los segmentos poligonales asociados de `boundaries_array`.

Este objeto se ve afectado por las siguientes *opciones gráficas*: `line_width`, `line_type` y `color`.

Ejemplos:

Un sencillo mapa hecho a mano:

```
(%i1) load("worldmap")$
(%i2) /* Vertices of boundary #0: {(1,1),(2,5),(4,3)} */
   ( bnd0: make_array(flonum,6),
     bnd0[0]:1.0, bnd0[1]:1.0, bnd0[2]:2.0,
     bnd0[3]:5.0, bnd0[4]:4.0, bnd0[5]:3.0 )$
(%i3) /* Vertices of boundary #1: {(4,3),(5,4),(6,4),(5,1)} */
   ( bnd1: make_array(flonum,8),
     bnd1[0]:4.0, bnd1[1]:3.0, bnd1[2]:5.0, bnd1[3]:4.0,
     bnd1[4]:6.0, bnd1[5]:4.0, bnd1[6]:5.0, bnd1[7]:1.0)$
(%i4) /* Vertices of boundary #2: {(5,1), (3,0), (1,1)} */
   ( bnd2: make_array(flonum,6),
     bnd2[0]:5.0, bnd2[1]:1.0, bnd2[2]:3.0,
     bnd2[3]:0.0, bnd2[4]:1.0, bnd2[5]:1.0 )$
(%i5) /* Vertices of boundary #3: {(1,1), (4,3)} */
   ( bnd3: make_array(flonum,4),
     bnd3[0]:1.0, bnd3[1]:1.0, bnd3[2]:4.0, bnd3[3]:3.0)$
(%i6) /* Vertices of boundary #4: {(4,3), (5,1)} */
   ( bnd4: make_array(flonum,4),
     bnd4[0]:4.0, bnd4[1]:3.0, bnd4[2]:5.0, bnd4[3]:1.0)$
(%i7) /* Pack all together in boundaries_array */
   ( boundaries_array: make_array(any,5),
     boundaries_array[0]: bnd0, boundaries_array[1]: bnd1,
     boundaries_array[2]: bnd2, boundaries_array[3]: bnd3,
```

```
     boundaries_array[4]: bnd4 )$
(%i8) draw2d(geomap([0,1,2,3,4]))$
```

El paquete auxiliar `worldmap` asigna al array global `boundaries_array` líneas fronterizas reales en coordenadas (longitud, latitud). Estos datos son de dominio público y proceden de http://www-cger.nies.go.jp/grid-e/gridtxt/grid19.html. El paquete `worldmap` también define fronteras de países, continentes y líneas costeras a partir de las componentes de `boundaries_array` (véase el fichero `share/draw/worldmap.mac` para más información). El paquete `worldmap` carga automáticamente el paquete `draw`.

```
(%i1) load("worldmap")$
(%i2) c1: gr2d(geomap([Canada,United_States,
                       Mexico,Cuba]))$
(%i3) c2: gr2d(geomap(Africa))$
(%i4) c3: gr2d(geomap([Oceania,China,Japan]))$
(%i5) c4: gr2d(geomap([France,Portugal,Spain,
                       Morocco,Western_Sahara]))$
(%i6) draw(columns  = 2,
           c1,c2,c3,c4)$
```

`worldmap` se puede utilizar para dibujar países como polígonos. En este caso, ya no será necesario hacer uso del objeto gráfico `geomap`, pero sí de `polygon`. Puesto que en este caso se utilizan listas en lugar de arrays, los mapas se reproducirán de forma más lenta. Véanse también `make_poly_country` y `make_poly_continent` para comprender el siguiente código.

```
(%i1) load("worldmap")$
(%i2) mymap: append(
   [color      = white],  /* borders are white */
   [fill_color = red],             make_poly_country(Bolivia),
   [fill_color = cyan],            make_poly_country(Paraguay),
   [fill_color = green],           make_poly_country(Colombia),
   [fill_color = blue],            make_poly_country(Chile),
   [fill_color = "#23ab0f"],       make_poly_country(Brazil),
   [fill_color = goldenrod],       make_poly_country(Argentina),
   [fill_color = "midnight-blue"], make_poly_country(Uruguay))$
(%i3) apply(draw2d, mymap)$
```

3D

`geomap` (*`numlist`*) proyecta los mapas sobre la esfera de radio 1 y centro (0,0,0). Es posible cambiar la esfera o el tipo de proyección haciendo uso de `geomap` (*`numlist`*,*`3Dprojection`*).

Proyecciones 3D disponibles:

- `[spherical_projection,`*`x`*`,`*`y`*`,`*`z`*`,`*`r`*`]`: proyecta los mapas sobre la esfera de radio *r* y centro (*x*,*y*,*z*).

```
(%i1) load("worldmap")$
(%i2) draw3d(geomap(Australia), /* default projection */
             geomap(Australia,
                    [spherical_projection,2,2,2,3]))$
```

- `[cylindrical_projection,x,y,z,r,rc]`: re-proyecta mapas esféricos sobre el cilindro de radio *rc* cuyo eje pasa a través de los polos del globo de radio *r* y centro (*x*,*y*,*z*).

```
(%i1) load("worldmap")$
(%i2) draw3d(geomap([America_coastlines,Eurasia_coastlines],
                    [cylindrical_projection,2,2,2,3,4]))$
```

- `[conic_projection,x,y,z,r,alpha]`: re-proyecta mapas esféricos sobre los conos de ángulo *alpha*, cuyos ejes pasan a través de los polos del globo de radio *r* y centro (*x*,*y*,*z*). Ambos conos, norte y sur, son tangentes a la esfera.

```
(%i1) load("worldmap")$
(%i2) draw3d(geomap(World_coastlines,
                    [conic_projection,0,0,0,1,90]))$
```

En https://riotorto.users.sourceforge.net/Maxima/gnuplot/geomap/ hay ejemplos más elaborados.

48 drawdf

48.1 Introducción a drawdf

La función `drawdf` dibuja el campo de direcciones de una ecuación diferencial ordinaria de primer orden (EDO) o de un sistema de dos ecuaciones autónomas de primer orden.

Puesto que `drawdf` es un paquete adicional, es necesario cargarlo en memoria ejecutando previamente la instrucción `load("drawdf")`. `drawdf` utiliza el paquete `draw`, que necesita como mínimo la versión 4.2 de Gnuplot.

Para dibujar el campo de direcciones de una EDO, ésta debe escribirse de la forma

$$\frac{dy}{dx} = F(x,y)$$

y ser la función F la que se pase a `drawdf` como argumento. Si las variables independiente y dependiente no son x e y, como en la ecuación anterior, entonces deben nombrarse de forma explícita en una lista que se pasará a `drawdf` (ver ejemplos más abajo).

Para dibujar el campo de direcciones de un conjunto de dos EDOs autónomas, deben escribirse de la forma

$$\frac{dx}{dt} = G(x,y) \qquad \frac{dy}{dt} = F(x,y)$$

y será una lista con las dos funciones G y F la que se pase a `drawdf` como argumento. El orden de las funciones es importante; la primera será la derivada de la abscisa respecto del tiempo y la segunda la derivada de la ordenada respecto del tiempo. Si las variables no son las x e y habituales, el segundo argumento que se pase a la función `drawdf` será una lista con los nombres de ambas variables, primero la correspondiente a las abscisas, seguida de la asociada a las ordenadas.

Si sólo se trata de una EDO, `drawdf` admitirá por defecto que `x=t` y `G(x,y)=1`, transformanado la ecuación no autónoma en un sistema de dos ecuaciones autónomas.

48.2 Funciones y variables para drawdf

48.2.1 Funciones

`drawdf` (*dydx, ...options and objects...*) [Función]
`drawdf` (*dvdu*, [*u*,*v*], *...opciones y objetos...*) [Función]
`drawdf` (*dvdu*, [*u*,*umin*,*umax*], [*v*,*vmin*,*vmax*], *...opciones y objetos...*) [Función]
`drawdf` ([*dxdt*,*dydt*], *...opciones y objetos...*) [Función]
`drawdf` ([*dudt*,*dvdt*], [*u*,*v*], *...opciones y objetos...*) [Función]
`drawdf` ([*dudt*,*dvdt*], [*u*,*umin*,*umax*], [*v*,*vmin*,*vmax*], *...opciones y objetos...*) [Función]

La función `drawdf` dibuja un campo de direcciones 2D, incluyendo opcionalmente curvas asociadas a soluciones particulares, así como otros objeto gráficos del paquete `draw`.

El primer argumento especifica la(s) derivada(s), el cual debe estar formado por una expresión o una lista de dos expresiones. *dydx*, *dxdt* y *dydt* son expresiones que dependen de x y y. *dvdu*, *dudt* y *dvdt* son expresiones que dependen de u y v.

Si las variables independiente y dependiente no son x e y, sus nombres deben especificarse a continuación de las derivadas, bien como una lista de dos nombres, [u,v], o como dos listas de la forma [u,$umin$,$umax$] y [v,$vmin$,$vmax$].

El resto de argumentos son *opciones gráficas*, *objetos gráficos*, o listas conteniendo elementos de ambos tipos. El conjunto de opciones y objetos gráficos soportados por `drawdf` incluye los soportados por `draw2d` y `gr2d` del paquete `draw`.

Los argumentos se iterpretan secuencialmente: las *opciones gráficas* afectan a todos los *objetos gráficos* que le siguen. Además, los *objetos gráficos* se dibujan en el orden en el que se especifican, pudiendo tapar otros gráficos dibujados con anterioridad. Algunas *opciones gráficas* afectan al aspecto general de la escena.

Los nuevos *objetos gráficos* que se definen en `drawdf` son: `solns_at`, `points_at`, `saddles_at`, `soln_at`, `point_at` y `saddle_at`.

Las nuevas *opciones gráficas* que se definen en `drawdf` son: `field_degree`, `soln_arrows`, `field_arrows`, `field_grid`, `field_color`, `show_field`, `tstep`, `nsteps`, `duration`, `direction`, `field_tstep`, `field_nsteps` y `field_duration`.

Objetos gráficos que se heredan del paquete `draw` incluyen: `explicit`, `implicit`, `parametric`, `polygon`, `points`, `vector`, `label` y cualesquiera otros soportados por `draw2d` y `gr2d`.

Opciones gráficas que se heredan del paquete `draw` incluyen: `points_joined`, `color`, `point_type`, `point_size`, `line_width`, `line_type`, `key`, `title`, `xlabel`, `ylabel`, `user_preamble`, `terminal`, `dimensions`, `file_name` y cualesquiera otros soportados por `draw2d` y `gr2d`.

Véase también `draw2d`.

Los usuarios de wxMaxima y Imaxima pueden hacer uso también `wxdrawdf`, que es idéntica a `drawdf`, pero que inserta el gráfico en el propio documento utilizando la función `wxdraw`.

Para hacer uso de esta función, ejecútese primero `load("drawdf")`.

Ejemplos:

```
(%i1) load("drawdf")$
(%i2) drawdf(exp(-x)+y)$        /* default vars: x,y */
(%i3) drawdf(exp(-t)+y, [t,y])$ /* default range: [-10,10] */
(%i4) drawdf([y,-9*sin(x)-y/5], [x,1,5], [y,-2,2])$
```

A efectos de compatibilidad, `drawdf` acepta la mayor parte de parámetros soportados por `plotdf`.

```
(%i5) drawdf(2*cos(t)-1+y, [t,y], [t,-5,10], [y,-4,9],
              [trajectory_at,0,0])$
```

`soln_at` y `solns_at` dibujan soluciones particulares que pasen por los puntos especificados, utilizando el integrador numérico de Runge Kutta de cuarto orden.

```
(%i6) drawdf(2*cos(t)-1+y, [t,-5,10], [y,-4,9],
              solns_at([0,0.1],[0,-0.1]),
              color=blue, soln_at(0,0))$
```

`field_degree=2` hace que el campo se componga de *splines* cuadráticos basados en las derivadas de primer y segundo orden en cada punto de la malla. `field_grid=[`*COLS*,*ROWS*`]` especifica el número de columnas y filas a utilizar en la malla.

```
(%i7) drawdf(2*cos(t)-1+y, [t,-5,10], [y,-4,9],
             field_degree=2, field_grid=[20,15],
             solns_at([0,0.1],[0,-0.1]),
             color=blue, soln_at(0,0))$
```

`soln_arrows=true` añade flechas a las soluciones particulares y, por defecto, las borra. También cambia los colores por defecto para destacar las curvas de las soluciones particulares.

```
(%i8) drawdf(2*cos(t)-1+y, [t,-5,10], [y,-4,9],
             soln_arrows=true,
             solns_at([0,0.1],[0,-0.1],[0,0]))$
```

`duration=40` especifica el intervalo temporal de la integración numérica, cuyo valor por defecto es 10. La integración se detendrá automáticamente si la solución se aparta mucho de la región a dibujar, o si la derivada se vuelve compleja o infinita. Aquí también hacemos `field_degree=2` para dibujar *splines* cuadráticos. Las ecuaciones de este ejemplo modelizan un sistema depredador-presa.

```
(%i9) drawdf([x*(1-x-y), y*(3/4-y-x/2)], [x,0,1.1], [y,0,1],
             field_degree=2, duration=40,
             soln_arrows=true, point_at(1/2,1/2),
             solns_at([0.1,0.2], [0.2,0.1], [1,0.8], [0.8,1],
                      [0.1,0.1], [0.6,0.05], [0.05,0.4],
                      [1,0.01], [0.01,0.75]))$
```

`field_degree='solns` hace que el campo se componga de muchas pequeñas soluciones calculadas a partir del Runge Kutta de cuarto orden.

```
(%i10) drawdf([x*(1-x-y), y*(3/4-y-x/2)], [x,0,1.1], [y,0,1],
              field_degree='solns, duration=40,
              soln_arrows=true, point_at(1/2,1/2),
              solns_at([0.1,0.2], [0.2,0.1], [1,0.8],
                       [0.8,1], [0.1,0.1], [0.6,0.05],
                       [0.05,0.4], [1,0.01], [0.01,0.75]))$
```

`saddles_at` trata de linearizar automáticamente la ecuación en cada punto de silla y dibujar la solución numérica correspondiente a cada vector propio, incluyendo las separatrices. `tstep=0.05` establece el salto temporal máximo para el integrador numérico, cuyo valor por defecto es 0.1. Las siguientes ecuaciones modelizan un péndulo amortiguado.

```
(%i11) drawdf([y,-9*sin(x)-y/5], tstep=0.05,
              soln_arrows=true, point_size=0.5,
              points_at([0,0], [2*%pi,0], [-2*%pi,0]),
              field_degree='solns,
              saddles_at([%pi,0], [-%pi,0]))$
```

`show_field=false` elimina el campo completamente.

```
(%i12) drawdf([y,-9*sin(x)-y/5], tstep=0.05,
              show_field=false, soln_arrows=true,
              point_size=0.5,
              points_at([0,0], [2*%pi,0], [-2*%pi,0]),
              saddles_at([3*%pi,0], [-3*%pi,0],
```

```
                                [%pi,0], [-%pi,0]))$
```

`drawdf` pasa todos los parámetros que no reconoce a `draw2d` o `gr2d`, permitiendo combinar la potencia del paquete `draw` con `drawdf`.

```
(%i13) drawdf(x^2+y^2, [x,-2,2], [y,-2,2], field_color=gray,
              key="soln 1", color=black, soln_at(0,0),
              key="soln 2", color=red, soln_at(0,1),
              key="isocline", color=green, line_width=2,
              nticks=100, parametric(cos(t),sin(t),t,0,2*%pi))$
```

`drawdf` acepta listas anidadas de opciones y objetos gráficos, permitiendo el uso de `makelist` y otras funciones de forma más flexible para generar gráficos.

```
(%i14) colors : ['red,'blue,'purple,'orange,'green]$
(%i15) drawdf([x-x*y/2, (x*y - 3*y)/4],
              [x,2.5,3.5], [y,1.5,2.5],
              field_color = gray,
              makelist([ key   = concat("soln",k),
                         color = colors[k],
                         soln_at(3, 2 + k/20) ],
                       k,1,5))$
```

49 dynamics

49.1 El paquete dynamics

El paquete adicional `dynamics` incluye funciones para visualización 3D, animaciones, análisis gráfico de ecuaciones diferenciales y ecuaciones de diferencias y para resolución numérica de ecuaciones diferenciales. Las funciones para ecuaciones diferenciales se describen en la sección sobre `Capítulo 22 Métodos numéricos` y las funciones para representar las gráficas de los conjuntos de Mandelbrot y de Julia se describen en la sección sobre `Capítulo 12 Gráficos`.

Todas las funciones en este paquete se cargan automáticamente la primera vez que se usan.

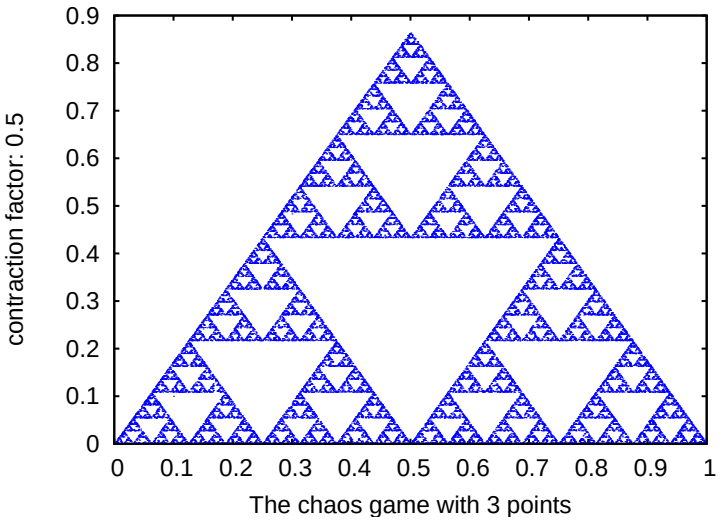
49.2 Análisis gráfico de sistemas dinámicos discretos

`chaosgame` ([[*x1*, *y1*]...[*xm*, *ym*]], [*x0*, *y0*], *b*, *n*, *opciones*, ...); [Función]

Usa el método llamado juego del caos, para producir fractales: se dibuja un punto inicial (*x0*, *y0*) y luego se elije aleatoriamente uno de los *m* puntos [*x1*, *y1*]...[*xm*, *ym*]. Después se dibuja un nuevo punto que estará en el segmento entre el último punto dibujado y el punto que se acabó de elegir aleatoriamente, a una distancia del punto elegido que será *b* veces la longitud del segmento. El proceso se repite *n* veces. Este programa acepta las mismas opciones de `plot2d`.

Ejemplo. Gráfico del triángulo de Sierpinsky:

```
(%i1) chaosgame([[0, 0], [1, 0], [0.5, sqrt(3)/2]], [0.1, 0.1], 1/2,
                30000, [style, dots]);
```



`evolution` *evolution* (*F*, *y0*, *n*, ..., *opciones*, ...); [Función]

Dibuja *n+1* puntos en una gráfica bidimensional (serie de tiempo), en que las coordenadas horizontales de los puntos son los números enteros 0, 1, 2, ..., *n*, y las coordenadas verticales son los valores *y(n)* correspondientes, obtenidos a partir de la

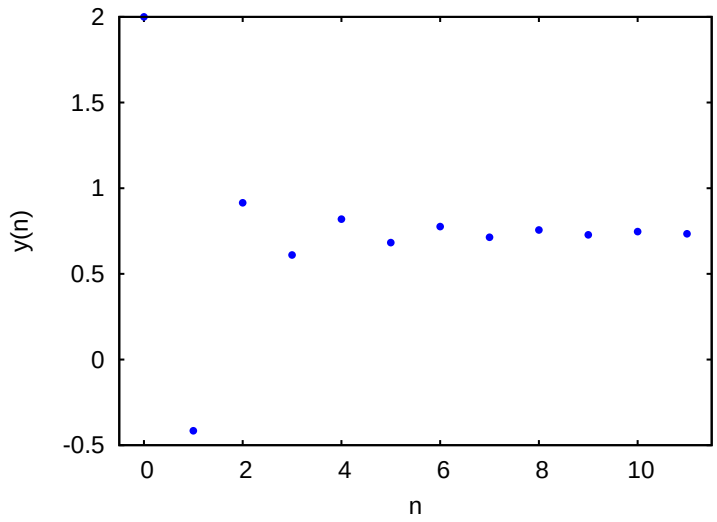
relación de recurrencia

$$y_{n+1} = F(y_n)$$

Con valor inicial *y(0)* igual a *y0*. *F* deberá ser una expresión que dependa únicamente de la variable *y* (y no de *n*), *y0* deberá ser un número real y *n* un número entero positivo. Esta función acepta las mismas opciones que `plot2d`.

Ejemplo.

```
(%i1) evolution(cos(y), 2, 11);
```



`evolution2d` (*[F, G], [u, v], [u0, y0], n, opciones, ...*); [Función]

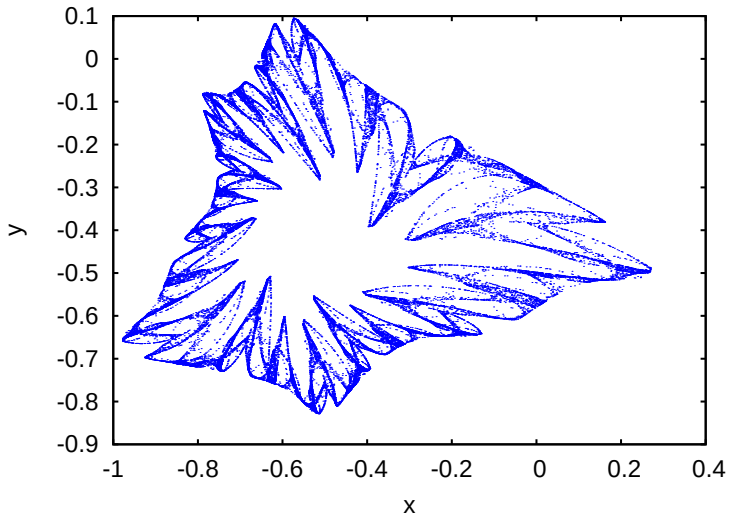
Muestra, en una gráfica bidimensional, los primeros *n+1* puntos de la sucesión definida a partir del sistema dinámico discreto con relaciones de recurrencia:

$$\begin{cases} x_{n+1} = F(x_n, y_n) \\ y_{n+1} = G(x_n, y_n) \end{cases}$$

Con valores iniciales *x0* y *y0*. *F* y *G* deben ser dos expresiones que dependan únicamente de *x* y *y*. Esta función acepta las mismas opciones que `plot2d`.

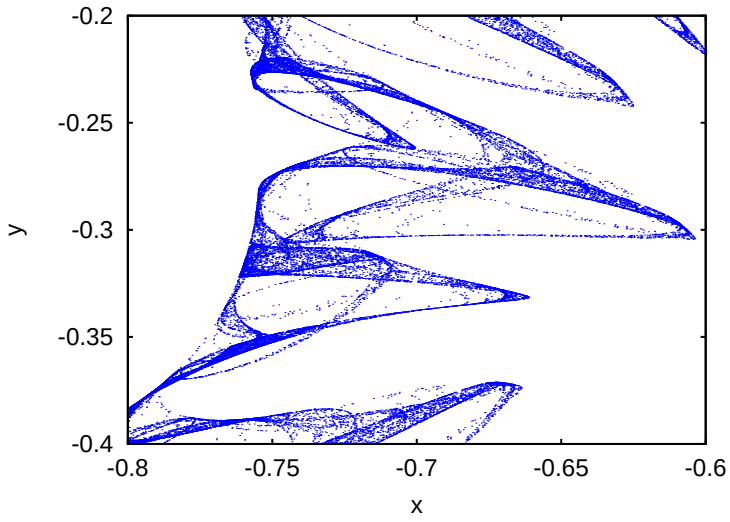
Ejemplo. Evolución de un sistema dinámico discreto en dos dimensiones:

```
(%i1) f: 0.6*x*(1+2*x)+0.8*y*(x-1)-y^2-0.9$
(%i2) g: 0.1*x*(1-6*x+4*y)+0.1*y*(1+9*y)-0.4$
(%i3) evolution2d([f,g], [x,y], [-0.5,0], 50000, [style,dots]);
```



Y un acercamiento de una pequeña región en ese fractal:

```
(%i9) evolution2d([f,g], [x,y], [-0.5,0], 300000, [x,-0.8,-0.6],
                  [y,-0.4,-0.2], [style, dots]);
```



`ifs` (*[r1, ..., rm], [A1,..., Am], [[x1, y1], ..., [xm, ym]], [x0, y0], n, opciones, ...*); [Función]

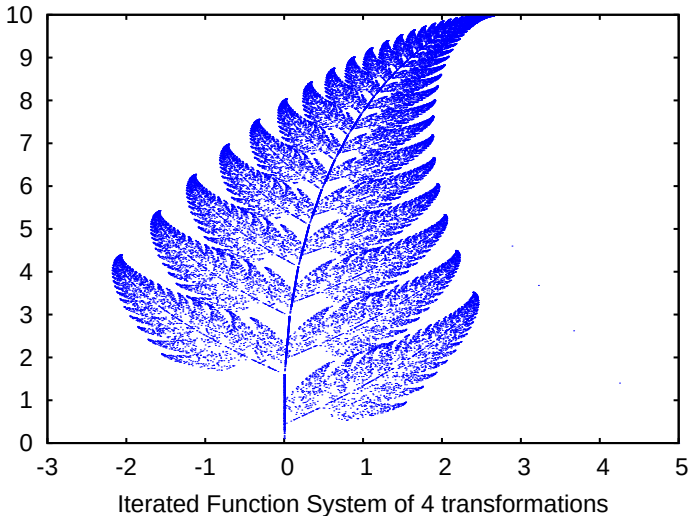
Usa el método del Sistema de Funciones Iteradas (IFS, en inglés Iterated Function System). Ese método es semejante al método descrito en la función `chaosgame`. pero en vez de aproximar el último punto al punto elegido aleatoriamente, las dos coordenadas del último punto se multiplican por una matriz 2 por 2 *Ai* correspondiente al punto que fue elegido aleatoriamente.

La selección aleatoria de uno de los *m* puntos atractivos puede ser realizada con una función de probabilidad no uniforme, definida con los pesos *r1*,...,*rm*. Esos pesos deben ser dados en forma acumulada; por ejemplo, si se quieren 3 puntos con probabilidades 0.2, 0.5 y 0.3, los pesos *r1*, *r2* y *r3* podrían ser 2, 7 y 10, o cualquier otro grupo de

números que tengan la misma proporción. Esta función acepta las mismas opciones que `plot2d`.

Ejemplo. El helecho de Barnsley, creado con 4 matrices y 4 puntos:

```
(%i1) a1: matrix([0.85,0.04],[-0.04,0.85])$
(%i2) a2: matrix([0.2,-0.26],[0.23,0.22])$
(%i3) a3: matrix([-0.15,0.28],[0.26,0.24])$
(%i4) a4: matrix([0,0],[0,0.16])$
(%i5) p1: [0,1.6]$
(%i6) p2: [0,1.6]$
(%i7) p3: [0,0.44]$
(%i8) p4: [0,0]$
(%i9) w: [85,92,99,100]$
(%i10) ifs(w, [a1,a2,a3,a4], [p1,p2,p3,p4], [5,0], 50000, [style,dots]);
```



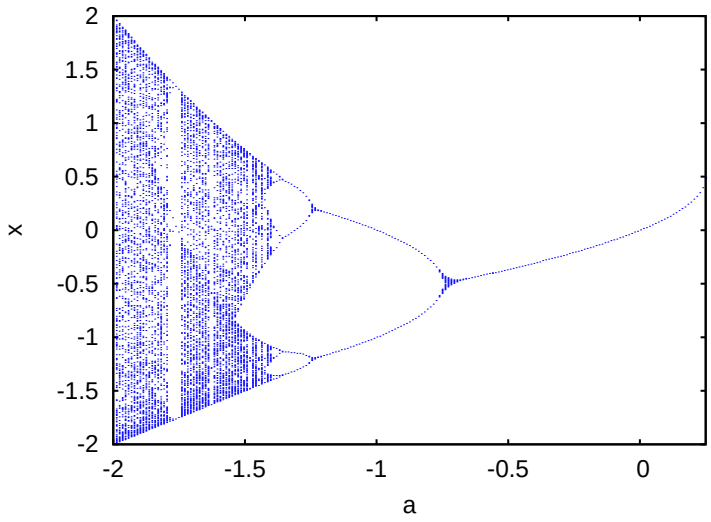
`orbits` (*F*, *y0*, *n1*, *n2*, [*x*, *x0*, *xf*, *xstep*], *opciones*, ...); [Función]

Dibuja el diagrama de órbitas de una familia de sistemas dinámicos discretos unidimensionales, con un parámetro *x*; ese tipo de diagrama se usa para mostrar las bifurcaciones de un sistema discreto unidimensional.

La función *F(y)* define una secuencia que comienza con un valor inicial *y0*, igual que en el caso de la función `evolution`, pero en este caso la función también dependerá del parámetro *x*, el cual tomará valores comprendidos en el intervalo de *x0* a *xf*, con incrementos *xstep*. Cada valor usado para el parámetro *x* se muestra en el eje horizontal. En el eje vertical se mostrarán *n2* valores de la sucesión *y(n1+1)*,..., *y(n1+n2+1)*, obtenidos después de dejarla evolucionar durante *n1* iteraciones iniciales.

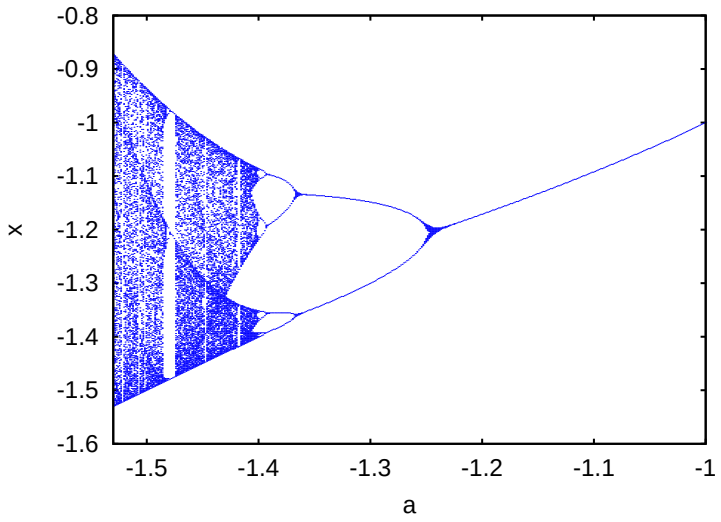
Ejemplo. Diagrama de órbitas para el mapa cuadrático

```
(%i1) orbits(x^2+a, 0, 50, 200, [a, -2, 0.25], [style, dots]);
```



Para ampliar la región alrededor de la bifurcación en la parte de abajo, cerca de x = -1.25, se usa el comando:

```
(%i2) orbits(x^2+a, 0, 100, 400, [a,-1,-1.53], [x,-1.6,-0.8],
                [nticks, 400], [style,dots]);
```



`staircase` (*F*, *y0*, *n*,*opciones*,. . .); [Función]

Dibuja un diagrama de escalera (o diagrama de red) para la sucesión definida por la ecuación de recurrencia

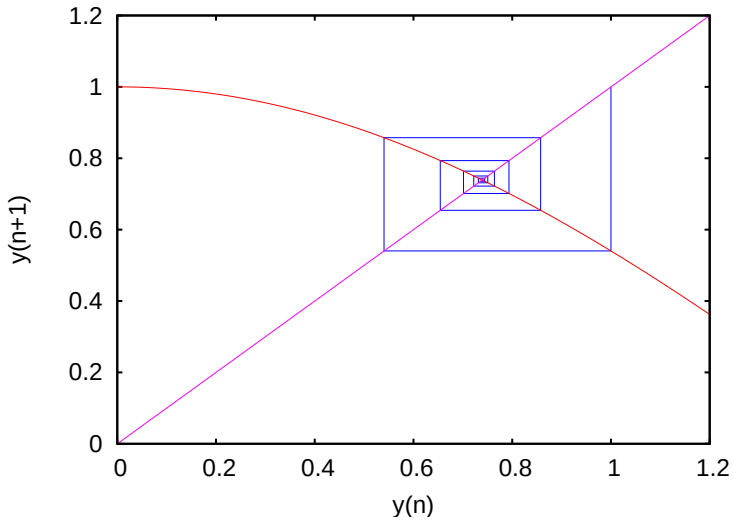
$$y_{n+1} = F(y_n)$$

La interpretación y valores permitidos de los parámetros de entrada es la misma que para la función `evolution`. Un diagrama de escalera consiste en una gráfica de la función *F(y)*, junto con la recta *G(y)* = *y*. Se comienza por dibujar un segmento vertical desde el punto (*y0*, *y0*) en la recta, hasta el punto de intersección con la función *F*. En seguida, desde ese punto se dibuja un segmento horizontal hasta el

punto de intersección con la recta, (*y1*, *y1*); el procedimiento se repite *n* veces hasta alcanzar el punto (*yn*, *yn*). Esta función acepta las mismas opciones que `plot2d`.

Ejemplo.

```
(%i1) staircase(cos(y), 1, 11, [y, 0, 1.2]);
```



49.3 Visualización usando VTK

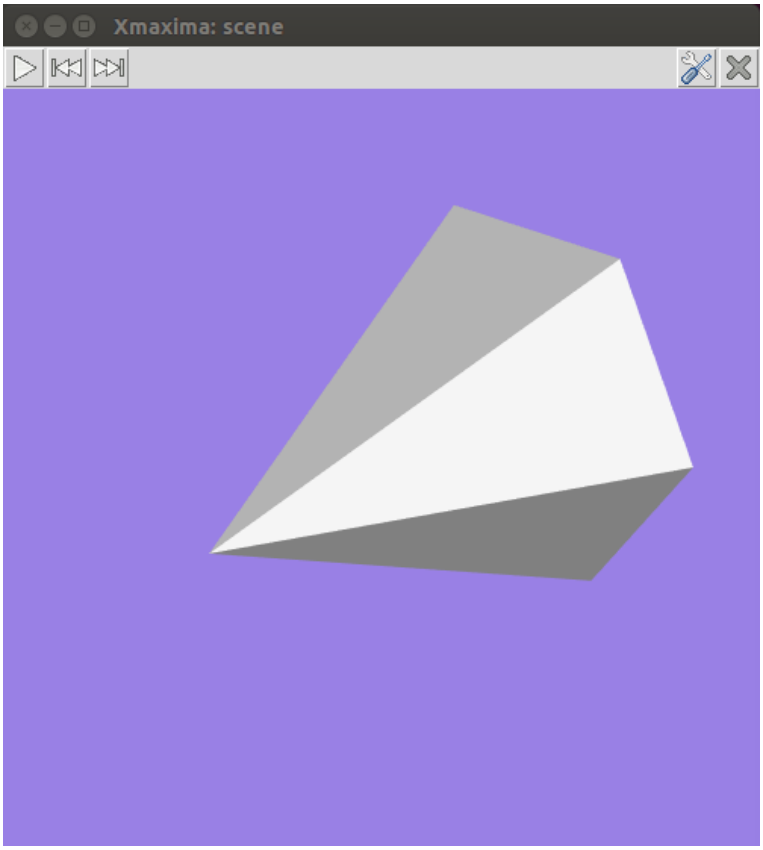
La función scene crea imágenes tridimensionales y animaciones usando el software *Visualization ToolKit* (VTK). Para poder usar esa función es necesario tener Xmaxima y VTK instalados en el sistema (incluyendo la biblioteca para usar VTK desde TCL que en algunos sistemas viene en un paquete separado).

`scene` (*objetos*, ..., *opciones*, ...); [Función]

Acepta una lista vacía o una lista de varios `[scene_objetos], página 785` y `[scene_opciones], página 784`. El programa ejecuta Xmaxima, que abre una ventana donde se representan los objetos dados, en un espacio tridimensional y aplicando las opciones dadas. Cada objeto debe pertenecer una de las 4 clases: sphere, cube, cylinder o cone (ver `[scene_objetos], página 785`). Los objetos se identifican dando el nombre de su clase, o una lista en que el primer elemento es el nombre de la clase y los restantes elementos son opciones para ese objeto.

Ejemplo. Una pirámide hexagonal con fondo azul:

```
(%i1) scene(cone, [background,"#9980e5"])$
```



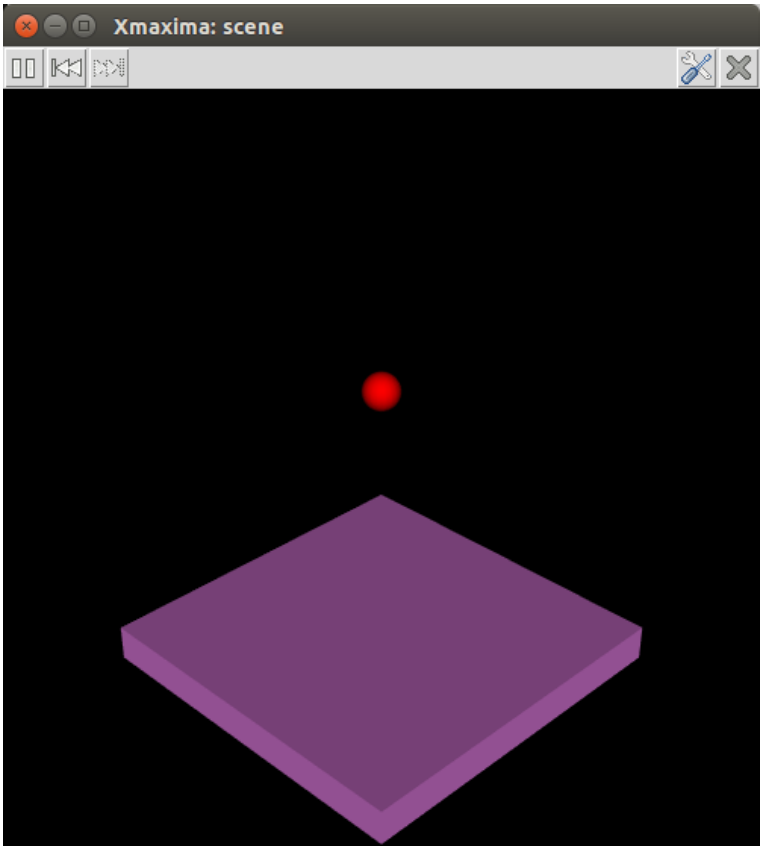
Presionando el botón izquierdo del ratón mientras se mueve dentro de la ventana gráfica, se puede rotar la cámara, mostrando diferentes vistas de la pirámide. Las dos opciones [scene_elevation], página 784 y [scene_azimuth], página 784 se pueden usar también para cambiar la orientación de la cámara. La cámara se puede mover presionando el botón del medio y moviendo el ratón y el botón del lado derecho se usa para aumentar o disminuir el zoom, moviendo el ratón para arriba y para bajo.

Cada opción de un objeto deberá ser una lista comenzando con el nombre de la opción, seguido por su valor. La lista de posibles opciones se encuentra en la sección [objeto_opciones], página 786.

Ejemplo. Este ejemplo mostrará una esfera cayendo para el piso y rebotando sin perder energía. Para comenzar o poner en pausa la animación se debe oprimir el botón de play/pausa.

```
(%i1) p: makelist ([0,0,2.1- 9.8*t^2/2], t, 0, 0.64, 0.01)$

(%i2) p: append (p, reverse(p))$

(%i3) bola: [sphere, [radius,0.1], [thetaresolution,20],
  [phiresolution,20], [position,0,0,2.1], [color,red],
  [animate,position,p]]$

(%i4) piso: [cube, [xlength,2], [ylength,2], [zlength,0.2],
  [position,0,0,-0.1],[color,violet]]$
```

```
(%i5) scene (bola, piso, restart)$
```



La opción *restart* fue usada para hacer que la animación recomienze cada vez que se llega al último punto en la lista. Los valores permitidos para los colores son los mismo que para la opción `color` de plot2d.

49.3.1 Opciones de scene

`azimuth` [*azimuth*, *ángulo*] [Opción de scene]
Valor predefinido: `135`

La rotación de la cámara en el plano horizontal (x, y). *ángulo* debe ser un número real; un ángulo de 0 significa que la cámara apunta en la dirección del eje y, haciendo que el eje x aparezca a la derecha.

`background` [*background*, *color*] [Opción de scene]
Valor predefinido: `black`

El color del fondo de la ventana gráfica. Acepta nombres de colores o secuencias de caracteres hexadecimales para rojo-verde-azul (ver la opción `color` de plot2d).

`elevation` [*elevation*, *ángulo*] [Opción de scene]
Valor predefinido: `30`

La rotación vertical de la cámara. El *ángulo* debe ser un número real; un ángulo de 0 significa que la cámara apunta en la horizontal y el valor predefinido de 30 significa que la cámara apunta 30 grados para abajo de la horizontal.

`height` [*height*, `pixels`] [Opción de scene]
Valor predefinido: `500`

La altura, en pixels, de la ventana gráfica. *pixels* debe ser un número entero positivo.

`restart` [*restart*, `valor`] [Opción de scene]
Valor predefinido: `false`

Un valor 'true' significa que la animación recomenzará automáticamente cuando llegue al final. Escribir simplemente "restart" es equivalente a escribir [restart, *true*].

`tstep` [*tstep*, `tiempo`] [Opción de scene]
Valor predefinido: `10`

El intervalo de tiempo, en milisegundos, entre iteraciones consecutivas de las partes de una animación. *tiempo* debe ser un número real.

`width` [*width*, `pixels`] [Opción de scene]
Valor predefinido: `500`

El ancho, en pixels, de la ventana gráfica. *pixels* deberá ser un número entero positivo.

`windowname` [*windowtitle*, `nombre`] [Opción de scene]
Valor predefinido: `.scene`

nombre deberá ser una secuencia de caracteres que pueda ser usada para el nombre de una ventana de Tk, creada por Xmaxima para los gráficos de `scene`. El valor predefinido `.scene` implica que será creada una nueva ventana.

`windowtitle` [*windowtitle*, `nombre`] [Opción de scene]
Valor predefinido: `Xmaxima: scene`

nombre debe ser una secuencia de caracteres que serán escritos en el encabezado de la ventana creada por `scene`.

49.3.2 Objetos de scene

`cone` [*cone*, `opciones`] [Objeto de scene]
Crea una pirámide regular con altura 1 unidad y base hexagonal con vértices a 0.5 unidades del eje. Las opciones `[objeto_height], página 787` y `[objeto_radius], página 788` se pueden usar para alterar esos valores predefinidos y la opción `[objeto_resolution], página 788` se usa para alterar el número de aristas de la base; valores mayores harán que la pirámide parezca un cono. Por omisión, el eje estará a lo largo del eje x, el punto medio de la altura estará en el origen y el vértice en el lado positivo del eje x; las opciones `[objeto_orientation], página 787` y `[objeto_center], página 786` permiten alterar eso.

Ejemplo. El siguiente comando muestra una pirámide que comienza a dar vueltas cuando se oprime el botón de animación.

```
(%i1) scene([cone, [orientation,0,30,0], [tstep,100],
   [animate,orientation,makelist([0,30,i],i,5,360,5)]], restart)$
```

`cube` [*cube*, *opciones*] [Objeto de scene]

Un cubo con aristas de 1 unidad y caras paralelas al los planos xy, xz y yz. Las longitudes de las 3 aristas se pueden modificar con las opciones [objeto_xlength], página 790, [objeto_ylength], página 790 y [objeto_zlength], página 790, convirtiéndolo en un paralelepípedo y las caras se pueden hacer rotar con la opción [objeto_orientation], página 787.

`cylinder` [*cylinder*, *opciones*] [Objeto de scene]

Crea un prisma regular con altura de 1 unidad y base hexagonal con los vértices a 0.5 unidades del eje. Las opciones [objeto_height], página 787 y [objeto_radius], página 788 permiten modificar esos valores predefinidos y la opción [objeto_resolution], página 788 se puede usar para modificar el número de aristas de la base; valores mayores tornan el prisma más parecido a un cilindro. La altura predefinida se puede alterar con la opción [objeto_height], página 787. Por omisión el eje estará orientado a lo largo del eje x, y el punto medio de la altura estará en el origen; se pueden cambiar esas propiedades con las opciones [objeto_orientation], página 787 y [objeto_center], página 786.

`sphere` [*sphere*, *opciones*] [Objeto de scene]

Una esfera de radio igual a 0.5 unidades y centro en el origen.

49.3.3 Opciones de objetos de scene

`animation` [*animation*, *propiedad*, *posiciones*] [Opción de objeto]

propiedad deberá ser una de las 4 siguientes propiedades del objeto: [objeto_origin], página 787, [objeto_scale], página 788, [objeto_position], página 788 o [objeto_orientation], página 787 y *posiciones* deberá ser una lista de puntos. Cuando se oprime el botón "play", esa propiedad tomará sucesivamente los valores en la lista, en intervalos regulares de tiempo definidos por la opción [scene_tstep], página 785. El botón de volver al comienzo puede usarse para recomenzar desde el principio de la lista, haciendo que la animación comience nuevamente cuando se presione otra vez el botón de play.

Consulte también [objeto_track], página 789.

`capping` [*capping*, *número*] [Opción de objeto]

Valor predefinido: `1`

En un objeto "cone" o "cylinder", determina si la base (o bases) se muestran o no. Un valor de 1 para *número* torna la base visible y un valor de 0 la hace invisible.

`center` [*center*, *punto*] [Opción de objeto]

Valor predefinido: `[0, 0, 0]`

Coordenadas del centro geométrico del objeto, en relación a su posición ([objeto_position], página 788); *punto* puede ser una lista de 3 números reales o 3 números separados por comas. En objetos de clase "cylinder","cone" o "cube" estará localizado en el punto medio de la altura y en objetos de clase "sphere" estará en su centro.

`color` [*color*, *nombre*] [Opción de objeto]

Valor predefinido: `white`

El color del objeto. Acepta nombres de colores conocidos o secuencias de 6 dígitos (rojo-verde-azul) precedidos por # (ver la opción `color` de plot2d).

`endphi` [*endphi*, *ángulo*] [Opción de objeto]
Valor predefinido: `180`

En una esfera, phi es el ángulo en el plano vertical que pasa por el eje z, medido a partir de la parte positiva del eje z. *ángulo* debe ser un múmero entre 0 y 180 que determina el valor final de phi hasta donde se mostrará la superficie. Valores menores que 180 eliminan una parte de la superficie.

Consulte también `[objeto_startphi], página 788` y `[objeto_phiresolution], página 788`.

`endtheta` [*endtheta*, *ángulo*] [Opción de objeto]
Valor predefinido: `360`

En una esfera, theta es el ángulo en el plano horizontal (longitud), medido desde la parte positiva del eje x. *ángulo* debe ser un número entre 0 y 360 que determina el valor final donde la superficie terminará. Valores menores que 360 hacen que se elimine una parte de la superficie de la esfera.

Consulte también `[objeto_starttheta], página 789` y `[objeto_thetaresolution], página 789`.

`height` [*height*, *valor*] [Opción de objeto]
Valor predefinido: `1`

valor debe ser un número positivo que define la altura de un cono o cilindro.

`linewidth` [*linewidth*, *valor*] [Opción de objeto]
Valor predefinido: `1`

El ancho de las líneas cuando se usa la opción `[objeto_wireframe], página 790`. *valor* debe ser un número positivo.

`opacity` [*opacity*, *valor*] [Opción de objeto]
Valor predefinido: `1`

valor debe se un número entre 0 y 1. Cuanto menor sea el número, mas transparente será el objeto. El valor predefinido de 1 implica un objeto completamente opaco.

`orientation` [*orientation*, *ángulos*] [Opción de objeto]
Valor predefinido: `[0, 0, 0]`

Tres ángulos que de rotaciones que serán realizadas en el objeto, alrededor de los tres ejes.*ángulos* puede ser una lista con tres números reales o tres números separados por comas. **Ejemplo**: `[0, 0, 90]` rota el eje x del objeto hasta el eje y del sistema de referencia.

`origin` [*origin*, *punto*] [Opción de objeto]
Valor predefinido: `[0, 0, 0]`

Coordenadas del origen del objeto, en relación al cual las otras dimensiones serán definidas. *punto* pueden ser tres números separados por comas, o una lista de tres números.

`phiresolution` [*phiresolution, num*] [Opción de objeto]

Valor predefinido:

Número de sub-intervalos en que se divide el intervalo de valores del ángulo phi, que comienza en `[objeto_startphi], página 788` y termina en `[objeto_endphi], página 787`. *num* debe ser un número entero positivo.

Consulte también `[objeto_startphi], página 788` y `[objeto_endphi], página 787`.

`points` [*points*] [Opción de objeto]

Esta opción hace que en vez de que se muestre la superficie, se muestren únicamente los vértices en la triangulación usada para construir la superficie. **Ejemplo**: `[sphere, [points]]`

Consulte también `[objeto_surface], página 789` y `[objeto_wireframe], página 790`.

`pointsize` [*pointsize, valor*] [Opción de objeto]

Valor predefinido: `1`

Tamaño de los puntos, cuando se usa la opción `[objeto_points], página 788`. *valor* debe ser un número positivo.

`position` [*position, punto*] [Opción de objeto]

Valor predefinido: `[0, 0, 0]`

Coordenadas de la posición del objeto. *punto* puede ser tres números separados por comas o una lista de tres números.

`radius` [*radius, valor*] [Opción de objeto]

Valor predefinido: `0.5`

El radio de una esfera o la distancia desde el eje hasta los vértices de la base en cilindros o conos. *valor* debe ser un número positivo.

`resolution` [*resolution, número*] [Opción de objeto]

Valor predefinido: `6`

número debe ser un número entero mayor que 2, que define el número de aristas de la base en objetos de clase “cone” o “cylinder”.

`scale` [*scale, factores*] [Opción de objeto]

Valor predefinido: `[1, 1, 1]`

Tres factores de escala en que serán deformadas las dimensiones del objeto en relación a los tres ejes. *factors* pueden ser tres número separados por comas o una lista con tres números. **Ejemplo**: `[2, 0.5, 1]` estira el objeto suplicando sus dimensiones paralelas la eje x, reduce a mitad las dimensiones paralelas al eje y, dejando iguales las dimensiones paralelas al eje z.

`startphi` [*startphi, ángulo*] [Opción de objeto]

Valor predefinido: `0`

En una esfera, phi es el ángulo en el plano vertical que pasa por el eje z, medido a partir de la parte positiva del eje z. *ángulo* debe ser un múmero entre 0 y 180

que determina el valor inicial de phi desde donde se mostrará la superficie. Valores superiores a 0 eliminan una parte de la superficie.

Consulte también [objeto_endphi], página 787 y [objeto_phiresolution], página 788.

`starttheta` [*starttheta*, *ángulo*] [Opción de objeto]

Valor predefinido: `0`

En una esfera, theta es el ángulo en el plano horizontal (longitud), medido desde la parte positiva del eje x. *ángulo* debe ser un número entre 0 y 360 que determina el valor inicial donde la superficie comienza. Valores mayores que 0 hacen que se elimine una parte de la superficie de la esfera.

Consulte también [objeto_endtheta], página 787 y [objeto_thetaresolution], página 789.

`surface` [*surface*] [Opción de objeto]

Hace que se muestre la superficie del objeto y no los las líneas ni los vértices de la triangulación usada para construirlo. Ese es el comportamiento habitual que se puede alterar con las opciones [objeto_points], página 788 o [objeto_wireframe], página 790.

`thetaresolution` [*thetaresolution*, *num*] [Opción de objeto]

Valor predefinido:

Número de sub-intervalos en que se divide el intervalo de valores del ángulo theta, que comienza en [objeto_starttheta], página 789 y termina en [objeto_endtheta], página 787. *num* debe ser un número entero positivo. Consulte también [objeto_starttheta], página 789 y [objeto_endtheta], página 787.

`track` [*track*, *posiciones*] [Opción de objeto]

posiciones deberá ser una lista de puntos. Cuando se oprime el botón de "play", la posición del objeto tomara secuencialmente los valores en la lista, a intervalos regulares de tiempo definidos por la opción [scene_tstep], página 785, dejando un rastro de la trayectoria del objeto. El botón de volver al comienzo puede usarse para recomenzar desde el principio de la lista, haciendo que la animación comience nuevamente cuando se presione otra vez el botón de play.

Ejemplo. Estos comandos muestran la trayectoria de una bola lanzada con velocidad inicial de 5 m/s, a un ángulo de 45 grados, cuando se puede ignorar la resistencia del aire:

```
(%i1) p: makelist ([0,4*t,4*t- 9.8*t^2/2], t, 0, 0.82, 0.01)$

(%i2) bola: [sphere, [radius,0.1], [color,red], [track,p]]$

(%i3) piso: [cube, [xlength,2], [ylength,4], [zlength,0.2],
      [position,0,1.5,-0.2],[color,green]]$

(%i4) scene (bola, piso)$
```

Consulte también [objeto_animation], página 786.

`xlength` [*xlength*, `altura`] [Opción de objeto]
Valor predefinido: `1`

Altura de un objeto de clase "cube", en la dirección x. *altura* debe ser un número positivo. Consulte también `[objeto_ylength], página 790` y `[objeto_zlength], página 790`.

`ylength` [*ylength*, `altura`] [Opción de objeto]
Valor predefinido: `1`

Altura de un objeto de clase "cube", en la dirección y. *altura* debe ser un número positivo. Consulte también `[objeto_xlength], página 790` y `[objeto_zlength], página 790`.

`zlength` [*zlength*, `altura`] [Opción de objeto]
Valor predefinido: `1`

Altura de un objeto de clase "cube", en la dirección z. *altura* debe ser un número positivo. Consulte también `[objeto_xlength], página 790` y `[objeto_ylength], página 790`.

`wireframe` [*wireframe*] [Opción de objeto]
Esta opción hace que en vez de que se muestre la superficie, se muestren únicamente las aristas de la triangulación usada para construir la superficie. **Ejemplo**: `[cube, [wireframe]]`

Consulte también `[objeto_surface], página 789` y `[objeto_points], página 788`.

50 ezunits

50.1 Introducción a ezunits

`ezunits` es un paquete para trabajar con magnitudes dimensionales, incluyendo algunas funciones para realizar análisis dimensional. `ezunits` puede hacer operaciones aritméticas con magnitudes dimensionales y efectuar conversiones entre unidades. Las unidades que se definen son las del Sistema Internacional (SI) y otras comunes en los Estados Unidos, siendo posible declarar otras nuevas.

Véase también `physical_constants`, una colección de constantes físicas.

Es necesario ejecutar primero `load("ezunits")` para utilizar este paquete. Con `demo(ezunits)` se podrán ver algunos ejemplos de utilización. La función `known_units` devuelve una lista con todas las unidades que están definidas y `display_known_unit_conversions` muestra las conversiones conocidas por el sistema en un formato de lectura sencilla.

Una expresión tal como a`` `b `` representa una magnitud dimensional, siendo `a` una magnitud adimensional y `b` las unidades. Se puede utilizar un símbolo como unidad, sin necesidad de declararlo como tal ni de que deba cumplir propiedades especiales. Tanto la magnitud como la unidad de una expresión de la forma a`` `b `` pueden extraerse invocando las funciones `qty` y `units`, respectivamente.

Una expresión tal como `a ` b `` c` convierte las unidades `b` en `c`. El paquete `ezunits` contiene funciones conversoras para unidades fundamentales del SI, unidades derivadas, así como algunas otras unidades ajenas al SI. Las conversiones entre unidades que no estén programadas en `ezunits` podrán declararse a posteriori. Las conversiones conocidas por `ezunits` están especificadas en la variable global `known_unit_conversions`, incluyendo tanto las ya declaradas por defecto como aquéllas introducidas por el usuario. Las conversiones para los productos, cocientes y potencias de unidades se derivan del conjunto de conversiones ya conocidas.

En general, Maxima prefiere números exactos (enteros o racionales) a inexactos (decimales en coma flotante), por lo que `ezunits` respetará los exactos cuando aparezcan en expresiones de magnitudes dimensionales. Todas las conversiones del paquete se han definido en términos de números exactos.

No hay un sistema de representación de unidades que se considere preferible, razón por la cual las unidades no se convierten a otras a menos que se indique de forma explícita. `ezunits` reconoce los prefijos m-, k-, M y G- para mili-, kilo-, mega- y giga-, respectivamente, tal como se utilizan en el SI; estos prefijos sólo se utilizan cuando así se indica de forma explícita.

Las operaciones aritméticas con magnitudes dimensionales se realizan de la forma convencional.

- `(x ` a) * (y ` b)` es igual a $(x * y)$`` `(a * b) ``.
- `(x ` a) + (y ` a)` es igual a $(x + y)$`` `a ``.
- `(x ` a)^y` es igual a x^y`` `a^y `` si `y` es adimensional.

`ezunits` no necesita que las unidades en una suma tengan las mismas dimensiones; estos términos serán sumados sin emitirse mensaje de error.

`ezunits` incluye funciones para el análisis dimensional elemental, como las dimensiones fundamentales, las unidades fundamentales de una magnitud dimensional o el cálculo de magnitudes adimensionales y unidades naturales. Las funciones de análisis dimensional son adaptaciones de funciones semejantes escritas por Barton Willis en otro paquete.

Con el fin de poder llevar a cabo análisis dimensionales, se mantiene una lista de dimensiones fundamentales y otra lista asociada de unidades fundamentales; por defecto, las dimensiones fundamentales son longitud, masa, tiempo, carga, temperatura y cantidad de materia, siendo las unidades fundamentales las propias del Sistema Internacional. En cualquier caso, es posible declarar otras dimensiones y unidades fundamentales.

50.2 Introducción a physical_constants

`physical_constants` contiene constantes físicas recomendadas por el CODATA 2006 (`http://physics.nist.gov/constants`). La instrucción `load("physical_constants")` carga este paquete en memoria junto con el propio `ezunits`, si éste no estaba previamente cargado.

Una constante física se representa por un símbolo con la propiedad de ser un valor constante. El valor constante es una magnitud dimensional en la sintaxis de `ezunits`. La función `constvalue` extrae el valor constante, el cual no es el valor ordinario del símbolo, por lo que las constantes físicas se mantienen inalteradas en las expresiones evaluadas hasta que sus valores sea extraído con la función `constvalue`.

`physical_constants` incluye cierta información adicional, como la descripción de cada constante, una estimación del error de su valor numérico y una propiedad para ser representada en TeX. Para identificar constantes físicas, cada símbolo tiene la propiedad `physical_constant`, de forma que `propvars(physical_constant)` muestra la lista de todas las constantes físicas.

`physical_constants` contiene las siguientes constantes:

`%c`	velocidad de la luz en el vacío
`%mu_0`	constante magnética
`%e_0`	constante eléctrica
`%Z_0`	impedancia característica del vacío
`%G`	constante gravitatoria de Newton
`%h`	constante de Planck
`%h_bar`	constante de Planck
`%m_P`	masa de Planck
`%T_P`	temperature de Planck
`%l_P`	longitud de Planck
`%t_P`	tiempo de Planck
`%%e`	carga elemental
`%Phi_0`	flujo magnético cuántico

`%G_0` conductancia cuántica

`%K_J` constante de Josephson

`%R_K` constante de von Klitzing

`%mu_B` magnetón de Bohr

`%mu_N` magnetón nuclear

`%alpha` constante de estructura fina

`%R_inf` constante de Rydberg

`%a_0` radio de Bohr

`%E_h` energía de Hartree

`%ratio_h_me`
cuanto de circulación

`%m_e` masa del electrón

`%N_A` número de Avogadro

`%m_u` constante de masa atómica atomic mass constant

`%F` constante de Faraday

`%R` constante molar de los gases

`%%k` constante de Boltzmann

`%V_m` volumen molar del gas ideal

`%n_0` constante de Loschmidt

`%ratio_S0_R`
constante de Sackur-Tetrode (constante de entropía absoluta)

`%sigma` constante de Stefan-Boltzmann

`%c_1` primera constante de radiación

`%c_1L` primera constante de radiación para radiancia espectral

`%c_2` segunda constante de radiación

`%b` Constante de la ley del desplazamiento de Wien

`%b_prime` Constante de la ley del desplazamiento de Wien

Ejemplos:

Lista de todos los símbolos que tienen la propiedad `physical_constant`.

```
(%i1) load ("physical_constants")$
(%i2) propvars (physical_constant);
(%o2) [%c, %mu_0, %e_0, %Z_0, %G, %h, %h_bar, %m_P, %T_P, %l_P,
%t_P, %%e, %Phi_0, %G_0, %K_J, %R_K, %mu_B, %mu_N, %alpha,
%R_inf, %a_0, %E_h, %ratio_h_me, %m_e, %N_A, %m_u, %F, %R, %%k,
%V_m, %n_0, %ratio_S0_R, %sigma, %c_1, %c_1L, %c_2, %b, %b_prime]
```

Propiedades de la constante física %c.

```
(%i1) load ("physical_constants")$
(%i2) constantp (%c);
(%o2)                          true
(%i3) get (%c, description);
(%o3)               speed of light in vacuum
(%i4) constvalue (%c);
                                        m
(%o4)                    299792458 ‘ -
                                        s
(%i5) get (%c, RSU);
(%o5)                            0
(%i6) tex (%c);
$$c$$
(%o6)                          false
```

Energía equivalente de una libra-masa. El símbolo %c se mantiene hasta que su valor es extraído con la llamada a la función constvalue.

```
(%i1) load ("physical_constants")$
(%i2) m * %c^2;
                               2
(%o2)                         %c  m
(%i3) %, m = 1 ‘ lbm;
                               2
(%o3)                         %c  ‘ lbm
(%i4) constvalue (%);
                                           2
                                       lbm m
(%o4)             89875517873681764 ‘ ------
                                         2
                                        s
(%i5) E : % ‘‘ J;
Computing conversions to base units; may take a moment.
                    366838848464007200
(%o5)               ------------------ ‘ J
                            9
(%i6) E ‘‘ GJ;
                     458548560580009
(%o6)                --------------- ‘ GJ
                        11250000
(%i7) float (%);
(%o7)             4.0759872051556356e+7 ‘ GJ
```

50.3 Funciones y variables para ezunits

‘ [Operador]

Operador de magnitud dimensional. Una expresión tal como a‘b representa una magnitud dimensional, siendo `a` una magnitud adimensional y `b` las unidades. Se puede utilizar un símbolo como unidad, sin necesidad de declararlo como tal ni de que deba cumplir propiedades especiales. Tanto la magnitud como la unidad de una expresión de la forma a‘b pueden extraerse invocando las funciones `qty` y `units`, respectivamente.

Las operaciones aritméticas con magnitudes dimensionales se realizan de la forma convencional.

- `(x ‘ a) * (y ‘ b)` es igual a $(x * y)$‘$(a * b)$.
- `(x ‘ a) + (y ‘ a)` es igual a $(x + y)$‘a.
- `(x ‘ a)^y` es igual a x^y‘a^y si `y` es adimensional.

`ezunits` no necesita que las unidades en una suma tengan las mismas dimensiones; estos términos serán sumados sin emitirse mensaje de error.

Para utilizar este operador ejecútese primero `load("ezunits")`.

Ejemplos:

Unidades del Sistema Internacional.

```
(%i1) load ("ezunits")$
(%i2) foo : 10 ‘ m;
(%o2)                          10 ‘ m
(%i3) qty (foo);
(%o3)                            10
(%i4) units (foo);
(%o4)                             m
(%i5) dimensions (foo);
(%o5)                          length
```

Unidades definidas por el usuario.

```
(%i1) load ("ezunits")$
(%i2) bar : x ‘ acre;
(%o2)                         x ‘ acre
(%i3) dimensions (bar);
                                     2
(%o3)                         length
(%i4) fundamental_units (bar);
                                    2
(%o4)                              m
```

Unidades ad hoc.

```
(%i1) load ("ezunits")$
(%i2) baz : 3 ‘ sheep + 8 ‘ goat + 1 ‘ horse;
(%o2)           8 ‘ goat + 3 ‘ sheep + 1 ‘ horse
(%i3) subst ([sheep = 3*goat, horse = 10*goat], baz);
(%o3)                        27 ‘ goat
```

```
(%i4) baz2 : 1000‘gallon/fortnight;
                                  gallon
(%o4)                    1000 ‘ ---------
                                 fortnight
(%i5) subst (fortnight = 14*day, baz2);
                            500   gallon
(%o5)                       --- ‘ ------
                             7     day
```

Operaciones aritméticas y magnitudes dimensionales.

```
(%i1) load ("ezunits")$
(%i2) 100 ‘ kg + 200 ‘ kg;
(%o2)                          300 ‘ kg
(%i3) 100 ‘ m^3 - 100 ‘ m^3;
                                   3
(%o3)                           0 ‘ m
(%i4) (10 ‘ kg) * (17 ‘ m/s^2);
                                  kg m
(%o4)                        170 ‘ ----
                                    2
                                   s
(%i5) (x ‘ m) / (y ‘ s);
                                 x   m
(%o5)                            - ‘ -
                                 y   s
(%i6) (a ‘ m)^2;
                                 2    2
(%o6)                           a  ‘ m
```

‘‘ [Operador]

Operador de conversión de unidades. Una expresión tal como a‘b‘‘c convierte las unidades `b` en `c`. El paquete `ezunits` contiene funciones conversoras para unidades fundamentales del SI, unidades derivadas, así como algunas otras unidades ajenas al SI. Las conversiones entre unidades que no estén programadas en `ezunits` podrán declararse a posteriori. Las conversiones conocidas por `ezunits` están especificadas en la variable global `known_unit_conversions`, incluyendo tanto las ya declaradas por defecto como aquéllas introducidas por el usuario. Las conversiones para los productos, cocientes y potencias de unidades se derivan del conjunto de conversiones ya conocidas.

No hay un sistema de representación de unidades que se considere preferible, razón por la cual las unidades no se convierten a otras a menos que se indique de forma explícita. Del mismo modo, `ezunits` no transforma prefijos (milli-, centi-, deci-, etc) a menos que se le indique.

Para utilizar este operador ejecútese primero `load("ezunits")`.

Ejemplos:

Conjunto de conversiones conocidas.

```
(%i1) load ("ezunits")$
```

```
(%i2) display2d : false$
(%i3) known_unit_conversions;
(%o3) {acre = 4840*yard^2,Btu = 1055*J,cfm = feet^3/minute,
       cm = m/100,day = 86400*s,feet = 381*m/1250,ft = feet,
       g = kg/1000,gallon = 757*l/200,GHz = 1000000000*Hz,
       GOhm = 1000000000*Ohm,GPa = 1000000000*Pa,
       GWb = 1000000000*Wb,Gg = 1000000*kg,Gm = 1000000000*m,
       Gmol = 1000000*mol,Gs = 1000000000*s,ha = hectare,
       hectare = 100*m^2,hour = 3600*s,Hz = 1/s,inch = feet/12,
       km = 1000*m,kmol = 1000*mol,ks = 1000*s,l = liter,
       lbf = pound_force,lbm = pound_mass,liter = m^3/1000,
       metric_ton = Mg,mg = kg/1000000,MHz = 1000000*Hz,
       microgram = kg/1000000000,micrometer = m/1000000,
       micron = micrometer,microsecond = s/1000000,
       mile = 5280*feet,minute = 60*s,mm = m/1000,
       mmol = mol/1000,month = 2629800*s,MOhm = 1000000*Ohm,
       MPa = 1000000*Pa,ms = s/1000,MWb = 1000000*Wb,
       Mg = 1000*kg,Mm = 1000000*m,Mmol = 1000000000*mol,
       Ms = 1000000*s,ns = s/1000000000,ounce = pound_mass/16,
       oz = ounce,Ohm = s*J/C^2,
       pound_force = 32*ft*pound_mass/s^2,
       pound_mass = 200*kg/441,psi = pound_force/inch^2,
       Pa = N/m^2,week = 604800*s,Wb = J/A,yard = 3*feet,
       year = 31557600*s,C = s*A,F = C^2/J,GA = 1000000000*A,
       GC = 1000000000*C,GF = 1000000000*F,GH = 1000000000*H,
       GJ = 1000000000*J,GK = 1000000000*K,GN = 1000000000*N,
       GS = 1000000000*S,GT = 1000000000*T,GV = 1000000000*V,
       GW = 1000000000*W,H = J/A^2,J = m*N,kA = 1000*A,
       kC = 1000*C,kF = 1000*F,kH = 1000*H,kHz = 1000*Hz,
       kJ = 1000*J,kK = 1000*K,kN = 1000*N,kOhm = 1000*Ohm,
       kPa = 1000*Pa,kS = 1000*S,kT = 1000*T,kV = 1000*V,
       kW = 1000*W,kWb = 1000*Wb,mA = A/1000,mC = C/1000,
       mF = F/1000,mH = H/1000,mHz = Hz/1000,mJ = J/1000,
       mK = K/1000,mN = N/1000,mOhm = Ohm/1000,mPa = Pa/1000,
       mS = S/1000,mT = T/1000,mV = V/1000,mW = W/1000,
       mWb = Wb/1000,MA = 1000000*A,MC = 1000000*C,
       MF = 1000000*F,MH = 1000000*H,MJ = 1000000*J,
       MK = 1000000*K,MN = 1000000*N,MS = 1000000*S,
       MT = 1000000*T,MV = 1000000*V,MW = 1000000*W,
       N = kg*m/s^2,R = 5*K/9,S = 1/Ohm,T = J/(m^2*A),V = J/C,
       W = J/s}
```

Converiones de unidades fundamentales.

```
(%i1) load ("ezunits")$
(%i2) 1 ‘ ft ‘‘ m;
Computing conversions to base units; may take a moment.
```

```
(%o2)                          ---- ‘ m
                               1250
(%i3) %, numer;
(%o3)                         0.3048 ‘ m
(%i4) 1 ‘ kg ‘‘ lbm;
                               441
(%o4)                          --- ‘ lbm
                               200
(%i5) %, numer;
(%o5)                         2.205 ‘ lbm
(%i6) 1 ‘ W ‘‘ Btu/hour;
                              720   Btu
(%o6)                         --- ‘ ----
                              211   hour
(%i7) %, numer;
                                                Btu
(%o7)                3.412322274881517 ‘ ----
                                                hour
(%i8) 100 ‘ degC ‘‘ degF;
(%o8)                          212 ‘ degF
(%i9) -40 ‘ degF ‘‘ degC;
(%o9)                        (- 40) ‘ degC
(%i10) 1 ‘ acre*ft ‘‘ m^3;
                            60228605349    3
(%o10)                      ----------- ‘ m
                             48828125
(%i11) %, numer;
                                                 3
(%o11)                 1233.48183754752 ‘ m
```

Transformando pies a metros y viceversa.

```
(%i1) load ("ezunits")$
(%i2) 100 ‘ m + 100 ‘ ft;
(%o2)                    100 ‘ m + 100 ‘ ft
(%i3) (100 ‘ m + 100 ‘ ft) ‘‘ ft;
                              163100
(%o3)                         ------ ‘ ft
                               381
(%i4) %, numer;
(%o4)                  428.0839895013123 ‘ ft
(%i5) (100 ‘ m + 100 ‘ ft) ‘‘ m;
                               3262
(%o5)                          ---- ‘ m
                                25
(%i6) %, numer;
(%o6)                         130.48 ‘ m
```

Análisis dimensional para encontrar dimensiones y unidades fundamentales.

```
(%i1) load ("ezunits")$
(%i2) foo : 1 ‘ acre * ft;
(%o2)                        1 ‘ acre ft
(%i3) dimensions (foo);
                                      3
(%o3)                          length
(%i4) fundamental_units (foo);
                                    3
(%o4)                              m
(%i5) foo ‘‘ m^3;
                            60228605349    3
(%o5)                       ----------- ‘ m
                             48828125
(%i6) %, numer;
                                                3
(%o6)                  1233.48183754752 ‘ m
```

Declaración de conversiones.

```
(%i1) load ("ezunits")$
(%i2) declare_unit_conversion (MMBtu = 10^6*Btu, kW = 1000*W);
(%o2)                             done
(%i3) declare_unit_conversion (kWh = kW*hour, MWh = 1000*kWh,
                                bell = 1800*s);
(%o3)                             done
(%i4) 1 ‘ kW*s ‘‘ MWh;
Computing conversions to base units; may take a moment.
                                 1
(%o4)                        ------- ‘ MWh
                             3600000
(%i5) 1 ‘ kW/m^2 ‘‘ MMBtu/bell/ft^2;
                          1306449      MMBtu
(%o5)                     ---------- ‘ --------
                          8242187500          2
                                     bell ft
```

`constvalue` (*x*) [Función]
`declare_constvalue` (*a*, *x*) [Función]
`remove_constvalue` (*a*) [Función]

Devuelve la constante declarada para un símbolo. Los valores constantes se declaran con `declare_constvalue`.

Los valores constantes reconocidos por `constvalue` son distintos de los valores declarados por `numerval` y reconocidos por `constantp`.

El paquete `physical_units` declara los valores constantes de las constantes físicas.

`remove_constvalue` deshace la acción de `declare_constvalue`.

Para utilizar estas funciones ejecútese primero `load("ezunits")`.

Ejemplos:

Valor de una constante física.

```
(%i1) load ("physical_constants")$
(%i2) constvalue (%G);
                                      3
                                     m
(%o2)                    6.67428 ‘ -----
                                        2
                                    kg s
(%i3) get ('%G, 'description);
(%o3)           Newtonian constant of gravitation
```

Declarando una nueva constante.

```
(%i1) load ("ezunits")$
(%i2) declare_constvalue (FOO, 100 ‘ lbm / acre);
                                  lbm
(%o2)                      100 ‘ ----
                                 acre
(%i3) FOO * (50 ‘ acre);
(%o3)                      50 FOO ‘ acre
(%i4) constvalue (%);
(%o4)                       5000 ‘ lbm
```

`units` (*x*) [Función]

`declare_units` (*a*, *u*) [Función]

Devuelve las unidades de la magnitud dimensional *x*, o 1 en caso de que *x* sea adimensional.

x puede ser una expresión literal dimensional *a*‘*b*, un símbolo con unidades declaradas por medio de `declare_units`, o una expresión que contenga cualquiera o ambos de los anteriores.

`declare_constvalue` declara que `units(a)` debe devolver *u*, siendo *u* una expresión.

Para utilizar estas funciones ejecútese primero `load("ezunits")`.

Ejemplos:

`units` aplicado a expresiones dimensionales literales.

```
(%i1) load ("ezunits")$
(%i2) foo : 100 ‘ kg;
(%o2)                              100 ‘ kg
(%i3) bar : x ‘ m/s;
                                         m
(%o3)                                x ‘ -
                                         s
(%i4) units (foo);
(%o4)                                 kg
(%i5) units (bar);
                                       m
(%o5)                                  -
                                       s
```

```
(%i6) units (foo * bar);
                                        kg m
(%o6)                                   ----
                                         s
(%i7) units (foo / bar);
                                        kg s
(%o7)                                   ----
                                         m
(%i8) units (foo^2);
                                          2
(%o8)                                    kg
```

`units` aplicado a símbolos con unidades declaradas.

```
(%i1) load ("ezunits")$
(%i2) linenum:0;
(%o0)                                     0
(%i1) units (aa);
(%o1)                                     1
(%i2) declare_units (aa, J);
(%o2)                                     J
(%i3) units (aa);
(%o3)                                     J
(%i4) units (aa^2);
                                          2
(%o4)                                    J
(%i5) foo : 100 ‘ kg;
(%o5)                                100 ‘ kg
(%i6) units (aa * foo);
(%o6)                                  kg J
```

`qty` (*x*) [Función]

`declare_qty` (`a`, *x*) [Función]

`qty` devuelve la parte adimensional de la magnitud dimensional *x*, o *x*, si *x* es adimensional. *x* puede ser una expresión literal dimensional *a*‘*b*, un símbolo con unidades declaradas o una expresión que contenga cualquiera o ambos de los anteriores.

`declare_qty` declara que `qty(a)` debe devolver *x*, siendo *x* una magnitud dimensional.

Para utilizar estas funciones ejecútese primero `load("ezunits")`.

Ejemplos:

`qty` aplicado a expresiones dimensionales literales.

```
(%i1) load ("ezunits")$
(%i2) foo : 100 ‘ kg;
(%o2)                          100 ‘ kg
(%i3) qty (foo);
(%o3)                             100
(%i4) bar : v ‘ m/s;
                                    m
```

```
(%o4)                          v ` -
                                   s
(%i5) foo * bar;
                                   kg m
(%o5)                    100 v ` ----
                                    s
(%i6) qty (foo * bar);
(%o6)                          100 v
```

`qty` aplicado a símbolos con unidades declaradas.

```
(%i1) load ("ezunits")$
(%i2) declare_qty (aa, xx);
(%o2)                           xx
(%i3) qty (aa);
(%o3)                           xx
(%i4) qty (aa^2);
                                  2
(%o4)                           xx
(%i5) foo : 100 ` kg;
(%o5)                       100 ` kg
(%i6) qty (aa * foo);
(%o6)                        100 xx
```

`unitp` (*x*) [Función]

Devuelve `true` si *x* es una expresión dimensional literal, un símbolo declarado como dimensional o una expresión en la que su operador principal ha sido declarado como dimensional. En cualquier otro caso, `unitp` devuelve `false`.

Para utilizar esta función ejecútese primero `load("ezunits")`.

Ejemplos:

`unitp` aplicado a expresiones dimensionales literales.

```
(%i1) load ("ezunits")$
(%i2) unitp (100 ` kg);
(%o2)                          true
```

`unitp` applied to a symbol declared dimensional.

```
(%i1) load ("ezunits")$
(%i2) unitp (foo);
(%o2)                          false
(%i3) declare (foo, dimensional);
(%o3)                          done
(%i4) unitp (foo);
(%o4)                          true
```

`unitp` aplicado a una expresión en la que el operador principal se declara dimensional.

```
(%i1) load ("ezunits")$
(%i2) unitp (bar (x, y, z));
(%o2)                          false
(%i3) declare (bar, dimensional);
```

```
(%o3)                          done
(%i4) unitp (bar (x, y, z));
(%o4)                          true
```

`declare_unit_conversion` (*u* = *v*, ...) [Función]

Añade las ecuaciones $u = v$, ... a la lista de conversiones de unidades conocidas por el operador de conversión ‘‘. u y v son términos multiplicativos en las que las variables son unidades o expresiones dimensionales literales.

De momento, es imperativo expresar las conversiones de forma que el miembro izquierdo de cada ecuación sea una unidad simple (en opsición a una expresión multiplicativa) o una expresión dimensional literal con la cantidad igual a 1 y con unidad simple. Está previsto eliminar esta restricción en versiones futuras.

`known_unit_conversions` es la lista de conversiones de unidades conocidas.

Para utilizar esta función ejecútese primero `load("ezunits")`.

Ejemplos:

Conversión de unidades expresadas por ecuaciones con términos multiplicativos.

```
(%i1) load ("ezunits")$
(%i2) declare_unit_conversion (nautical_mile = 1852 * m,
                               fortnight = 14 * day);
(%o2)                          done
(%i3) 100 ‘ nautical_mile / fortnight ‘‘ m/s;
Computing conversions to base units; may take a moment.
                             463    m
(%o3)                        ---- ‘ -
                             3024   s
```

Conversión de unidades expresadas por ecuaciones con expresiones dimensionales literales.

```
(%i1) load ("ezunits")$
(%i2) declare_unit_conversion (1 ‘ fluid_ounce = 2 ‘ tablespoon);
(%o2)                          done
(%i3) declare_unit_conversion (1 ‘ tablespoon = 3 ‘ teaspoon);
(%o3)                          done
(%i4) 15 ‘ fluid_ounce ‘‘ teaspoon;
Computing conversions to base units; may take a moment.
(%o4)                     90 ‘ teaspoon
```

`declare_dimensions` (*a_1*, *d_1*, ..., *a_n*, *d_n*) [Función]

`remove_dimensions` (*a_1*, ..., *a_n*) [Función]

`declare_dimensions` declara *a_1*, ..., *a_n* con las dimensiones *d_1*, ..., *d_n*, respectivamente.

Cada *a_k* es un símbolo o lista de símbolos. En caso de ser una lista, cada símbolo en *a_k* se declara de dimensión *d_k*.

`remove_dimensions` invierte el efecto de `declare_dimensions`.

Ejecútese `load("ezunits")` para hacer uso de estas funciones.

Ejemplos:

```
(%i1) load ("ezunits") $
(%i2) declare_dimensions ([x, y, z], length, [t, u], time);
(%o2)                          done
(%i3) dimensions (y^2/u);
                                    2
                               length
(%o3)                          -------
                                time
(%i4) fundamental_units (y^2/u);
0 errors, 0 warnings
                                 2
                                m
(%o4)                            --
                                 s
```

`declare_fundamental_dimensions` (*d_1*, *d_2*, *d_3*, ...) [Función]
`remove_fundamental_dimensions` (*d_1*, *d_2*, *d_3*, ...) [Función]
`fundamental_dimensions` [Variable global]

`declare_fundamental_dimensions` declara dimensiones fundamentales. Los símbolos *d_1*, *d_2*, *d_3*, ... se añaden a la lista de dimensiones fundamentales si no están ya presentes en la lista.

`remove_fundamental_dimensions` invierte el efecto de `declare_fundamental_dimensions`.

`fundamental_dimensions` es la lista de dimensiones fundamentales. Por defecto, la lista comprende algunas dimensiones físicas.

Ejecútese `load("ezunits")` para hacer uso de estas funciones.

Ejemplos:

```
(%i1) load ("ezunits") $
(%i2) fundamental_dimensions;
(%o2) [length, mass, time, current, temperature, quantity]
(%i3) declare_fundamental_dimensions (money, cattle, happiness);
(%o3)                          done
(%i4) fundamental_dimensions;
(%o4) [length, mass, time, current, temperature, quantity,
                                        money, cattle, happiness]
(%i5) remove_fundamental_dimensions (cattle, happiness);
(%o5)                          done
(%i6) fundamental_dimensions;
(%o6) [length, mass, time, current, temperature, quantity, money]
```

`declare_fundamental_units` (*u_1*, *d_1*, ..., *u_n*, *d_n*) [Función]
`remove_fundamental_units` (*u_1*, ..., *u_n*) [Función]

`declare_fundamental_units` declara *u_1*, ..., *u_n* de dimensiones *d_1*, ..., *d_n*, respectivamente. Todos los argumentos deben símbolos.

Tras la llamada a `declare_fundamental_units`, `dimensions(u_k)` devuelve *d_k* para cada argumento *u_1*, ..., *u_n*, y `fundamental_units(d_k)` devuelve *u_k* para cada *d_1*, ..., *d_n*.

`remove_fundamental_units` invierte el efecto de `declare_fundamental_units`.

Ejecútese `load("ezunits")` para hacer uso de estas funciones.

Ejemplos:

```
(%i1) load ("ezunits") $
(%i2) declare_fundamental_dimensions (money, cattle, happiness);
(%o2)                          done
(%i3) declare_fundamental_units (dollar, money, goat, cattle,
                                   smile, happiness);
(%o3)                 [dollar, goat, smile]
(%i4) dimensions (100 ‘ dollar/goat/km^2);
                                money
(%o4)                     --------------
                                      2
                           cattle length
(%i5) dimensions (x ‘ smile/kg);
                              happiness
(%o5)                         ---------
                                mass
(%i6) fundamental_units (money*cattle/happiness);
0 errors, 0 warnings
                             dollar goat
(%o6)                        -----------
                                smile
```

`dimensions (x)` [Función]
`dimensions_as_list (x)` [Función]

`dimensions` devuelve las dimensiones de la magnitud dimensional *x* en forma de expresión que contiene productos y potencias de dimensiones fundamentales.

`dimensions_as_list` devuelve las dimensiones de la magnitud dimensional *x* en forma de lista, cuyos elementos indican las potencias de las dimensiones fundamentales correspondientes.

Para utilizar estas funciones ejecútese primero `load("ezunits")`.

Ejemplos:

```
(%i1) load ("ezunits")$
(%i2) dimensions (1000 ‘ kg*m^2/s^3);
                                2
                           length  mass
(%o2)                      ------------
                                 3
                              time
(%i3) declare_units (foo, acre*ft/hour);
                              acre ft
```

```
(%o3)                          -------
                                hour
(%i4) dimensions (foo);
                                     3
                               length
(%o4)                          -------
                                time
(%i1) load ("ezunits")$
(%i2) fundamental_dimensions;
(%o2)  [length, mass, time, charge, temperature, quantity]
(%i3) dimensions_as_list (1000 ‘ kg*m^2/s^3);
(%o3)                 [2, 1, - 3, 0, 0, 0]
(%i4) declare_units (foo, acre*ft/hour);
                               acre ft
(%o4)                          -------
                                hour
(%i5) dimensions_as_list (foo);
(%o5)                 [3, 0, - 1, 0, 0, 0]
```

`fundamental_units` (*x*) [Función]

`fundamental_units` () [Función]

`fundamental_units(x)` devuelve las unidades asociadas a las dimensiones fundamentales de *x*, tal como queda determinada por `dimensions(x)`.

x puede ser una expresión literal dimensional *a*‘*b*, un símbolo con unidades declaradas a través de `declare_units` o una expresión que contenga a ambos.

`fundamental_units()` devuelve una lista con las unidades fundamentales conocidas, tal como fueron declaradas por `declare_fundamental_units`.

Para utilizar esta función ejecútese primero `load("ezunits")`.

Ejemplos:

```
(%i1) load ("ezunits")$
(%i2) fundamental_units ();
(%o2)                 [m, kg, s, A, K, mol]
(%i3) fundamental_units (100 ‘ mile/hour);
                                m
(%o3)                           -
                                s
(%i4) declare_units (aa, g/foot^2);
                                g
(%o4)                          -----
                                  2
                               foot
(%i5) fundamental_units (aa);
                                kg
(%o5)                           --
                                 2
                                m
```

`dimensionless` (*L*) [Función]

Devuelve una expresión sin dimensiones que se puede formar a partir de una lista *L* de cantidades dimensionales

Para utilizar esta función ejecútese primero `load("ezunits")`.

Ejemplos:

```
(%i1) load ("ezunits") $
(%i2) dimensionless ([x ‘ m, y ‘ m/s, z ‘ s]);
0 errors, 0 warnings
0 errors, 0 warnings
                                 y z
(%o2)                           [---]
                                  x
```

Cantidades adimensionales obtenidas a partir de cantidades físicas. Nótese que el primer elemento de la lista es proporcional a la constante de estructura fina.

```
(%i1) load ("ezunits") $
(%i2) load ("physical_constants") $
(%i3) dimensionless([%h_bar, %m_e, %m_P, %%e, %c, %e_0]);
0 errors, 0 warnings
0 errors, 0 warnings
                                  2
                               %%e        %m_e
(%o3)                  [--------------, ----]
                        %c %e_0 %h_bar  %m_P
```

`natural_unit` (*expr*, [*v_1*, ..., *v_n*]) [Función]

Busca los exponentes *e_1*, ..., *e_n* tales que `dimension(`*expr*`) = dimension(`*v_1*`^`*e_1* ... *v_n*`^`*e_n*`)`.

Para utilizar esta función ejecútese primero `load("ezunits")`.

51 f90

51.1 Funciones y variables para f90

f90 (*expr_1*, ..., *expr_n*) [Función]

Imprime una o más expresiones *expr_1*, ..., *expr_n* como un programa Fortran 90. El programa se obtiene a través de la salida estándar.

La función `f90` imprime su salida en el llamado formato libre de Fortran 90: no se presta atención alguna a las posiciones de caracteres respecto de las columnas y los renglones largos se dividen a un ancho fijo con el carácter `&` indicando continuación de código.

Ejecútese `load("f90")` antes de utilizar esta función.

Véase también `fortran`.

Ejemplos:

```
(%i1) load ("f90")$
(%i2) foo : expand ((xxx + yyy + 7)^4);
         4            3         3        2    2             2
(%o2) yyy  + 4 xxx yyy  + 28 yyy  + 6 xxx  yyy  + 84 xxx yyy
          2        3             2
 + 294 yyy  + 4 xxx  yyy + 84 xxx  yyy + 588 xxx yyy + 1372 yyy
      4         3          2
 + xxx  + 28 xxx  + 294 xxx  + 1372 xxx + 2401
(%i3) f90 ('foo = foo);
foo = yyy**4+4*xxx*yyy**3+28*yyy**3+6*xxx**2*yyy**2+84*xxx*yyy**2&
+294*yyy**2+4*xxx**3*yyy+84*xxx**2*yyy+588*xxx*yyy+1372*yyy+xxx**&
4+28*xxx**3+294*xxx**2+1372*xxx+2401
(%o3)                          false
```

Expresiones múltiples. Captura de la salida estándar a un fichero por medio de la función `with_stdout`.

```
(%i1) load ("f90")$
(%i2) foo : sin (3*x + 1) - cos (7*x - 2);
(%o2)              sin(3 x + 1) - cos(7 x - 2)
(%i3) with_stdout ("foo.f90",
                   f90 (x=0.25, y=0.625, 'foo=foo, 'stop, 'end));
(%o3)                          false
(%i4) printfile ("foo.f90");
x = 0.25
y = 0.625
foo = sin(3*x+1)-cos(7*x-2)
stop
end
(%o4)                         foo.f90
```

52 finance

52.1 Introducción a finance

Este es el Paquete "Finance" (Ver 0.1).

En todas las funciones, *rate* es la tasa de interés compuesto, *num* es el número de periodos y debe ser positivo, y *flow* se refiere al flujo de caja; entonces, si se tiene un egreso el flujo es negativo y para un ingreso un valor positivo.

Note que antes de usar las funciones definidas en este paquete, debe cargarla escribiendo `load("finance")$`.

Autor: Nicolás Guarin Zapata.

52.2 Funciones y Variables para finance

`days360` (*año1*,*mes1*,*dia1*,*año2*,*mes2*,*dia2*) [Function]

Calcula la distancia entre 2 fechas, asumiendo años de 360 dias y meses de 30 días.

Ejemplo:

```
(%i1) load("finance")$
(%i2) days360(2008,12,16,2007,3,25);
(%o2)                        - 621
```

`fv` (*rate*,*PV*,*num*) [Function]

Calcular el Valor Futuro a partir de uno en el Presente para una tasa de interés dada. *rate* es la tasa de interés, *PV* es el valor prestente y *num* es el número de periodos.

Ejemplo:

```
(%i1) load("finance")$
(%i2) fv(0.12,1000,3);
(%o2)                       1404.928
```

`pv` (*rate*,*FV*,*num*) [Function]

Calcula el valor actual de un valor futuro dada la tasa de interés. *rate* es la tasa de interés, *FV* es el valor futuro y *num* es el número de periodos.

Ejemplo:

```
(%i1) load("finance")$
(%i2) pv(0.12,1000,3);
(%o2)                 711.7802478134108
```

`graph_flow` (*val*) [Function]

Grafica el flujo de caja en una línea de tiempo, los valores positivos están en azul y hacia arriba; los negativos están en rojo y hacia abajo. La dirección del flujo está dada por el signo de los valores. *val* es una lista de los valores del flujo de caja.

Ejemplo:

```
(%i1) load("finance")$
(%i2) graph_flow([-5000,-3000,800,1300,1500,2000])$
```

annuity_pv (*rate*,*PV*,*num*) [Function]

Calcula una anualidad conociendo el valor presente (tipo deuda), para unos pagos periódicos y constantes. *rate* es la tasa de interés, *PV* es el valor presente y *num* es el número de periodos.

Ejemplo:

```
(%i1) load("finance")$
(%i2) annuity_pv(0.12,5000,10);
(%o2)                884.9208207992202
```

annuity_fv (*rate*,*FV*,*num*) [Function]

Calcula una anualidad conociendo el valor deseado (valor futuro), para una serie de pagos periódicos y constantes. *rate* es la tasa de interés, *FV* es el valor futuroe y *num* es el número de periodos.

Ejemplo:

```
(%i1) load("finance")$
(%i2) annuity_fv(0.12,65000,10);
(%o2)                3703.970670389863
```

geo_annuity_pv (*rate*,*growing_rate*,*PV*,*num*) [Function]

Calcula una anualidad conociendo el valor presente (tipo deuda) en una serie de pagos periodicos crecientes. *rate* es la tasa de interés, *growing_rate* es el crecimiento de los pagos, *PV* es el valor presente, y *num* es el número de periodos.

Ejemplo:

```
(%i1) load("finance")$
(%i2) geo_annuity_pv(0.14,0.05,5000,10);
(%o2)                802.6888176505123
```

geo_annuity_fv (*rate*,*growing_rate*,*FV*,*num*) [Function]

Calcular una anualidad conociendo el valor deseado (valor futuro) en una serie de pagos periodicos crecientes. *rate* es la tasa de interés, *growing_rate* es el crecimiento de los pagos, *FV* es el valor futuro, y *num* es el número de periodos.

Ejemplo:

```
(%i1) load("finance")$
(%i2) geo_annuity_fv(0.14,0.05,5000,10);
(%o2)                216.5203395312695
```

amortization (*rate*,*amount*,*num*) [Function]

La tabla de amortización determinada por una tasa. Siendo *rate* es la tasa de interés, *amount* es el valor de la deuda, and *num* es el número de periodos.

Ejemplo:

```
(%i1) load("finance")$
(%i2) amortization(0.05,56000,12)$
      "n"    "Balance"     "Interest"   "Amortization"  "Payment"
    0.000     56000.000         0.000         0.000         0.000
    1.000     52481.777      2800.000      3518.223      6318.223
    2.000     48787.643      2624.089      3694.134      6318.223
```

```
    3.000     44908.802      2439.382      3878.841      6318.223
    4.000     40836.019      2245.440      4072.783      6318.223
    5.000     36559.597      2041.801      4276.422      6318.223
    6.000     32069.354      1827.980      4490.243      6318.223
    7.000     27354.599      1603.468      4714.755      6318.223
    8.000     22404.106      1367.730      4950.493      6318.223
    9.000     17206.088      1120.205      5198.018      6318.223
   10.000     11748.170       860.304      5457.919      6318.223
   11.000      6017.355       587.408      5730.814      6318.223
   12.000         0.000       300.868      6017.355      6318.223
```

arit_amortization (*rate*,*increment*,*amount*,*num*) [Function]

La tabla de amortización determinada por una tasa específica y unos pagos crecientes se puede hallar con `arit_amortization`. Nótese que los pagos no son constantes, estos presentan un crecimiento aritmético, el incremento es la diferencia entre dos filas consecutivas en la columna "Payment". *rate* es la tasa de interés, *increment* es el incremento, *amount* es el valor de la deuda, and *num* es el número de periodos.

Ejemplo:

```
(%i1) load("finance")$
(%i2) arit_amortization(0.05,1000,56000,12)$
      "n"    "Balance"     "Interest"   "Amortization"  "Payment"
    0.000     56000.000         0.000         0.000         0.000
    1.000     57403.679      2800.000     -1403.679      1396.321
    2.000     57877.541      2870.184      -473.863      2396.321
    3.000     57375.097      2893.877       502.444      3396.321
    4.000     55847.530      2868.755      1527.567      4396.321
    5.000     53243.586      2792.377      2603.945      5396.321
    6.000     49509.443      2662.179      3734.142      6396.321
    7.000     44588.594      2475.472      4920.849      7396.321
    8.000     38421.703      2229.430      6166.892      8396.321
    9.000     30946.466      1921.085      7475.236      9396.321
   10.000     22097.468      1547.323      8848.998     10396.321
   11.000     11806.020      1104.873     10291.448     11396.321
   12.000        -0.000       590.301     11806.020     12396.321
```

geo_amortization (*rate*,*growing_rate*,*amount*,*num*) [Function]

La tabla de amortización determinada por la tasa, el valor de la deuda, y el número de periodos se puede hallar con `geo_amortization`. Nótese que los pagos no son constantes, estos presentan un crecimiento geométrico, growin_rate es entonces el cociente entre dos filas consecutivas de la columna "Payment". *rate* es la tasa de interés, *growing_rate* es el crecimeinto de los pagos, *amount* es el valor de la deuda, y *num* es el numero de periodos.

Ejemplo:

```
(%i1) load("finance")$
(%i2) geo_amortization(0.05,0.03,56000,12)$
      "n"    "Balance"     "Interest"   "Amortization"  "Payment"
    0.000     56000.000         0.000         0.000         0.000
```

```
 1.000     53365.296      2800.000      2634.704      5434.704
 2.000     50435.816      2668.265      2929.480      5597.745
 3.000     47191.930      2521.791      3243.886      5765.677
 4.000     43612.879      2359.596      3579.051      5938.648
 5.000     39676.716      2180.644      3936.163      6116.807
 6.000     35360.240      1983.836      4316.475      6300.311
 7.000     30638.932      1768.012      4721.309      6489.321
 8.000     25486.878      1531.947      5152.054      6684.000
 9.000     19876.702      1274.344      5610.176      6884.520
10.000     13779.481       993.835      6097.221      7091.056
11.000      7164.668       688.974      6614.813      7303.787
12.000         0.000       358.233      7164.668      7522.901
```

`saving` (*rate*,*amount*,*num*) [Function]

La tabla que presenta los valores para un ahorro constante y periódico se pueden hallar con `saving`. *amount* representa la cantidad deseada y *num* el número de periodos durante los que se ahorrará.

Ejemplo:

```
(%i1) load("finance")$
(%i2) saving(0.15,12000,15)$
      "n"    "Balance"     "Interest"   "Payment"
     0.000         0.000         0.000         0.000
     1.000       252.205         0.000       252.205
     2.000       542.240        37.831       252.205
     3.000       875.781        81.336       252.205
     4.000      1259.352       131.367       252.205
     5.000      1700.460       188.903       252.205
     6.000      2207.733       255.069       252.205
     7.000      2791.098       331.160       252.205
     8.000      3461.967       418.665       252.205
     9.000      4233.467       519.295       252.205
    10.000      5120.692       635.020       252.205
    11.000      6141.000       768.104       252.205
    12.000      7314.355       921.150       252.205
    13.000      8663.713      1097.153       252.205
    14.000     10215.474      1299.557       252.205
    15.000     12000.000      1532.321       252.205
```

`npv` (*rate*,*val*) [Function]

Calcular el valor presente neto de una serie de valores para evaluar la viabilidad de un proyecto. *flowValues* es una lista con los valores para cada periodo.

Ejemplo:

```
(%i1) load("finance")$
(%i2) npv(0.25,[100,500,323,124,300]);
(%o2)                714.4703999999999
```

`irr` (*val*,*I0*) [Function]

Tasa interna de retorno (en inglés Internal Rate of Return - IRR), es el valor de tasa que hace que el Valor Presente Neto (NPV) sea cero. *flowValues* los valores para cada periodo (para periodos mayores a 0) y *I0* el valor para el periodo cero.

Ejemplo:

```
(%i1) load("finance")$
(%i2) res:irr([-5000,0,800,1300,1500,2000],0)$
(%i3) rhs(res[1][1]);
(%o3)                .03009250374237132
```

`benefit_cost` (*rate*,*input*,*output*) [Function]

Calcular la relación Beneficio/Costo, Beneficio es el Valor Presente Neto (NPV) de los flujos de caja positivos (inputs), y Costo es el Valor Presente Neto de los flujos de caja negativos (outputs). Nótese que si se desea tener un valor de cero para un periodo especifico, esta entrada/salida debe indicarse como cero para ese periodo. *rate* es la tasa de interés, *input* es una lista con los ingresos, y *output* es una lista con los egresos.

Ejemplo:

```
(%i1) load("finance")$
(%i2) benefit_cost(0.24,[0,300,500,150],[100,320,0,180]);
(%o2)               1.427249324905784
```

53 fractals

53.1 Introducción a fractals

Este paquete define algunos fractales:

- con IFS (Iterated Function System) aleatorias: triángulo de Sierpinsky, un árbol y un helecho.

- Fractales complejos: conjuntos de Mandelbrot y de Julia.

- Copos de Koch.

- Funciones de Peano: funciones de Sierpinski y Hilbert.

Autor: José Rammírez Labrador.

Para preguntas, sugerencias y fallos,

pepe DOT ramirez AAATTT uca DOT es

53.2 Definiciones para IFS fractals

Algunos fractales se pueden generar por medio de la aplicación iterativa de transformaciones afines contractivas de forma aleatoria; véase

Hoggar S. G., "Mathematics for computer graphics", Cambridge University Press 1994.

Definimos una lista con varias transformaciones afines contractivas, luego las vamos seleccionando de forma aleatoria y recursiva. La probabilidad de selección de una transformación debe estar relacionada con la razón de contracción.

Se pueden cambiar las transformaciones y encontrar nuevos fractales.

`sierpinskiale` (*n*) [Función]

Triángulo de Sierpinski: 3 aplicaciones contractivas; constante de contracción de 0.5 y traslaciones. Todas las aplicaciones tienen la misma constante de contracción. El argumento *n* debe ser suficientemente alto, 10000 o mayor.

Ejemplo:

```
(%i1) load("fractals")$
(%i2) n: 10000$
(%i3) plot2d([discrete,sierpinskiale(n)], [style,dots])$
```

`treefale` (*n*) [Función]

3 aplicaciones contractivas, todas ellas con el mismo coeficiente de contracción. El argumento *n* debe ser suficientemente alto, 10000 o mayor.

Ejemplo:

```
(%i1) load("fractals")$
(%i2) n: 10000$
(%i3) plot2d([discrete,treefale(n)], [style,dots])$
```

`fernfale` (*n*) [Función]

4 aplicaciones contractivas, cuyas probabilidades de selección deben estar relacionadas con su constante de contracción. El argumento *n* debe ser suficientemente alto, 10000 o mayor.

Ejemplo:

```
(%i1) load("fractals")$
(%i2) n: 10000$
(%i3) plot2d([discrete,fernfale(n)], [style,dots])$
```

53.3 Definiciones para fractales complejos

`mandelbrot_set` (*x*, *y*) [Función]

Conjunto de Mandelbrot.

Esta función debe realizar muchas operaciones y puede tardar bastante tiempo en ejecutarse, tiempo que también depende del número de puntos de la malla.

Ejemplo:

```
(%i1) load("fractals")$
(%i2) plot3d (mandelbrot_set, [x, -2.5, 1], [y, -1.5, 1.5],
                [gnuplot_preamble, "set view map"],
                [gnuplot_pm3d, true],
                [grid, 150, 150])$
```

`julia_set` (*x*, *y*) [Función]

Conjuntos de Julia.

Esta función debe realizar muchas operaciones y puede tardar bastante tiempo en ejecutarse, tiempo que también depende del número de puntos de la malla.

Ejemplo:

```
(%i1) load("fractals")$
(%i2) plot3d (julia_set, [x, -2, 1], [y, -1.5, 1.5],
                [gnuplot_preamble, "set view map"],
                [gnuplot_pm3d, true],
                [grid, 150, 150])$
```

Véase también `julia_parameter`.

`julia_parameter` [Variable opcional]

Valor por defecto: `%i`

Parámetro complejo para fractales de Julia. Su valor por defecto es `%i`, y otros que se sugieren son: `-.745+%i*.113002`, `-.39054-%i*.58679`, `-.15652+%i*1.03225`, `-.194+%i*.6557` y `.011031-%i*.67037`.

`julia_sin` (*x*, *y*) [Función]

Mientras que la función `julia_set` implementa la transformación `julia_parameter+z^2`, la función `julia_sin` implementa `julia_parameter*sin(z)`. Véase el código fuente para más detalles.

Este programa es lento porque calcula muchos senos; el tiempo de ejecución también depende del número de puntos de la malla.

Ejemplo:

```
(%i1) load("fractals")$
(%i2) julia_parameter:1+.1*%i$
```

```
(%i3) plot3d (julia_sin, [x, -2, 2], [y, -3, 3],
                [gnuplot_preamble, "set view map"],
                [gnuplot_pm3d, true],
                [grid, 150, 150])$
```

Véase también `julia_parameter`.

53.4 Definiciones para cops de Koch

`snowmap` (*ent*, *nn*) [Función]

Copos de Koch. La función `snowmap` dibuja el copo de Koch sobre los vértices de un polígono convexo inicial del plano complejo. La orientación del polígono es importante. El argumento *nn* es el número de recursividades de la transformación de Koch, el cual debe ser pequeño (5 o 6).

Ejemplos:

```
(%i1) load("fractals")$
(%i2) plot2d([discrete,
              snowmap([1,exp(%i*%pi*2/3),exp(-%i*%pi*2/3),1],4)])$
(%i3) plot2d([discrete,
              snowmap([1,exp(-%i*%pi*2/3),exp(%i*%pi*2/3),1],4)])$
(%i4) plot2d([discrete, snowmap([0,1,1+%i,%i,0],4)])$
(%i5) plot2d([discrete, snowmap([0,%i,1+%i,1,0],4)])$
```

53.5 Definiciones para curvas de Peano

Funciones continuas que cubren un área. Aviso: el número de puntos crece exponencialmente con *n*.

`hilbertmap` (*nn*) [Función]

Curva de Hilbert. El argumento *nn* debe ser pequeño (por ejemplo, 5). Maxima se puede detener si *nn* es 7 o mayor.

Ejemplo:

```
(%i1) load("fractals")$
(%i2) plot2d([discrete,hilbertmap(6)])$
```

`sierpinskimap` (*nn*) [Función]

Curva de Sierpinski. El argumento *nn* debe ser pequeño (por ejemplo, 5). Maxima se puede detener si *nn* es 7 o mayor.

Ejemplo:

```
(%i1) load("fractals")$
(%i2) plot2d([discrete,sierpinskimap(6)])$
```

54 ggf

54.1 Funciones y variables para ggf

`GGFINFINITY` [Variable opcional]

Valor por defecto: 3

Variable opcional para la función `ggf`.

Cuando se calcula la fracción continua de la función generatriz, si un cociente parcial tiene grado estrictamente mayor que *GGFINFINITY* será descartado y la convergencia alcanzada hasta ese momento será considerada como exacta para la función generatriz. Lo más frecuente es que el grado de todos los cocientes parciales sea 0 ó 1, de modo que si se utiliza un valor mayor se deberán dar más términos para conseguir un cálculo más exacto.

Véase también `ggf`.

`GGFCFMAX` [Variable opcional]

Valor por defeco: 3

Variable opcional para la función `ggf`.

Cuando se calcula la fracción continua de la función generatriz, si no se ha encontrado un resultado aceptable (véase la variable *GGFINFINITY*) después de haber calculado *GGFCFMAX* cocientes parciales, la función generatriz será considerada no equivalente a una fracción racional y la función `ggf` se detendrá. Puede asignársele a *GGFCFMAX* un valor mayor para funciones generatrices más complicadas.

Véase también `ggf`.

`ggf (l)` [Función]

Calcula la función generatriz de una sucesión de la que se suministran tan solo los primeros valores y cuyo término general es una fracción algebraica (cociente de dos polinomios).

La solución se devuelve como una fracción de polinomios. En caso de no poder encontrar una solución, se devuelve `done`.

Esta función está controlada por las variables globales *GGFINFINITY* y *GGFCFMAX*. Véanse también *GGFINFINITY* y *GGFCFMAX*.

Antes de hacer uso de esta función ejecútese `load("ggf")`.

55 graphs

55.1 Introducción a graphs

El paquete `graphs` permite trabajar con estructuras de grafos y digrafos en Maxima. Tanto los grafos como los digrafos son de estructura simples (no tienen ni aristas mÃºltiples ni bucles), pero los digrafos pueden tener una arista dirigida desde u hasta v y otra desde v hasta u.

Los grafos se representan internamente como listas de adyacencia y se implementan como estructuras de lisp. Los vértices se identifican por sus números de identificaciń (siempre enteros). Las aristas/arcos se representan por listas de longitud 2. Se pueden asignar etiquetas a los vértices de los grafos/digrafos y pesos a sus aristas/arcos.

La función `draw_graph` dibuja grafos siguiendo un criterio rígido de posicionamiento de los vértices. También puede hacer uso del programa graphviz disponible en `http://www.graphviz.org`. La función `draw_graph` utiliza el paquete `draw` de Maxima.

Para hacer uso de este paquete, ejecútese primero `load("graphs")`.

55.2 Funciones y variables para graphs

55.2.1 Construyendo grafos

`create_graph` (*v_list*, *e_list*) [Función]
`create_graph` (*n*, *e_list*) [Función]
`create_graph` (*v_list*, *e_list*, *directed*) [Función]

Crea un nuevo grafo sobre el conjunto de vértices *v_list* con aristas *e_list*.

v_list es una lista de vértices (`[v1, v2,..., vn]`) o una lista de vértices junto con sus respectivas etiquetas (`[[v1,l1], [v2,l2],..., [vn,ln]]`).

n es el número de vértices, los cuales se identificarán desde 0 hasta n-1.

e_list es una lista de aristas (`[e1, e2,..., em]`) o una lista de aristas con sus respectivas ponderaciones (`[[e1, w1], ..., [em, wm]]`).

Si *directed* is not `false`, se devolverá un grafo orientado.

Ejemplos:

Crea un ciclo de 3 vértices.

```
(%i1) load ("graphs")$
(%i2) g : create_graph([1,2,3], [[1,2], [2,3], [1,3]])$
(%i3) print_graph(g)$
Graph on 3 vertices with 3 edges.
Adjacencies:
  3 :  1  2
  2 :  3  1
  1 :  3  2
```

Crea un ciclo de 3 vértices y aristas ponderadas:

```
(%i1) load ("graphs")$
(%i2) g : create_graph([1,2,3], [[[1,2], 1.0], [[2,3], 2.0],
```

```
                                        [[1,3], 3.0]])$
(%i3) print_graph(g)$
Graph on 3 vertices with 3 edges.
Adjacencies:
  3 :  1  2
  2 :  3  1
  1 :  3  2
```

Crea un grafo orientado:

```
(%i1) load ("graphs")$
(%i2) d : create_graph(
       [1,2,3,4],
       [
        [1,3], [1,4],
        [2,3], [2,4]
       ],
       'directed = true)$
(%i3) print_graph(d)$
Digraph on 4 vertices with 4 arcs.
Adjacencies:
  4 :
  3 :
  2 :  4  3
  1 :  4  3
```

`copy_graph` (*g*) [Función]

Devuelve una copia del grafo *g*.

`circulant_graph` (*n*, *d*) [Función]

Devuelve un grafo cirlulante de parámetros *n* y *d*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : circulant_graph(10, [1,3])$
(%i3) print_graph(g)$
Graph on 10 vertices with 20 edges.
Adjacencies:
  9 :  2  6  0  8
  8 :  1  5  9  7
  7 :  0  4  8  6
  6 :  9  3  7  5
  5 :  8  2  6  4
  4 :  7  1  5  3
  3 :  6  0  4  2
  2 :  9  5  3  1
  1 :  8  4  2  0
  0 :  7  3  9  1
```

`clebsch_graph` () [Función]

Devuelve el grafo de Clebsch.

`complement_graph` (*g*) [Función]
Devuelve el complemento del grafo *g*.

`complete_bipartite_graph` (*n*, *m*) [Función]
Devuelve el grafo bipartido completo de *n*+*m* vértices.

`complete_graph` (*n*) [Función]
Devuelve el grafo completo de *n* vértices.

`cycle_digraph` (*n*) [Función]
Devuelve el ciclo dirigido de *n* vértices.

`cycle_graph` (*n*) [Función]
Devuelve el ciclo de *n* vértices.

`cuboctahedron_graph` (*n*) [Función]
Devuelve el grafo cubooctaédrico.

`cube_graph` (*n*) [Función]
Devuelve el cubo de *n* dimensiones.

`dodecahedron_graph` () [Función]
Devuelve el grafo del dodecaedro.

`empty_graph` (*n*) [Función]
Devuelve el grafo vacío de *n* vértices.

`flower_snark` (*n*) [Función]
Devuelve el grafo de flor de *4n* vértices.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) f5 : flower_snark(5)$
(%i3) chromatic_index(f5);
(%o3)                                4
```

`from_adjacency_matrix` (*A*) [Función]
Devuelve el grafo definido por la matriz de adyacencia *A*.

`frucht_graph` () [Función]
Devuelve el grafo de Frucht.

`graph_product` (*g1*, *g1*) [Función]
Devuelve el producto dirigido de los grafos *g1* y *g2*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) grid : graph_product(path_graph(3), path_graph(4))$
(%i3) draw_graph(grid)$
```

`graph_union` (*g1*, *g1*) [Función]
Devuelve la unión (suma) de los grafos *g1* y *g2*.

`grid_graph` (*n*, *m*) [Función]
Devuelve la rejilla *n* x *m*.

`great_rhombicosidodecahedron_graph` () [Función]
Devuelve el grafo gran rombicosidodecaédrico.

`great_rhombicuboctahedron_graph` () [Función]
Devuelve el grafo gran rombicocubicooctaédrico.

`grotzch_graph` () [Función]
Devuelve el grafo de Grotzch.

`heawood_graph` () [Función]
Devuelve el grafo de Heawood.

`icosahedron_graph` () [Función]
Devuelve el grafo icosaédrico.

`icosidodecahedron_graph` () [Función]
Devuelve el grafo icosidodecaédrico.

`induced_subgraph` (*V*, *g*) [Función]
Devuelve el grafo inducido por el subconjunto *V* de vértices del grafo *g*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) V : [0,1,2,3,4]$
(%i4) g : induced_subgraph(V, p)$
(%i5) print_graph(g)$
Graph on 5 vertices with 5 edges.
Adjacencies:
  4 :  3  0
  3 :  2  4
  2 :  1  3
  1 :  0  2
  0 :  1  4
```

`line_graph` (*g*) [Función]
Devuelve el grafo de línea del grafo *g*.

`make_graph` (*vrt*, *f*) [Función]
`make_graph` (*vrt*, *f*, *oriented*) [Función]
Crea un grafo por medio de la función de predicado *f*.

vrt es una lista o conjunto de vértices o un simplemente un número entero. Si *vrt* es un número entero, entonces los vértices del grafo serán los enteros desde 1 hasta *vrt*.

f es una función de predicado. Dos vértices *a* y *b* se conectarán si `f(a,b)=true`.

Si *directed* no es *false*, entonces en grafo será dirigido.

Ejemplo 1:

```
(%i1) load("graphs")$
```

```
(%i2) g : make_graph(powerset({1,2,3,4,5}, 2), disjointp)$
(%i3) is_isomorphic(g, petersen_graph());
(%o3)                          true
(%i4) get_vertex_label(1, g);
(%o4)                        {1, 2}
```

Ejemplo 2:

```
(%i1) load("graphs")$
(%i2) f(i, j) := is (mod(j, i)=0)$
(%i3) g : make_graph(20, f, directed=true)$
(%i4) out_neighbors(4, g);
(%o4)                    [8, 12, 16, 20]
(%i5) in_neighbors(18, g);
(%o5)                    [1, 2, 3, 6, 9]
```

`mycielski_graph` (*g*) [Función]

Devuelve el grafo de Mycielski del grafo *g*.

`new_graph` () [Función]

Devuelve el grafo sin vértices ni aristas.

`path_digraph` (*n*) [Función]

Devuelve el camino dirigido de *n* vértices.

`path_graph` (*n*) [Función]

Devuelve el camino de *n* vértices.

`petersen_graph` () [Función]

`petersen_graph` (*n*, *d*) [Función]

Devuelve el grafo de Petersen $P_{-}\{n,d\}$. Los valores por defecto para *n* y *d* son `n=5` y `d=2`.

`random_bipartite_graph` (`a`, *b*, *p*) [Función]

Devuelve un grafo aleatorio bipartido a partir de los vértices `a+b`. Cada arista se genera con probabilidad *p*.

`random_digraph` (*n*, *p*) [Función]

Devuelve un grafo aleatorio dirigido de *n* vértices. Cada arco se presenta con una probabilidad *p*.

`random_regular_graph` (*n*) [Función]

`random_regular_graph` (*n*, *d*) [Función]

Devuelve un grafo aleatorio *d*-regular de *n* vértices. El valor por defecto para *d* es `d=3`.

`random_graph` (*n*, *p*) [Función]

Devuelve un grafo aleatorio de *n* vértices. Cada arco se presenta con una probabilidad *p*.

`random_graph1` (*n*, *m*) [Función]

Devuelve un grafo aleatorio de *n* vértices y *m* arcos aleatorios.

`random_network` (*n*, *p*, *w*) [Función]

Devuelve una red aleatoria de *n* vértices. Cada arco se presenta con probabilidad *p* y tiene un peso dentro del rango `[0,w]`. La función devuelve una lista `[network, source, sink]`.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) [net, s, t] : random_network(50, 0.2, 10.0);
(%o2)                          [DIGRAPH, 50, 51]
(%i3) max_flow(net, s, t)$
(%i4) first(%);
(%o4)                   27.65981397932507
```

`random_tournament` (*n*) [Función]

Devuelve un torneo aleatorio de *n* vértices.

`random_tree` (*n*) [Función]

Devuelve un árbol aleatorio de *n* vértices.

`small_rhombicosidodecahedron_graph` () [Función]

Devuelve el grafo pequeño rombicosidodecaédrico.

`small_rhombicuboctahedron_graph` () [Función]

Devuelve el grafo pequeño rombicocubicooctaédrico.

`snub_cube_graph` () [Función]

Devuelve el grafo cúbico volteado.

`snub_dodecahedron_graph` () [Función]

Devuelve el grafo dodecaédrico volteado.

`truncated_cube_graph` () [Función]

Devuelve el grafo cúbico truncado.

`truncated_dodecahedron_graph` () [Función]

Devuelve el grafo dodecaédrico truncado.

`truncated_icosahedron_graph` () [Función]

Devuelve el grafo icosaédrico truncado.

`truncated_tetrahedron_graph` () [Función]

Devuelve el grafo del tetraedro truncado.

`tutte_graph` () [Función]

Devuelve el grafo de Tutte.

`underlying_graph` (*g*) [Función]

Devuelve el grafo asociado al grafo orientado *g*.

`wheel_graph` (*n*) [Función]

Devuelve el grafo de rueda de *n+1* vértices.

55.2.2 Propiedades de los grafos

`adjacency_matrix` (*gr*) [Función]

Devuelve la matriz de adyacencia del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) c5 : cycle_graph(4)$
(%i3) adjacency_matrix(c5);
                                [ 0  1  0  1 ]
                                [             ]
                                [ 1  0  1  0 ]
(%o3)                           [             ]
                                [ 0  1  0  1 ]
                                [             ]
                                [ 1  0  1  0 ]
```

`average_degree` (*gr*) [Función]

Devuelve el grado medio de los vértices del garfo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) average_degree(grotzch_graph());
                                      40
(%o2)                                 --
                                      11
```

`biconnected_components` (*gr*) [Función]

Devuelve los subconjuntos de vértices biconectados del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : create_graph(
            [1,2,3,4,5,6,7],
            [
             [1,2],[2,3],[2,4],[3,4],
             [4,5],[5,6],[4,6],[6,7]
            ])$
(%i3) biconnected_components(g);
(%o3)              [[6, 7], [4, 5, 6], [1, 2], [2, 3, 4]]
```

`bipartition` (*gr*) [Función]

Devuelve una bipartición de los vértices del grafo *gr*, o una lista vacía si *gr* no es bipartido.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) h : heawood_graph()$
(%i3) [A,B]:bipartition(h);
(%o3)         [[8, 12, 6, 10, 0, 2, 4], [13, 5, 11, 7, 9, 1, 3]]
(%i4) draw_graph(h, show_vertices=A, program=circular)$
```

chromatic_index (*gr*) [Función]

Devuelve el índice cromático del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) chromatic_index(p);
(%o3)                                4
```

chromatic_number (*gr*) [Función]

Devuelve el número cromático del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) chromatic_number(cycle_graph(5));
(%o2)                                3
(%i3) chromatic_number(cycle_graph(6));
(%o3)                                2
```

clear_edge_weight (*e*, *gr*) [Función]

Elimina el peso del arco *e* del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : create_graph(3, [[[0,1], 1.5], [[1,2], 1.3]])$
(%i3) get_edge_weight([0,1], g);
(%o3)                              1.5
(%i4) clear_edge_weight([0,1], g)$
(%i5) get_edge_weight([0,1], g);
(%o5)                                1
```

clear_vertex_label (*v*, *gr*) [Función]

Elimina la etiqueta del vértice *v* del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : create_graph([[0,"Zero"], [1, "One"]], [[0,1]])$
(%i3) get_vertex_label(0, g);
(%o3)                              Zero
(%i4) clear_vertex_label(0, g);
(%o4)                              done
(%i5) get_vertex_label(0, g);
(%o5)                              false
```

connected_components (*gr*) [Función]

Devuelve las componentes conexas del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g: graph_union(cycle_graph(5), path_graph(4))$
(%i3) connected_components(g);
(%o3)                 [[1, 2, 3, 4, 0], [8, 7, 6, 5]]
```

`diameter` (*gr*) [Función]

Devuelve el diámetro del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) diameter(dodecahedron_graph());
(%o2)                                5
```

`edge_coloring` (*gr*) [Función]

Devuelve una coloración óptima de los arcos del grafo *gr*.

La función devuelve el índice cromático y una lista que representa el coloreado de los arcos de *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) [ch_index, col] : edge_coloring(p);
(%o3) [4, [[[0, 5], 3], [[5, 7], 1], [[0, 1], 1], [[1, 6], 2],
[[6, 8], 1], [[1, 2], 3], [[2, 7], 4], [[7, 9], 2], [[2, 3], 2],
[[3, 8], 3], [[5, 8], 2], [[3, 4], 1], [[4, 9], 4], [[6, 9], 3],
[[0, 4], 2]]]
(%i4) assoc([0,1], col);
(%o4)                            1
(%i5) assoc([0,5], col);
(%o5)                            3
```

`degree_sequence` (*gr*) [Función]

Devuelve una lista con los grados de los vértices del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) degree_sequence(random_graph(10, 0.4));
(%o2)            [2, 2, 2, 2, 2, 2, 3, 3, 3, 3]
```

`edge_connectivity` (*gr*) [Función]

Devuelve la conectividad de las aristas del grafo *gr*.

Véase también `min_edge_cut`.

`edges` (*gr*) [Función]

Devuelve la lista de las aristas (arcos) del grafo (dirigido) *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) edges(complete_graph(4));
(%o2)         [[2, 3], [1, 3], [1, 2], [0, 3], [0, 2], [0, 1]]
```

`get_edge_weight` (*e*, *gr*) [Función]
`get_edge_weight` (*e*, *gr*, *ifnot*) [Función]

Devuelve el peso de la arista *e* del grafo *gr*.

Si la arista no tiene peso, la función devuelve 1. Si la arista no pertenece al grafo, la función emite un mensaje de error o devuelve el argumento opcional *ifnot*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) c5 : cycle_graph(5)$
(%i3) get_edge_weight([1,2], c5);
(%o3)                                1
(%i4) set_edge_weight([1,2], 2.0, c5);
(%o4)                              done
(%i5) get_edge_weight([1,2], c5);
(%o5)                               2.0
```

`get_vertex_label` (*v*, *gr*) [Función]

Devuelve la etiqueta del vértice *v* del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : create_graph([[0,"Zero"], [1, "One"]], [[0,1]])$
(%i3) get_vertex_label(0, g);
(%o3)                              Zero
```

`graph_charpoly` (*gr*, *x*) [Función]

Devuelve el polinomio característico (de variable *x*) del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) graph_charpoly(p, x), factor;
                                          5        4
(%o3)                  (x - 3) (x - 1)  (x + 2)
```

`graph_center` (*gr*) [Función]

Devuelve el centro del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : grid_graph(5,5)$
(%i3) graph_center(g);
(%o3)                              [12]
```

`graph_eigenvalues` (*gr*) [Función]

Devuelve los valores propios del grafo *gr*. La función devuelve los valores propios en el mismo formato en el que lo hace la función `eigenvalue`.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) graph_eigenvalues(p);
(%o3)                  [[3, - 2, 1], [1, 4, 5]]
```

graph_periphery (*gr*) [Función]

Devuelve la periferia del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : grid_graph(5,5)$
(%i3) graph_periphery(g);
(%o3)                          [24, 20, 4, 0]
```

graph_size (*gr*) [Función]

Devuelve el número de aristas del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) graph_size(p);
(%o3)                                  15
```

graph_order (*gr*) [Función]

Devuelve el número de vértices del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) graph_order(p);
(%o3)                                  10
```

girth (*gr*) [Función]

Devuelve la longitud del ciclo más corto del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : heawood_graph()$
(%i3) girth(g);
(%o3)                                   6
```

hamilton_cycle (*gr*) [Función]

Devuelve el ciclo de Hamilton del grafo *gr* o una lista vacía si *gr* no es hamiltoniano.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) c : cube_graph(3)$
(%i3) hc : hamilton_cycle(c);
(%o3)              [7, 3, 2, 6, 4, 0, 1, 5, 7]
(%i4) draw_graph(c, show_edges=vertices_to_cycle(hc))$
```

hamilton_path (*gr*) [Función]

Devuelve el camino de Hamilton del grafo *gr* o una lista vacía si *gr* no los tiene.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
```

```
(%i3) hp : hamilton_path(p);
(%o3)                  [0, 5, 7, 2, 1, 6, 8, 3, 4, 9]
(%i4) draw_graph(p, show_edges=vertices_to_path(hp))$
```

isomorphism (*gr1*, *gr2*) [Función]

Devuelve un isomorfismo entre los grafos/digrafos *gr1* y *gr2*. Si *gr1* y *gr2* no son isomorfos, devuelve una lista vacía.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) clk5:complement_graph(line_graph(complete_graph(5)))$
(%i3) isomorphism(clk5, petersen_graph());
(%o3) [9 -> 0, 2 -> 1, 6 -> 2, 5 -> 3, 0 -> 4, 1 -> 5, 3 -> 6,
                                          4 -> 7, 7 -> 8, 8 -> 9]
```

in_neighbors (*v*, *gr*) [Función]

Devuelve la lista de los nodos hijos del vértice *v* del grafo orientado *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) p : path_digraph(3)$
(%i3) in_neighbors(2, p);
(%o3)                                  [1]
(%i4) out_neighbors(2, p);
(%o4)                                  []
```

is_biconnected (*gr*) [Función]

Devuelve `true` si *gr* está biconectado y `false` en caso contrario.

Ejemplo:

Example:

```
(%i1) load ("graphs")$
(%i2) is_biconnected(cycle_graph(5));
(%o2)                          true
(%i3) is_biconnected(path_graph(5));
(%o3)                          false
```

is_bipartite (*gr*) [Función]

Devuelve `true` si *gr* es bipartido (2-coloreable) y `false` en caso contrario.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) is_bipartite(petersen_graph());
(%o2)                               false
(%i3) is_bipartite(heawood_graph());
(%o3)                               true
```

is_connected (*gr*) [Función]

Devuelve `true` si el grafo *gr* es conexo y `false` en caso contrario.

Ejemplo:

```
(%i1) load ("graphs")$
```

```
(%i2) is_connected(graph_union(cycle_graph(4), path_graph(3)));
(%o2)                                false
```

`is_digraph` (*gr*) [Función]

Devuelve `true` si *gr* es un grafo orientado (digrafo) y `false` en caso contrario.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) is_digraph(path_graph(5));
(%o2)                                false
(%i3) is_digraph(path_digraph(5));
(%o3)                                true
```

`is_edge_in_graph` (*e*, *gr*) [Función]

Devuelve `true` si *e* es una arista (arco) del grafo (digrafo) *g* y `false` en caso contrario.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) c4 : cycle_graph(4)$
(%i3) is_edge_in_graph([2,3], c4);
(%o3)                                true
(%i4) is_edge_in_graph([3,2], c4);
(%o4)                                true
(%i5) is_edge_in_graph([2,4], c4);
(%o5)                                false
(%i6) is_edge_in_graph([3,2], cycle_digraph(4));
(%o6)                                false
```

`is_graph` (*gr*) [Función]

Devuelve `true` si *gr* es un grafo y `false` en caso contrario.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) is_graph(path_graph(5));
(%o2)                                true
(%i3) is_graph(path_digraph(5));
(%o3)                                false
```

`is_graph_or_digraph` (*gr*) [Función]

Devuelve `true` si *gr* es una grafo, orientado o no, y `false` en caso contrario.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) is_graph_or_digraph(path_graph(5));
(%o2)                                true
(%i3) is_graph_or_digraph(path_digraph(5));
(%o3)                                true
```

`is_isomorphic` (*gr1*, *gr2*) [Función]

Devuelve `true` si los grafos/digrafos *gr1* y *gr2* son isomorfos y `false` en caso contrario.

Véase también `isomorphism`.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) clk5:complement_graph(line_graph(complete_graph(5)))$
(%i3) is_isomorphic(clk5, petersen_graph());
(%o3)                          true
```

`is_planar` (*gr*) [Función]

Devuelve `true` si *gr* es un grafo planar y `false` en caso contrario.

El algoritmo utilizado es el de Demoucron, que es de tiempo cuadrático.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) is_planar(dodecahedron_graph());
(%o2)                                true
(%i3) is_planar(petersen_graph());
(%o3)                                false
(%i4) is_planar(petersen_graph(10,2));
(%o4)                                true
```

`is_sconnected` (*gr*) [Función]

Devuelve `true` si el grafo orientado *gr* es fuertemente conexo, devolviendo `false` en caso contrario.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) is_sconnected(cycle_digraph(5));
(%o2)                               true
(%i3) is_sconnected(path_digraph(5));
(%o3)                               false
```

`is_vertex_in_graph` (*v*, *gr*) [Función]

Devuelve `true` si *v* es un vértice del grafo *g* y `false` en caso contrario.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) c4 : cycle_graph(4)$
(%i3) is_vertex_in_graph(0, c4);
(%o3)                               true
(%i4) is_vertex_in_graph(6, c4);
(%o4)                               false
```

`is_tree` (*gr*) [Función]

Devuelve `true` si *gr* es un árbol y `false` en caso contrario.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) is_tree(random_tree(4));
(%o2)                               true
(%i3) is_tree(graph_union(random_tree(4), random_tree(5)));
(%o3)                               false
```

`laplacian_matrix` (*gr*) [Función]

Devuelve el laplaciano de la matriz del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) laplacian_matrix(cycle_graph(5));
                          [  2   - 1   0    0   - 1 ]
                          [                         ]
                          [ - 1   2   - 1   0    0  ]
                          [                         ]
(%o2)                     [  0   - 1   2   - 1   0  ]
                          [                         ]
                          [  0    0   - 1   2   - 1 ]
                          [                         ]
                          [ - 1   0    0   - 1   2  ]
```

`max_clique` (*gr*) [Función]

Devuelve el clique máximo del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : random_graph(100, 0.5)$
(%i3) max_clique(g);
(%o3)       [6, 12, 31, 36, 52, 59, 62, 63, 80]
```

`max_degree` (*gr*) [Función]

Devuelve el grado máximo de los vértices del grafo *gr* y un vértice de grado máximo.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : random_graph(100, 0.02)$
(%i3) max_degree(g);
(%o3)                        [6, 79]
(%i4) vertex_degree(95, g);
(%o4)                           3
```

`max_flow` (*net*, *s*, *t*) [Función]

Devuelve el flujo maximal de la red *net* con origen en *s* y final en *t*.

La función devuelve el valor del flujo maximal y una lista con los pesos de los arcos del flujo óptimo.

Ejemplo:

Example:

```
(%i1) load ("graphs")$
(%i2) net : create_graph(
  [1,2,3,4,5,6],
  [[[1,2], 1.0],
   [[1,3], 0.3],
   [[2,4], 0.2],
   [[2,5], 0.3],
```

```
   [[3,4], 0.1],
   [[3,5], 0.1],
   [[4,6], 1.0],
   [[5,6], 1.0]],
  directed=true)$
(%i3) [flow_value, flow] : max_flow(net, 1, 6);
(%o3) [0.7, [[[1, 2], 0.5], [[1, 3], 0.2], [[2, 4], 0.2],
[[2, 5], 0.3], [[3, 4], 0.1], [[3, 5], 0.1], [[4, 6], 0.3],
[[5, 6], 0.4]]]
(%i4) fl : 0$
(%i5) for u in out_neighbors(1, net)
         do fl : fl + assoc([1, u], flow)$
(%i6) fl;
(%o6)                                0.7
```

`max_independent_set` (*gr*) [Función]

Devuelve un conjunto maximal independiente de vértices del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) d : dodecahedron_graph()$
(%i3) mi : max_independent_set(d);
(%o3)             [0, 3, 5, 9, 10, 11, 18, 19]
(%i4) draw_graph(d, show_vertices=mi)$
```

`max_matching` (*gr*) [Función]

Devuelve un conjunto maximal independiente de aristas del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) d : dodecahedron_graph()$
(%i3) m : max_matching(d);
(%o3) [[5, 7], [8, 9], [6, 10], [14, 19], [13, 18], [12, 17],
                                [11, 16], [0, 15], [3, 4], [1, 2]]
(%i4) draw_graph(d, show_edges=m)$
```

`min_degree` (*gr*) [Función]

Devuelve el grado mínimo de los vértices del grafo *gr* y un vértice de grado mínimo.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : random_graph(100, 0.1)$
(%i3) min_degree(g);
(%o3)                             [3, 49]
(%i4) vertex_degree(21, g);
(%o4)                               9
```

`min_edge_cut` (*gr*) [Función]

Devuelve el mínimo *edge cut* del grafo *gr*. Un *edge cut* es un conjunto de aristas cuya eliminación aumenta el número de componentes del grafo.

Véase también `edge_connectivity`.

`min_vertex_cover` (*gr*) [Función]
Devuelve el mínimo nodo *covering* del grafo *gr*.

`min_vertex_cut` (*gr*) [Función]
Devuelve el mínimo *vertex cut* del grafo *gr*.

Véase también `vertex_connectivity`.

`minimum_spanning_tree` (*gr*) [Función]
Devuelve el grafo de expansión mínimo del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : graph_product(path_graph(10), path_graph(10))$
(%i3) t : minimum_spanning_tree(g)$
(%i4) draw_graph(g, show_edges=edges(t))$
```

`neighbors` (*v*, *gr*) [Función]
Devuelve la lista de los vecinos del vértice *v* del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) neighbors(3, p);
(%o3)                              [4, 8, 2]
```

`odd_girth` (*gr*) [Función]
Devuelve la longitud del ciclo impar más corto del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : graph_product(cycle_graph(4), cycle_graph(7))$
(%i3) girth(g);
(%o3)                                   4
(%i4) odd_girth(g);
(%o4)                                   7
```

`out_neighbors` (*v*, *gr*) [Función]
Devuelve la lista de los nodos padres del vértice *v* del grafo orientado *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) p : path_digraph(3)$
(%i3) in_neighbors(2, p);
(%o3)                                  [1]
(%i4) out_neighbors(2, p);
(%o4)                                  []
```

`planar_embedding` (*gr*) [Función]
Devuelve la lista de caminos faciales en una proyección planar de *gr*, o `false` si *gr* no es un grafo planar.

El grafo *gr* debe estar biconectado.

El algoritmo utilizado es el de Demoucron, que es de tiempo cuadrático.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) planar_embedding(grid_graph(3,3));
(%o2) [[3, 6, 7, 8, 5, 2, 1, 0], [4, 3, 0, 1], [3, 4, 7, 6],
                                        [8, 7, 4, 5], [1, 2, 5, 4]]
```

`print_graph` (*gr*) [Función]

Muestra alguna información sobre el grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) c5 : cycle_graph(5)$
(%i3) print_graph(c5)$
Graph on 5 vertices with 5 edges.
Adjacencies:
  4 :  0  3
  3 :  4  2
  2 :  3  1
  1 :  2  0
  0 :  4  1
(%i4) dc5 : cycle_digraph(5)$
(%i5) print_graph(dc5)$
Digraph on 5 vertices with 5 arcs.
Adjacencies:
  4 :  0
  3 :  4
  2 :  3
  1 :  2
  0 :  1
(%i6) out_neighbors(0, dc5);
(%o6)                                [1]
```

`radius` (*gr*) [Función]

Devuelve el radio del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) radius(dodecahedron_graph());
(%o2)                                 5
```

`set_edge_weight` (*e*, *w*, *gr*) [Función]

Asigna el peso *w* a la arista *e* del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : create_graph([1, 2], [[[1,2], 1.2]])$
(%i3) get_edge_weight([1,2], g);
(%o3)                                1.2
```

```
(%i4) set_edge_weight([1,2], 2.1, g);
(%o4)                                done
(%i5) get_edge_weight([1,2], g);
(%o5)                                 2.1
```

`set_vertex_label` (*v*, *l*, *gr*) [Función]

Asigna la etiqueta *l* al vértice *v* del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : create_graph([[1, "One"], [2, "Two"]], [[1,2]])$
(%i3) get_vertex_label(1, g);
(%o3)                                 One
(%i4) set_vertex_label(1, "oNE", g);
(%o4)                                done
(%i5) get_vertex_label(1, g);
(%o5)                                 oNE
```

`shortest_path` (*u*, *v*, *gr*) [Función]

Devuelve el camino más corto desde *u* hasta *v* del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) d : dodecahedron_graph()$
(%i3) path : shortest_path(0, 7, d);
(%o3)                          [0, 1, 19, 13, 7]
(%i4) draw_graph(d, show_edges=vertices_to_path(path))$
```

`shortest_weighted_path` (*u*, *v*, *gr*) [Función]

Devuelve la longitud del camino más corto ponderado y el propio camino más corto ponderado desde *u* hasta *v* en el grafo *gr*.

La longitud del camino ponderado es la suma de los pesos de las aristas del camino. Si una arista no tiene peso asignado, su valor por defecto es la unidad.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g: petersen_graph(20, 2)$
(%i3) for e in edges(g) do set_edge_weight(e, random(1.0), g)$
(%i4) shortest_weighted_path(0, 10, g);
(%o4) [2.575143920268482, [0, 20, 38, 36, 34, 32, 30, 10]]
```

`strong_components` (*gr*) [Función]

Devuelve las componentes fuertes del grafo orientado *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) t : random_tournament(4)$
(%i3) strong_components(t);
(%o3)                          [[1], [0], [2], [3]]
(%i4) vertex_out_degree(3, t);
(%o4)                                 3
```

`topological_sort` (*dag*) [Función]

Devuelve el orden topológico de los vértices del grafo orientado *dag* o una lista vacía si *dag* no es un grafo orientado acíclico.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g:create_graph(
         [1,2,3,4,5],
         [
          [1,2], [2,5], [5,3],
          [5,4], [3,4], [1,3]
         ],
         directed=true)$
(%i3) topological_sort(g);
(%o3)                          [1, 2, 5, 3, 4]
```

`vertex_connectivity` (*g*) [Función]

Devuelve la conectividad de los vértices del grafo *g*.

Véase también `min_vertex_cut`.

`vertex_degree` (*v*, *gr*) [Función]

Devuelve el grado del vértice *v* del grafo *gr*.

`vertex_distance` (*u*, *v*, *gr*) [Función]

Devuelve la longitud del camino más corto entre *u* y *v* del grafo o digrafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) d : dodecahedron_graph()$
(%i3) vertex_distance(0, 7, d);
(%o3)                                4
(%i4) shortest_path(0, 7, d);
(%o4)                        [0, 1, 19, 13, 7]
```

`vertex_eccentricity` (*v*, *gr*) [Función]

Devuelve la excentricidad del vértice *v* del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g:cycle_graph(7)$
(%i3) vertex_eccentricity(0, g);
(%o3)                            3
```

`vertex_in_degree` (*v*, *gr*) [Función]

Devuelve el grado de entrada del vértice *v* del grafo orientado *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) p5 : path_digraph(5)$
(%i3) print_graph(p5)$
```

```
Digraph on 5 vertices with 4 arcs.
Adjacencies:
  4 :
  3 :  4
  2 :  3
  1 :  2
  0 :  1
(%i4) vertex_in_degree(4, p5);
(%o4)                                1
(%i5) in_neighbors(4, p5);
(%o5)                               [3]
```

`vertex_out_degree` (*v*, *gr*) [Función]

Devuelve el grado de salida del vértice *v* del grafo orientado *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) t : random_tournament(10)$
(%i3) vertex_out_degree(0, t);
(%o3)                            2
(%i4) out_neighbors(0, t);
(%o4)                       [7, 1]
```

`vertices` (*gr*) [Función]

Devuelve la lista de vértices del grafo *gr*.

Example

```
(%i1) load ("graphs")$
(%i2) vertices(complete_graph(4));
(%o2)                          [3, 2, 1, 0]
```

`vertex_coloring` (*gr*) [Función]

Devuelve un coloreado óptimo de los vértices del grafo *gr*.

La función devuelve el número cromático y una lista representando el coloreado de los vértices de *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) p:petersen_graph()$
(%i3) vertex_coloring(p);
(%o3) [3, [[0, 2], [1, 3], [2, 2], [3, 3], [4, 1], [5, 3],
                                  [6, 1], [7, 1], [8, 2], [9, 2]]]
```

`wiener_index` (*gr*) [Función]

Devuelve el índice de Wiener del grafo *gr*.

Ejemplo:

```
(%i1) wiener_index(dodecahedron_graph());
(%o1)                          500
```

55.2.3 Modificación de grafos

`add_edge` (*e*, *gr*) [Función]

Añade la arista *e* al grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) p : path_graph(4)$
(%i3) neighbors(0, p);
(%o3)                                [1]
(%i4) add_edge([0,3], p);
(%o4)                               done
(%i5) neighbors(0, p);
(%o5)                              [3, 1]
```

`add_edges` (*e_list*, *gr*) [Función]

Añade las aristas de la lista *e_list* al grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : empty_graph(3)$
(%i3) add_edges([[0,1],[1,2]], g)$
(%i4) print_graph(g)$
Graph on 3 vertices with 2 edges.
Adjacencies:
  2 :  1
  1 :  2  0
  0 :  1
```

`add_vertex` (*v*, *gr*) [Función]

Añade el vértice *v* al grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : path_graph(2)$
(%i3) add_vertex(2, g)$
(%i4) print_graph(g)$
Graph on 3 vertices with 1 edges.
Adjacencies:
  2 :
  1 :  0
  0 :  1
```

`add_vertices` (*v_list*, *gr*) [Función]

Añade los vértices de la lista *v_list* al grafo *gr*.

`connect_vertices` (*v_list*, *u_list*, *gr*) [Función]

Conecta todos los vértices de la lista *v_list* con los vértices de la lista *u_list* del grafo *gr*.

v_list y *u_list* pueden ser vértices aislados o una lista de vértices.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g : empty_graph(4)$
(%i3) connect_vertices(0, [1,2,3], g)$
(%i4) print_graph(g)$
Graph on 4 vertices with 3 edges.
Adjacencies:
  3 :  0
  2 :  0
  1 :  0
  0 :  3  2  1
```

`contract_edge` (*e*, *gr*) [Función]

Contrae la arista *e* del *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) g: create_graph(
       8, [[0,3],[1,3],[2,3],[3,4],[4,5],[4,6],[4,7]])$
(%i3) print_graph(g)$
Graph on 8 vertices with 7 edges.
Adjacencies:
  7 :  4
  6 :  4
  5 :  4
  4 :  7  6  5  3
  3 :  4  2  1  0
  2 :  3
  1 :  3
  0 :  3
(%i4) contract_edge([3,4], g)$
(%i5) print_graph(g)$
Graph on 7 vertices with 6 edges.
Adjacencies:
  7 :  3
  6 :  3
  5 :  3
  3 :  5  6  7  2  1  0
  2 :  3
  1 :  3
  0 :  3
```

`remove_edge` (*e*, *gr*) [Función]

Elimina la arista *e* del grafo *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) c3 : cycle_graph(3)$
```

```
(%i3) remove_edge([0,1], c3)$
(%i4) print_graph(c3)$
Graph on 3 vertices with 2 edges.
Adjacencies:
  2 :  0  1
  1 :  2
  0 :  2
```

remove_vertex (*v*, *gr*) [Función]
Elimina el vértice *v* del grafo *gr*.

vertex_coloring (*gr*) [Función]
Devuelve un coloreado óptimo de los vértice del grafo *gr*.

La función devuelve el número cromático y una lista representando el coloreado de los vértices de *gr*.

Ejemplo:

```
(%i1) load ("graphs")$
(%i2) p:petersen_graph()$
(%i3) vertex_coloring(p);
(%o3) [3, [[0, 2], [1, 3], [2, 2], [3, 3], [4, 1], [5, 3],
                                    [6, 1], [7, 1], [8, 2], [9, 2]]]
```

55.2.4 Lectura y escritura de ficheros

dimacs_export (*gr*, *fl*) [Función]
dimacs_export (*gr*, *fl*, *comment1*, ..., *commentn*) [Función]
Exporta el grafo al fichero *fl* en formato DIMACS. Los comentarios adicionales se anãdirán al comienzo del fichero.

dimacs_import (*fl*) [Función]
Lee el grafo almacenado en el fichero *fl* en formato DIMACS.

graph6_decode (*str*) [Función]
Devuelve el grafo codificado en formato graph6 en la cadena *str*.

graph6_encode (*gr*) [Función]
Devuelve una cadena codificando el grafo *gr* en formato graph6.

graph6_export (*gr_list*, *fl*) [Función]
Exporta los grafos de la lista *gr_list* al fichero *fl* en formato graph6.

graph6_import (*fl*) [Función]
Lee la lista de grafos almacenados en el fichero *fl* en formato graph6.

sparse6_decode (*str*) [Función]
Devuelve el grafo codificado en formato sparse6 en la cadena *str*.

sparse6_encode (*gr*) [Función]
Devuelve una cadena codificando el grafo *gr* en formato sparse6.

`sparse6_export` (*gr_list*, *fl*) [Función]
Exporta los grafos de la lista *gr_list* al fichero *fl* en formato sparse6.

`sparse6_import` (*fl*) [Función]
Lee la lista de grafos almacenados en el fichero *fl* en formato sparse6.

55.2.5 Visualización

`draw_graph` (*graph*) [Función]
`draw_graph` (*graph*, *option1*, ..., *optionk*) [Función]
Dibuja el grafo utilizando el paquete `draw`.

El algoritmo utilizado para posicionar los vértices se especifica con el argumento opcional *program*, cuyo valor por defecto es `program=spring_embedding`. *draw_graph* también puede utilizar los programas de graphviz para posicionar los vértices, para lo cual deberá instalarse separadamente el programa graphviz.

Ejemplo 1:

```
(%i1) load ("graphs")$
(%i2) g:grid_graph(10,10)$
(%i3) m:max_matching(g)$
(%i4) draw_graph(g,
   spring_embedding_depth=100,
   show_edges=m, edge_type=dots,
   vertex_size=0)$
```

Ejemplo 2:

```
(%i1) load ("graphs")$
(%i2) g:create_graph(16,
    [
     [0,1],[1,3],[2,3],[0,2],[3,4],[2,4],
     [5,6],[6,4],[4,7],[6,7],[7,8],[7,10],[7,11],
     [8,10],[11,10],[8,9],[11,12],[9,15],[12,13],
     [10,14],[15,14],[13,14]
    ])$
(%i3) t:minimum_spanning_tree(g)$
(%i4) draw_graph(
    g,
    show_edges=edges(t),
    show_edge_width=4,
    show_edge_color=green,
    vertex_type=filled_square,
    vertex_size=2
    )$
```

Ejemplo 3:

```
(%i1) load ("graphs")$
(%i2) mi : max_independent_set(g)$
(%i3) draw_graph(
    g,
```

```
    show_vertices=mi,
    show_vertex_type=filled_up_triangle,
    show_vertex_size=2,
    edge_color=cyan,
    edge_width=3,
    =true,
    text_color=brown
    )$
```

Ejemplo 4:

```
(%i1) load ("graphs")$
(%i2) net : create_graph(
    [0,1,2,3,4,5],
    [
     [[0,1], 3], [[0,2], 2],
     [[1,3], 1], [[1,4], 3],
     [[2,3], 2], [[2,4], 2],
     [[4,5], 2], [[3,5], 2]
    ],
    directed=true
    )$
(%i3) draw_graph(
    net,
    show_weight=true,
    vertex_size=0,
    show_vertices=[0,5],
    show_vertex_type=filled_square,
    head_length=0.2,
    head_angle=10,
    edge_color="dark-green",
    text_color=blue
    )$
```

Ejemplo 5:

```
(%i1) load("graphs")$
(%i2) g: petersen_graph(20, 2);
(%o2)                          GRAPH
(%i3) draw_graph(g, redraw=true, program=planar_embedding);
(%o3)                          done
```

Ejemplo 6:

```
(%i1) load("graphs")$
(%i2) t: tutte_graph();
(%o2)                          GRAPH
(%i3) draw_graph(t, redraw=true,
                    fixed_vertices=[1,2,3,4,5,6,7,8,9]);
(%o3)                          done
```

`draw_graph_program` [Variable opcional]
Valor por defecto: *spring_embedding*
Programa a utilizar por defecto para posicionar los vértices en la función `draw_graph`.

`show_id` [Opción de draw_graph]
Valor por defecto: *false*
Si *show_id* vale *true* entonces se muestran los números identificadores de los vértices.

`show_label` [Opción de draw_graph]
Valor por defecto: *false*
Si *show_label* vale *true* entonces se muestran las etiquetas de los vértices.

`label_alignment` [Opción de draw_graph]
Valor por defecto: *center*
Indica cómo se deben alinear las etiquetas o números identificadores de los vértices. Puede ser: `left`, `center` or `right`.

`show_weight` [Opción de draw_graph]
Valor por defecto: *false*
si *show_weight* vale *true* entonces se mostrarán los pesos de las aristas.

`vertex_type` [Opción de draw_graph]
Valor por defecto: *circle*
Establece cómo se mostrarán los vértices. Véase la opción *point_type* del paquete `draw`.

`vertex_size` [Opción de draw_graph]
Tamanõ de los vértices.

`vertex_color` [Opción de draw_graph]
Color a utilizar en los vértices.

`show_vertices` [Opción de draw_graph]
Valor por defecto: []
Dibuja los vértices seleccionados en la lista con colores diferentes.

`show_vertex_type` [Opción de draw_graph]
Establece cómo se mostrarán los vértices de *show_vertices*. Véase la opción *point_type* del paquete `draw`.

`show_vertex_size` [Opción de draw_graph]
Tamanõs de los vértices de *show_vertices*.

`show_vertex_color` [Opción de draw_graph]
Color a utilizar en los vértices de la lista *show_vertices*.

`vertex_partition` [Opción de draw_graph]
Valor por defecto: []
Una partición `[[v1,v2,...],...,[vk,...,vn]]` de los vértices del grafo. Los vértices de cada lista se dibujarán de diferente color.

`vertex_coloring` [Opción de draw_graph]
Colores de los vértices. Los colores *col* deben especificarse en el mismo formato que el devuelto por *vertex_coloring*.

`edge_color` [Opción de draw_graph]
Color a utilizar en las aristas.

`edge_width` [Opción de draw_graph]
Ancho de las aristas.

`edge_type` [Opción de draw_graph]
Establece cómo se dibujarán las aristas. Véase la opción *line_type* del paquete `draw`.

`show_edges` [Opción de draw_graph]
Dibuja las aristas de la lista *e_list* con colores diferentes.

`show_edge_color` [Opción de draw_graph]
Color a utilizar en las aristas de la lista *show_edges*.

`show_edge_width` [Opción de draw_graph]
Anchos de las aristas de *show_edges*.

`show_edge_type` [Opción de draw_graph]
Establece cómo se dibujarán las aristas de *show_edges*. Véase la opción *line_type* del paquete `draw`.

`edge_partition` [Opción de draw_graph]
Una partición `[[e1,e2,...],...,[ek,...,em]]` de las aristas del grafo. Las aristas de cada lista se dibujarán de diferente color.

`edge_coloring` [Opción de draw_graph]
Colores de las aristas. Los colores *col* deben especificarse en el mismo formato que el devuelto por *edge_coloring*.

`redraw` [Opción de draw_graph]
Valor por defecto: *false*

Si *redraw* vale `true`, las posiciones de los vértices se recalculan incluso si las posiciones están almacenadas de un dibujo previo del grafo.

`head_angle` [Opción de draw_graph]
Valor por defecto: 15

Ángulo de las flechas de los arcos en los grafos orientados.

`head_length` [Opción de draw_graph]
Valor por defecto: 0.1

Longitud de las flechas de los arcos en los grafos orientados.

`spring_embedding_depth` [Opción de draw_graph]
Valor por defecto: 50

Número de iteraciones del algoritmo de dibujo de grafos.

`terminal` [Opción de draw_graph]
Terminal utilizado para ver el gráfo. Véase la opción *terminal* del paquete `draw`.

`file_name` [Opción de draw_graph]
Nombre del fichero cuando el terminal especificado no es la pantalla.

`program` [Opción de draw_graph]
establece el programa para posicionado de vértices del grafo. Puede ser cualquiera de los programas graphviz (dot, neato, twopi, circ, fdp), *circular* o *spring_embedding* o *planar_embedding*; *planar_embedding* sólo está disponible para grafos planares 2-conectados. Si `program=spring_embedding`, se puede especificar un conjunto de vértices de posición fija con la opción *fixed_vertices*.

`fixed_vertices` [Opción de draw_graph]
Especifica una lista de vértices con posiciones fijas en un polígono regular. Se puede utilizar cuando `program=spring_embedding`.

`vertices_to_path (`*`v_list`*`)` [Función]
Convierte una lista *v_list* de vértices en la lista de aristas del camino definido por la propia *v_list*.

`vertices_to_cycle (`*`v_list`*`)` [Función]
Convierte una lista *v_list* de vértices en la lista de aristas del ciclo definido por la propia *v_list*.

56 grobner

56.1 Introducción a grobner

`grobner` es un paquete para operar con bases de Groebner en Maxima.

Para hacer uso de las funciones de este paquete es necesario cargar previamente el archivo `grobner.lisp`:

```
load("grobner");
```

Es posible ejecutar una demostración haciendo

```
demo("grobner.demo");
```

o

```
batch("grobner.demo")
```

Algunos de los cálculos de la demostración pueden llevar tiempo, razón por la cual sus resultados se han guardado en el archivo `grobner-demo.output`, que se encuentra en el mismo directorio que el archivo de demostración.

56.1.1 Notas sobre el paquete grobner

El autor del paquete es

Marek Rychlik

http://alamos.math.arizona.edu

habiendo sido distribuido el 24-05-2002 bajo los términos de la General Public License (GPL) (ver archivo `grobner.lisp`). Esta documentación ha sido extraída de los archivos

`README`, `grobner.lisp`, `grobner.demo` y `grobner-demo.output`

por Günter Nowak. Las sugerencias para mejorar la documentación se pueden hacer en la lista de correos de *maxima*, `maxima@math.utexas.edu`.

El código está algo anticuado. Las implementaciones modernas utilizan el algoritmo *F4*, más rápido, descrito en

A new efficient algorithm for computing Gröbner bases (F4)
Jean-Charles Faugère
LIP6/CNRS Université Paris VI
January 20, 1999

56.1.2 Implementaciones de órdenes admisibles de monomios

- `lex`

  lexicográfico puro; orden por defecto para la comparación de monomios.
- `grlex`

  grado total, con empates resueltos por el orden lexicográfico.
- `grevlex`

  grado total, con empates resueltos por el orden lexicográfico inverso.
- `invlex`

  orden lexicográfico inverso.

56.2 Funciones y variables para grobner

56.2.1 Variables opcionales

`poly_monomial_order` [Variable opcional]
Valor por defecto: `lex`

Controla qué orden de monomios utiliza en los cálculos con polinomios y bases de Groebner. Si no se le asigna valor alguno, se utilizará `lex`.

`poly_coefficient_ring` [Variable opcional]
Valor por defecto: `expression_ring`

Indica el anillo de coeficientes de los polinomios que se va a utilizar en los cálculos. Si no se le asigna ningún valor, se utilizará el anillo de expresiones propio de *maxima*. A esta variable se le puede asignar el valor `ring_of_integers`.

`poly_primary_elimination_order` [Variable opcional]
Valor por defecto: `false`

Nombre del orden por defecto para las variables eliminadas en las funciones basadas en eliminaciones. Si no se le asigna ningún valor, se utilizará `lex`.

`poly_secondary_elimination_order` [Variable opcional]
Valor por defecto: `false`

Nombre del orden por defecto para las variables almacenadas en funciones basadas en eliminaciones. Si no se le asigna ningún valor, se utilizará `lex`.

`poly_elimination_order` [Variable opcional]
Valor por defecto: `false`

Nombre del orden de eliminación por defecto utilizado en los cálculos de eliminación. Si se le asigna un valor, ignorará los guardados en `poly_primary_elimination_order` y `poly_secondary_elimination_order`. El usuario se asegurará que este es un orden válido de eliminación.

`poly_return_term_list` [Variable opcional]
Valor por defecto: `false`

Si vale `true`, todas las funciones de este paquete devolverán los polinomios como una lista de términos en el orden activo de monomios, en lugar de una expresión ordinaria de *maxima*.

`poly_grobner_debug` [Variable opcional]
Valor por defecto: `false`

Si vale `true`, genera una salida de seguimiento y depuración.

`poly_grobner_algorithm` [Variable opcional]
Valor por defecto: `buchberger`

Valores posibles:

- `buchberger`
- `parallel_buchberger`

- `gebauer_moeller`

Es el nombre del algoritmo utilizado para encontrar las bases de Groebner.

`poly_top_reduction_only` [Variable opcional]
Valor por defecto: `false`

Si no vale `false`, siempre que sea posible el algoritmo de división se detendrá tras la primera reducción.

56.2.2 Operadores simples

`poly_add`, `poly_subtract`, `poly_multiply` y `poly_expt` son los operadores aritméticos para polinomios. Se ejecutan utilizando la representación interna, pero los resultados se devuelven en forma de expresión ordinaria de *maxima*.

`poly_add` (*poly1*, *poly2*, *varlist*) [Función]
Suma los polinomios *poly1* y *poly2*.

```
(%i1) poly_add(z+x^2*y,x-z,[x,y,z]);
                                    2
(%o1)                              x  y + x
```

`poly_subtract` (*poly1*, *poly2*, *varlist*) [Función]
Resta el polinomio *poly2* de *poly1*.

```
(%i1) poly_subtract(z+x^2*y,x-z,[x,y,z]);
                                      2
(%o1)                          2 z + x  y - x
```

`poly_multiply` (*poly1*, *poly2*, *varlist*) [Función]
Multiplica *poly1* por *poly2*.

```
(%i2) poly_multiply(z+x^2*y,x-z,[x,y,z])-(z+x^2*y)*(x-z),expand;
(%o1)                                  0
```

`poly_s_polynomial` (*poly1*, *poly2*, *varlist*) [Función]
Devuelve el *polinomio syzygy* (*S-polinomio*) de dos polinomios *poly1* y *poly2*.

`poly_primitive_part` (*poly1*, *varlist*) [Función]
Devuelve el polinomio *poly* dividido por el MCD de sus coeficientes.

```
(%i1) poly_primitive_part(35*y+21*x,[x,y]);
(%o1)                              5 y + 3 x
```

`poly_normalize` (*poly*, *varlist*) [Función]
Devuelve el polinomio *poly* dividido por el coeficiente principal. Da por supuesto que la división es posible, lo cual puede no ser siempre cierto en anillos que no son campos.

56.2.3 Otras funciones

poly_expand (*poly*, *varlist*) [Función]

Esta función expande los polinomios. Equivale a `expand(poly)` si *poly* es un polinomio. Si la representación no es compatible con un polinomio de variables *varlist*, devuelve un error.

```
(%i1) poly_expand((x-y)*(y+x),[x,y]);
                                     2    2
(%o1)                               x  - y
(%i2) poly_expand((y+x)^2,[x,y]);
                                  2           2
(%o2)                            y  + 2 x y + x
(%i3) poly_expand((y+x)^5,[x,y]);
                 5      4         2  3       3  2      4      5
(%o3)           y  + 5 x y  + 10 x  y  + 10 x  y  + 5 x  y + x
(%i4) poly_expand(-1-x*exp(y)+x^2/sqrt(y),[x]);
                                        2
                                   y      x
(%o4)                       - x %e  + ------- - 1
                                      sqrt(y)

(%i5) poly_expand(-1-sin(x)^2+sin(x),[sin(x)]);
                                  2
(%o5)                       - sin (x) + sin(x) - 1
```

poly_expt (*poly*, *number*, *varlist*) [Función]

Eleva el polinomio *poly* a la potencia *number*, siendo este un entero positivo. Si *number* no es un número entero positivo, devolverá un error.

```
(%i1) poly_expt(x-y,3,[x,y])-(x-y)^3,expand;
(%o1)                                0
```

poly_content (*poly*. *varlist*) [Función]

`poly_content` calcula el MCD de los coeficientes.

```
(%i1) poly_content(35*y+21*x,[x,y]);
(%o1)                                7
```

poly_pseudo_divide (*poly*, *polylist*, *varlist*) [Función]

Realiza la seudo-división del polinomio *poly* por la lista de n polinomios de *polylist*. Devuelve varios resultados. El primer resultado es una lista de cocientes a. El segundo resultado es el resto r. El tercer resultado es un coeficiente escalar c, tal que $c * poly$ puede dividirse por *polylist* dentro del anillo de coeficientes, el cual no es necesariamente un campo. Por último, el cuarto resultado es un entero que guarda el recuento de reducciones realizadas. El objeto resultante satisface la ecuación:

$$c * poly = \sum_{i=1}^{n} (a_i * polylist_i) + r$$

`poly_exact_divide` (*poly1*, *poly2*, *varlist*) [Función]
Divide el polinomio *poly1* por otro polinomio *poly2*. Da por supuesto que es posible la división de resto nulo. Devuelve el cociente.

`poly_normal_form` (*poly*, *polylist*, *varlist*) [Función]
`poly_normal_form` encuentra la forma normal de un polinomio *poly* respecto de un conjunto de polinomios *polylist*.

`poly_buchberger_criterion` (*polylist*, *varlist*) [Función]
Devuelve `true` si *polylist* es una base de Groebner respecto del orden de términos activo, utilizando el criterio de Buchberger: para cualesquiera polinomios $h1$ y $h2$ de *polylist* el S-polinomio $S(h1, h2)$ se reduce a 0 *modulo polylist*.

`poly_buchberger` (*polylist_fl varlist*) [Función]
`poly_buchberger` ejecuta el algoritmo de Buchberger sobre una lista de polinomios y devuelve la base de Groebner resultante.

56.2.4 Postprocesamiento estándar de bases de Groebner

El *k-ésimo ideal de eliminación* I_k de un ideal I sobre $K[x_1, ..., x_1]$ es $I \cap K[x_{k+1}, ..., x_n]$.

El *ideal* $I : J$ es el ideal $\{h | \forall w \in J : wh \in I\}$.

El ideal $I : p^\infty$ es el ideal $\{h | \exists n \in N : p^n h \in I\}$.

El ideal $I : J^\infty$ es el ideal $\{h | \exists n \in N, \exists p \in J : p^n h \in I\}$.

El *ideal radical* $\sqrt{I}$ es el ideal $\{h | \exists n \in N : h^n \in I\}$.

`poly_reduction` (*polylist*, *varlist*) [Función]
`poly_reduction` reduce una lista de polinomios *polylist* de manera que cada polinomio se reduce completamente respecto de los otros polinomios.

`poly_minimization` (*polylist*, *varlist*) [Función]
Devuelve una sublista de la lista de polinomios *polylist* con el mismo ideal de monomios que *polylist*, pero mínimo, esto es, ningún monomio principal de los polinomios de la sublista divide a los monomios principales de los demás polinomios.

`poly_normalize_list` (*polylist*, *varlist*) [Función]
`poly_normalize_list` aplica `poly_normalize` a cada polinomio de la lista. Esto significa que divide cada polinomio de *polylist* por su coeficiente principal.

`poly_grobner` (*polylist*, *varlist*) [Función]
Devuelve la base de Groebner del ideal asociado a los polinomios de *polylist*. El resultado depende de las variables globales.

`poly_reduced_grobner` (*polylist*, *varlist*) [Función]
Devuelve la base de Groebner reducida del ideal asociado a los polinomios de *polylist*. El resultado depende de las variables globales.

poly_depends_p (*poly*, *var*, *varlist*) [Función]
poly_depends comprueba si el polinomio depende de la variable *var*.

poly_elimination_ideal (*polylist*, *n*, *varlist*) [Función]
poly_elimination_ideal devuelve la base de Groebner del n-ésimo ideal de eliminación de un ideal especificado como una lista de polinomios generadores (no necesariamente una base de Groebner).

poly_colon_ideal (*polylist1*, *polylist2*, *varlist*) [Función]
Devuelve la base de Groebner reducida del ideal

$I(polylist1) : I(polylist2)$

siendo *polylist*1 y *polylist*2 dos listas de polinomios.

poly_ideal_intersection (*polylist1*, *polylist2*, *varlist*) [Función]
poly_ideal_intersection devuelve la intersección de dos ideales.

poly_lcm (*poly1*, *poly2*, *varlist*) [Función]
Devuelve el MCM de *poly1* y *poly2*.

poly_gcd (*poly1*, *poly2*, *varlist*) [Función]
Devuelve el MCD de *poly1* y *poly2*.

Véanse también ezgcd, gcd, gcdex y gcdivide.

Ejemplo:

```
(%i1) p1:6*x^3+19*x^2+19*x+6;
                         3       2
(%o1)                 6 x  + 19 x  + 19 x + 6
(%i2) p2:6*x^5+13*x^4+12*x^3+13*x^2+6*x;
                   5       4       3       2
(%o2)          6 x  + 13 x  + 12 x  + 13 x  + 6 x
(%i3) poly_gcd(p1, p2, [x]);
                              2
(%o3)                     6 x  + 13 x + 6
```

poly_grobner_equal (*polylist1*, *polylist2*, *varlist*) [Función]
poly_grobner_equal comprueba si dos bases de Groebner generan el mismo ideal. Devuelve true si dos listas de polinomios *polylist1* y *polylist2*, supuestas bases de Groebner, generan el mismo ideal, o false en caso contrario. Eso equivale a comprobar si cada polinomio de la primera base se reduce a 0 módulo la segunda base y viceversa. Nótese que en el ejemplo que sigue la primera lista no es una base de Groebner, por lo que el resultado es false.

```
(%i1) poly_grobner_equal([y+x,x-y],[x,y],[x,y]);
(%o1)                          false
```

poly_grobner_subsetp (*polylist1*, *polylist2*, *varlist*) [Función]
poly_grobner_subsetp comprueba si el ideal generado por *polylist1* está contenido en el ideal generado por *polylist2*. Para que esta comprobación tenga éxito, *polylist2* debe ser una base de Groebner.

`poly_grobner_member` (*poly*, *polylist*, *varlist*) [Función]
Devuelve `true` si el polinomio *poly* pertenece al ideal generado por la lista de polinomios *polylist*, la cual se supone una base de Groebner. Devolverá `false` en caso contrario.

`poly_ideal_saturation1` (*polylist*, *poly*, *varlist*) [Función]
Devuelve la base de Groebner reducida de la saturación del ideal

$$I(polylist) : poly^{\infty}$$

Desde un punto de vista geométrico, sobre un campo algebraicamente cerrado, este es el conjunto de polinomios del ideal generado por *polylist* que no se anulan sobre la variedad de *poly*.

`poly_ideal_saturation` (*polylist1*, *polylist2*, *varlist*) [Función]
Devuelve la base de Groebner reducida de la saturación del ideal

$$I(polylist1) : I(polylist2)^{\infty}$$

Desde un punto de vista geométrico, sobre un campo algebraicamente cerrado, este es el conjunto de polinomios del ideal generado por *polylist1* que no se anulan sobre la variedad de *polylist2*.

`poly_ideal_polysaturation1` (*polylist1*, *polylist2*, *varlist*) [Función]
polylist2 es una lista de n poliomios `[poly1,...,polyn]`. Devuelve la base de Groebner reducida del ideal

$$I(polylist) : poly1^{\infty} : ... : polyn^{\infty}$$

obtenida a partir de una secuencia de saturaciones sucesivas de los polinomios de la lista *polylist2* del ideal generado por la lista de polinomios *polylist1*.

`poly_ideal_polysaturation` (*polylist*, *polylistlist*, *varlist*) [Función]
polylistlist es una lista de n listas de polinomios `[polylist1,...,polylistn]`. Devuelve la base de Groebner reducida de la saturación del ideal

$$I(polylist) : I(polylist_1)^{\infty} : ... : I(polylist_n)^{\infty}$$

`poly_saturation_extension` (*poly*, *polylist*, *varlist1*, *varlist2*) [Función]
`poly_saturation_extension` ejecuta el truco de Rabinowitz.

`poly_polysaturation_extension` (*poly*, *polylist*, *varlist1*, *varlist2*) [Función]

57 impdiff

57.1 Funciones y variables para impdiff

`implicit_derivative` (*f*,*indvarlist*,*orderlist*,*depvar*) [Función]

Calcula las derivadas implícitas de funciones multivariantes. *f* es una función array, los índices son los grados de las derivadas en el orden establecido en *indvarlist*, *indvarlist* es la lista de variables independientes, *orderlist* es el orden deseado y *depvar* es la variable dependiente.

Antes de hacer uso de esta función ejecútese `load("impdiff")`.

58 interpol

58.1 Introducción a interpol

El paquete `interpol` desarrolla los métodos de interpolación polinómica de Lagrange, lineal y de *splines* cúbicos.

Para comentarios, fallos o sugerencias, contactar con *'mario ARROBA edu PUNTO xunta PUNTO es'*.

58.2 Funciones y variables para interpol

`lagrange` (*points*) [Función]
`lagrange` (*points*, *option*) [Función]

Calcula el polinomio de interpolación por el método de Lagrange. El argumento *points* debe ser:

- una matriz de dos columnas, `p:matrix([2,4],[5,6],[9,3])`,
- una lista de pares de números, `p: [[2,4],[5,6],[9,3]]`,
- una lista de números, `p: [4,6,3]`, en cuyo caso las abscisas se asignarán automáticamente a 1, 2, 3, etc.

En los dos primeros casos los pares se ordenan con respecto a la primera coordenada antes de proceder a los cálculos.

Mediante el argumento *option* es posible seleccionar el nombre de la variable independiente, que por defecto es `'x`; para definir otra, escríbase algo como `varname='z`.

Téngase en cuenta que cuando se trabaja con polinomios de grado alto, los cálculos con números decimales en coma flotante pueden ser muy inestables.

Véanse también `linearinterpol`, `cspline` y `ratinterpol`.

Ejemplos:

```
(%i1) load("interpol")$
(%i2) p:[[7,2],[8,2],[1,5],[3,2],[6,7]]$
(%i3) lagrange(p);
       (x - 7) (x - 6) (x - 3) (x - 1)
(%o3)  -------------------------------
                     35
   (x - 8) (x - 6) (x - 3) (x - 1)
 - -------------------------------
                12
   7 (x - 8) (x - 7) (x - 3) (x - 1)
 + ---------------------------------
                 30
   (x - 8) (x - 7) (x - 6) (x - 1)
 - -------------------------------
                60
   (x - 8) (x - 7) (x - 6) (x - 3)
 + -------------------------------
```

```
                84
(%i4) f(x):=''%;
              (x - 7) (x - 6) (x - 3) (x - 1)
(%o4)  f(x) := -------------------------------
                              35
   (x - 8) (x - 6) (x - 3) (x - 1)
 - -------------------------------
                12
   7 (x - 8) (x - 7) (x - 3) (x - 1)
 + ---------------------------------
                 30
   (x - 8) (x - 7) (x - 6) (x - 1)
 - -------------------------------
                60
   (x - 8) (x - 7) (x - 6) (x - 3)
 + -------------------------------
                84
(%i5) /* Evaluate the polynomial at some points */
      expand(map(f,[2.3,5/7,%pi]));
                                    4          3           2
                    919062  73 %pi    701 %pi    8957 %pi
(%o5)  [- 1.567535, ------, ------- - -------- + ---------
                    84035     420       210         420
                                            5288 %pi   186
                                          - -------- + ---]
                                              105       5
(%i6) %,numer;
(%o6) [- 1.567535, 10.9366573451538, 2.89319655125692]
(%i7) load("draw")$  /* load draw package */
(%i8) /* Plot the polynomial together with points */
      draw2d(
        color      = red,
        key        = "Lagrange polynomial",
        explicit(f(x),x,0,10),
        point_size = 3,
        color      = blue,
        key        = "Sample points",
        points(p))$
(%i9) /* Change variable name */
      lagrange(p, varname=w);
       (w - 7) (w - 6) (w - 3) (w - 1)
(%o9)  -------------------------------
                     35
   (w - 8) (w - 6) (w - 3) (w - 1)
 - -------------------------------
                12
   7 (w - 8) (w - 7) (w - 3) (w - 1)
```

```
+ --------------------------------
                   30
  (w - 8) (w - 7) (w - 6) (w - 1)
- --------------------------------
                  60
  (w - 8) (w - 7) (w - 6) (w - 3)
+ --------------------------------
                  84
```

`charfun2` (*x*, `a`, *b*) [Función]

Devuelve `true` si el número *x* pertenece al intervalo $[a, b)$, y `false` en caso contrario.

`linearinterpol` (*points*) [Función]
`linearinterpol` (*points*, *option*) [Función]

Calcula rectas de interpolación. El argumento *points* debe ser:

- una matriz de dos columnas, `p:matrix([2,4],[5,6],[9,3])`,
- una lista de pares de números, `p: [[2,4],[5,6],[9,3]]`,
- una lista de números, `p: [4,6,3]`, en cuyo caso las abscisas se asignarán automáticamente a 1, 2, 3, etc.

En los dos primeros casos los pares se ordenan con respecto a la primera coordenada antes de proceder a los cálculos.

Mediante el argumento *option* es posible seleccionar el nombre de la variable independiente, que por defecto es `'x`; para definir otra, escríbase algo como `varname='z`.

Véanse también `lagrange`, `cspline` y `ratinterpol`.

Ejemplos:

```
(%i1) load("interpol")$
(%i2) p: matrix([7,2],[8,3],[1,5],[3,2],[6,7])$
(%i3) linearinterpol(p);
        13   3 x
(%o3)  (-- - ---) charfun2(x, minf, 3)
        2     2
 + (x - 5) charfun2(x, 7, inf) + (37 - 5 x) charfun2(x, 6, 7)
    5 x
 + (--- - 3) charfun2(x, 3, 6)
     3

(%i4) f(x):=''%;
                 13   3 x
(%o4)  f(x) := (-- - ---) charfun2(x, minf, 3)
                 2     2
 + (x - 5) charfun2(x, 7, inf) + (37 - 5 x) charfun2(x, 6, 7)
    5 x
 + (--- - 3) charfun2(x, 3, 6)
     3
(%i5)  /* Evaluate the polynomial at some points */
```

```
       map(f,[7.3,25/7,%pi]);
                              62  5 %pi
(%o5)                 [2.3, --, ----- - 3]
                              21    3
(%i6) %,numer;
(%o6)  [2.3, 2.952380952380953, 2.235987755982989]
(%i7) load("draw")$  /* load draw package */
(%i8)  /* Plot the polynomial together with points */
       draw2d(
         color      = red,
         key        = "Linear interpolator",
         explicit(f(x),x,-5,20),
         point_size = 3,
         color      = blue,
         key        = "Sample points",
         points(args(p)))$
(%i9)  /* Change variable name */
       linearinterpol(p, varname='s);
       13   3 s
(%o9) (-- - ---) charfun2(s, minf, 3)
       2     2
 + (s - 5) charfun2(s, 7, inf) + (37 - 5 s) charfun2(s, 6, 7)
    5 s
 + (--- - 3) charfun2(s, 3, 6)
     3
```

cspline (*points*) [Función]
cspline (*points*, *option1*, *option2*, ...) [Función]

Calcula el polinomio de interpolación por el método de los *splines* cúbicos. El argumento *points* debe ser:

- una matriz de dos columnas, `p:matrix([2,4],[5,6],[9,3])`,
- una lista de pares de números, `p: [[2,4],[5,6],[9,3]]`,
- una lista de números, `p: [4,6,3]`, en cuyo caso las abscisas se asignarán automáticamente a 1, 2, 3, etc.

En los dos primeros casos los pares se ordenan con respecto a la primera coordenada antes de proceder a los cálculos.

Esta función dispone de tres opciones para acomodarse a necesidades concretas:

- `'d1`, por defecto `'unknown`, es la primera derivada en x_1; si toma el valor `'unknown`, la segunda derivada en x_1 se iguala a 0 (*spline* cúbico natural); en caso de tomar un valor numérico, la segunda derivada se calcula en base a este número.
- `'dn`, por defecto `'unknown`, es la primera derivada en x_n; si toma el valor `'unknown`, la segunda derivada en x_n se iguala a 0 (*spline* cúbico natural); en caso de tomar un valor numérico, la segunda derivada se calcula en base a este número.

- `'varname`, por defecto `'x`, es el nombre de la variable independiente.

Véanse también `lagrange`, `linearinterpol` y `ratinterpol`.

Ejemplos:

```
(%i1) load("interpol")$
(%i2) p:[[7,2],[8,2],[1,5],[3,2],[6,7]]$
(%i3) /* Unknown first derivatives at the extremes
         is equivalent to natural cubic splines */
      cspline(p);
              3         2
        1159 x    1159 x    6091 x   8283
(%o3)  (------- - ------- - ------ + ----) charfun2(x, minf, 3)
         3288      1096      3288    1096
            3         2
      2587 x    5174 x    494117 x   108928
 + (- ------- + ------- - -------- + ------) charfun2(x, 7, inf)
       1644       137       1644      137
          3          2
    4715 x    15209 x    579277 x   199575
 + (------- - -------- + -------- - ------) charfun2(x, 6, 7)
     1644       274        1644      274
            3         2
      3287 x    2223 x    48275 x   9609
 + (- ------- + ------- - ------- + ----) charfun2(x, 3, 6)
       4932       274       1644     274

(%i4) f(x):=''%$
(%i5) /* Some evaluations */
      map(f,[2.3,5/7,%pi]), numer;
(%o5) [1.991460766423356, 5.823200187269903, 2.227405312429507]
(%i6) load("draw")$  /* load draw package */
(%i7) /* Plotting interpolating function */
      draw2d(
        color      = red,
        key        = "Cubic splines",
        explicit(f(x),x,0,10),
        point_size = 3,
        color      = blue,
        key        = "Sample points",
        points(p))$
(%i8) /* New call, but giving values at the derivatives */
      cspline(p,d1=0,dn=0);
              3          2
        1949 x    11437 x    17027 x   1247
(%o8)  (------- - -------- + ------- + ----) charfun2(x, minf, 3)
         2256       2256       2256     752
            3          2
```

```
      1547 x    35581 x    68068 x   173546
 + (- ------- + -------- - ------- + ------) charfun2(x, 7, inf)
        564       564        141      141
         3         2
    607 x    35147 x    55706 x   38420
 + (------ - -------- + ------- - -----) charfun2(x, 6, 7)
     188       564        141      47
            3         2
      3895 x    1807 x    5146 x   2148
 + (- ------- + ------- - ------ + ----) charfun2(x, 3, 6)
       5076       188      141      47
(%i8) /* Defining new interpolating function */
      g(x):=''%$
(%i9) /* Plotting both functions together */
      draw2d(
        color      = black,
        key        = "Cubic splines (default)",
        explicit(f(x),x,0,10),
        color      = red,
        key        = "Cubic splines (d1=0,dn=0)",
        explicit(g(x),x,0,10),
        point_size = 3,
        color      = blue,
        key        = "Sample points",
        points(p))$
```

`ratinterpol` (*points*, *numdeg*) [Función]
`ratinterpol` (*points*, *numdeg*, *option1*) [Función]

Genera el interpolador racional para los datos dados por *points* y con grado *numdeg* en el numerador; el grado del denominador se calcula automáticamente. El argumento *points* debe ser:

- una matriz de dos columnas, `p:matrix([2,4],[5,6],[9,3])`,
- una lista de pares de números, `p: [[2,4],[5,6],[9,3]]`,
- una lista de números, `p: [4,6,3]`, en cuyo caso las abscisas se asignarán automáticamente a 1, 2, 3, etc.

En los dos primeros casos los pares se ordenan con respecto a la primera coordenada antes de proceder a los cálculos.

Esta función dispone de una opción para acomodarse a necesidades concretas:

- `'varname`, por defecto `'x`, es el nombre de la variable independiente.

Véanse también `lagrange`, `linearinterpol`, `cspline`, `minpack_lsquares` y `lbfgs`.

Ejemplos:

```
(%i1) load("interpol")$
(%i2) load("draw")$
(%i3) p:[[7.2,2.5],[8.5,2.1],[1.6,5.1],[3.4,2.4],[6.7,7.9]]$
(%i4) for k:0 thru length(p)-1 do
```

```
draw2d(
  explicit(ratinterpol(p,k),x,0,9),
  point_size = 3,
  points(p),
  title = concat("Grado del numerador = ",k),
  yrange=[0,10])$
```

59 lapack

59.1 Introducción a lapack

`lapack` es una traducción automática a Common Lisp (con el programa `f2c`) de la librería LAPACK escrita en Fortran.

59.2 Funciones y variables para lapack

`dgeev` (*A*) [Función]
`dgeev` (*A*, *right_p*, *left_p*) [Función]

Calcula los autovalores y, opcionalmente, también los autovectores de la matriz A. Todos los elementos de A deben ser enteros o números decimales en coma flotante. Además, A debe ser cuadrada (igual número de filas que de columnas) y puede ser o no simétrica.

`dgeev(A)` calcula sólo los autovalores de A. `dgeev(A, right_p, left_p)` calcula los autovalores de A y los autovectores por la derecha cuando *right_p* = `true`, y los autovectores por la izquierda cuando *left_p* = `true`.

La función devuelve una lista de tres elementos. El primer elemento es una lista con los autovalores. El segundo elemento es `false` o la matriz de autovectores por la derecha. El tercer elemento es `false` o la matriz de autovectores por la izquierda.

El autovector por la derecha $v(j)$ (la j-ésima columna de la matriz de autovectores por la derecha) satisface

$A.v(j) = lambda(j).v(j)$

donde $lambda(j)$ es su autovalor asociado.

El autovector por la izquierda $u(j)$ (la j-ésima columna de la matriz de autovectores por la izquierda) satisface

$u(j) **H.A = lambda(j).u(j) **H$

donde $u(j) **H$ denota la transpuesta conjugada de $u(j)$.

La función de Maxima `ctranspose` calcula la transpuesta conjugada.

Los autovectores calculados están normalizados para que su norma euclídea valga 1 y su componente mayor tenga su parte imaginaria igual a cero.

Ejemplo:

```
(%i1) load ("lapack")$
(%i2) fpprintprec : 6;
(%o2)                          6
(%i3) M : matrix ([9.5, 1.75], [3.25, 10.45]);
                         [ 9.5   1.75  ]
(%o3)                    [             ]
                         [ 3.25  10.45 ]
(%i4) dgeev (M);
(%o4)          [[7.54331, 12.4067], false, false]
(%i5) [L, v, u] : dgeev (M, true, true);
                            [ - .666642  - .515792 ]
```

```
(%o5) [[7.54331, 12.4067], [                      ],
                           [  .745378   - .856714 ]
                                                  [ - .856714  - .745378 ]
                                                  [                      ]]
                                                  [  .515792   - .666642 ]
(%i6) D : apply (diag_matrix, L);
                        [ 7.54331     0    ]
(%o6)                   [                  ]
                        [    0     12.4067 ]
(%i7) M . v - v . D;
                  [      0.0       - 8.88178E-16 ]
(%o7)             [                              ]
                  [ - 8.88178E-16       0.0      ]
(%i8) transpose (u) . M - D . transpose (u);
                       [ 0.0  - 4.44089E-16 ]
(%o8)                  [                    ]
                       [ 0.0       0.0      ]
```

dgeqrf (*A*) [Función]

Calcula la descomposición QR de la matriz A. Todos los elementos de A deben ser enteros o números reales. No es necesario que A tenga el mismo número de filas que de columnas.

La función devuelve una lista con dos elementos; el primero es la matriz cuadrada ortonormal Q, con el mismo número de filas que A, y el segundo es la matriz triangular superior R, de iguales dimensiones que A. El producto `Q . R`, siendo `"."` el operador de la multiplicación matricial, es igual a A, ignorando errores de redondeo.

```
(%i1) load ("lapack") $
(%i2) fpprintprec : 6 $
(%i3) M : matrix ([1, -3.2, 8], [-11, 2.7, 5.9]) $
(%i4) [q, r] : dgeqrf (M);
       [ - .0905357  .995893  ]
(%o4) [[                      ],
       [  .995893    .0905357 ]
                                [ - 11.0454   2.97863   5.15148 ]
                                [                               ]]
                                [     0      - 2.94241  8.50131 ]
(%i5) q . r - M;
         [ - 7.77156E-16   1.77636E-15   - 8.88178E-16 ]
(%o5)    [                                            ]
         [      0.0       - 1.33227E-15   8.88178E-16  ]
(%i6) mat_norm (%, 1);
(%o6)                     3.10862E-15
```

dgesv (*A*, *b*) [Función]

Calcula la solución x de la ecuación $Ax = b$, siendo A una matriz cuadrada y b otra matriz con el mismo número de filas que A y un número arbitrario de columnas. Las dimensiones de la solución x son las mismas de b.

Los elementos de A y b deben ser reducibles a números decimales si se les aplica la función `float`, por lo que tales elementos pueden en principio ser de cualquier tipo numérico, constantes numéricas simbólicas o cualesquiera expresiones reducibles a un número decimal. Los elementos de x son siempre números decimales. Todas las operaciones aritméticas se realizan en coma flotante.

`dgesv` calcula la solución mediante la descomposición LU de A.

Ejemplos:

`dgesv` calcula la solución x de la ecuación $Ax = b$.

```
(%i1) A : matrix ([1, -2.5], [0.375, 5]);
                                [   1    - 2.5 ]
(%o1)                           [              ]
                                [ 0.375    5   ]
(%i2) b : matrix ([1.75], [-0.625]);
                                   [  1.75   ]
(%o2)                              [         ]
                                   [ - 0.625 ]
(%i3) x : dgesv (A, b);
                              [  1.210526315789474  ]
(%o3)                         [                     ]
                              [ - 0.215789473684211 ]
(%i4) dlange (inf_norm, b - A.x);
(%o4)                                 0.0
```

b una matriz con el mismo número de filas que A y un número arbitrario de columnas. Las dimensiones de x son las mismas de b.

```
(%o0)                                  done
(%i1) A : matrix ([1, -0.15], [1.82, 2]);
                                [  1    - 0.15 ]
(%o1)                           [              ]
                                [ 1.82    2    ]
(%i2) b : matrix ([3.7, 1, 8], [-2.3, 5, -3.9]);
                               [  3.7   1    8   ]
(%o2)                          [                 ]
                               [ - 2.3  5  - 3.9 ]
(%i3) x : dgesv (A, b);
       [  3.103827540695117   1.20985481742191    6.781786185657722  ]
(%o3)  [                                                            ]
       [ - 3.974483062032557  1.399032116146062  - 8.121425428948527 ]
(%i4) dlange (inf_norm, b - A . x);
(%o4)                        1.1102230246251565E-15
```

Los elementos de A y b deben ser reducibles a números decimales.

```
(%i1) A : matrix ([5, -%pi], [1b0, 11/17]);
                                [   5    - %pi ]
                                [              ]
(%o1)                           [         11   ]
                                [ 1.0b0   --   ]
```

```
                                    [         17   ]
(%i2) b : matrix ([%e], [sin(1)]);
                                    [   %e   ]
(%o2)                               [        ]
                                    [ sin(1) ]
(%i3) x : dgesv (A, b);
                                 [ 0.690375643155986 ]
(%o3)                            [                   ]
                                 [ 0.233510982552952 ]
(%i4) dlange (inf_norm, b - A . x);
(%o4)                          2.220446049250313E-16
```

`dgesvd` (*A*) [Función]
`dgesvd` (*A*, *left_p*, *right_p*) [Función]

Calcula la descomposición singular (SVD, en inglés) de la matriz A, que contiene los valores singulares y, opcionalmente, los vectores singulares por la derecha o por la izquierda. Todos los elementos de A deben ser enteros o números decimales en coma flotante. La matriz A puede ser cuadrada o no (igual número de filas que de columnas).

Sea m el número de filas y n el de columnas de A. La descomposición singular de A consiste en calcular tres matrices: U, *Sigma* y $V\hat{}T$, tales que

$A = U.Sigma.V^T$

donde U es una matriz unitaria m-por-m, *Sigma* es una matriz diagonal m-por-n y $V\hat{}T$ es una matriz unitaria n-por-n.

Sea $sigma[i]$ un elemento diagonal de $Sigma$, esto es, $Sigma[i, i] = sigma[i]$. Los elementos $sigma[i]$ se llaman valores singulares de A, los cuales son reales y no negativos, siendo devueltos por la función `dgesvd` en orden descendente.

Las primeras $min(m, n)$ columnas de U y V son los vectores singulares izquierdo y derecho de A. Nótese que `dgesvd` devuelve la transpuesta de V, no la propia matriz V.

`dgesvd(A)` calcula únicamente los valores singulares de A. `dgesvd(A, left_p, right_p)` calcula los valores singulares de A y los vectores sigulares por la izquierda cuando *left_p* = `true`, y los vectores sigulares por la derecha cuando *right_p* = `true`.

La función devuelve una lista de tres elementos. El primer elemento es una lista con los valores singulares. El segundo elemento es `false` o la matriz de vectores singulares por la izquierda. El tercer elemento es `false` o la matriz de vectores singulares por la derecha.

Ejemplo:

```
(%i1) load ("lapack")$
(%i2) fpprintprec : 6;
(%o2)                           6
(%i3) M: matrix([1, 2, 3], [3.5, 0.5, 8], [-1, 2, -3], [4, 9, 7]);
                          [  1    2    3  ]
                          [               ]
                          [ 3.5  0.5   8  ]
```

```
(%o3)                    [               ]
                         [ - 1   2   - 3 ]
                         [               ]
                         [  4    9    7  ]
(%i4) dgesvd (M);
(%o4)      [[14.4744, 6.38637, .452547], false, false]
(%i5) [sigma, U, VT] : dgesvd (M, true, true);
(%o5) [[14.4744, 6.38637, .452547],
[ - .256731  .00816168   .959029    - .119523 ]
[                                             ]
[ - .526456   .672116   - .206236   - .478091 ]
[                                             ],
[  .107997   - .532278  - .0708315  - 0.83666 ]
[                                             ]
[ - .803287  - .514659  - .180867    .239046  ]
[ - .374486  - .538209  - .755044 ]
[                               ]
[  .130623   - .836799   0.5317   ]]
[                               ]
[ - .917986   .100488    .383672  ]
(%i6) m : length (U);
(%o6)                          4
(%i7) n : length (VT);
(%o7)                          3
(%i8) Sigma:
       genmatrix(lambda ([i, j], if i=j then sigma[i] else 0),
                 m, n);
                 [ 14.4744     0        0    ]
                 [                           ]
                 [    0     6.38637     0    ]
(%o8)            [                           ]
                 [    0        0     .452547 ]
                 [                           ]
                 [    0        0        0    ]
(%i9) U . Sigma . VT - M;
          [  1.11022E-15        0.0       1.77636E-15 ]
          [                                           ]
          [  1.33227E-15    1.66533E-15       0.0     ]
(%o9)     [                                           ]
          [ - 4.44089E-16  - 8.88178E-16  4.44089E-16 ]
          [                                           ]
          [  8.88178E-16    1.77636E-15   8.88178E-16 ]
(%i10) transpose (U) . U;
       [     1.0      5.55112E-17    2.498E-16     2.77556E-17  ]
       [                                                        ]
       [ 5.55112E-17      1.0       5.55112E-17    4.16334E-17  ]
(%o10) [                                                        ]
```

```
                [  2.498E-16   5.55112E-17       1.0       - 2.08167E-16 ]
                [                                                        ]
                [ 2.77556E-17  4.16334E-17  - 2.08167E-16       1.0      ]
        (%i11) VT . transpose (VT);
                   [      1.0           0.0      - 5.55112E-17 ]
                   [                                           ]
        (%o11)     [      0.0           1.0       5.55112E-17  ]
                   [                                           ]
                   [ - 5.55112E-17  5.55112E-17       1.0      ]
```

dlange (*norm*, *A*) [Función]
zlange (*norm*, *A*) [Función]

Calcula una norma o seudonorma de la matriz A.

`max` Calcula $max(abs(A(i,j)))$, siendo i y j números de filas y columnas, respectivamente, de A. Nótese que esta función no es una norma matricial.

`one_norm` Calcula la norma $L[1]$ de A, esto es, el máximo de la suma de los valores absolutos de los elementos de cada columna.

`inf_norm` Calcula la norma $L[inf]$ de A, esto es, el máximo de la suma de los valores absolutos de los elementos de cada fila.

`frobenius`
Calcula la norma de Frobenius de A, esto es, la raíz cuadrada de la suma de los cuadrados de los elementos de la matriz.

dgemm (*A*, *B*) [Función]
dgemm (*A*, *B*, *options*) [Función]

Calcula el producto de dos matrices y, opcionalmente, suma este producto con una tercera matriz.

En su forma más simple, `dgemm(A, B)` calcula el producto de las matrices reales A y B.

En la segunda forma, `dgemm` calcula $alpha * A * B + beta * C$, donde A, B y C son matrices reales de dimensiones apropiadas, siendo *alpha* y *beta* números reales. De forma opcional, tanto A como B pueden transponerse antes de calcular su producto. Los parámetros adicionales se pueden especificar en cualquier orden, siendo su sintaxis de la forma `clave=valor`. Las claves reconocidas son:

`C` La matriz C que debe ser sumada. El valor por defecto es `false`, lo que significa que no se sumará ninguna matriz.

`alpha` El producto de A por B se multiplicará por este vaalor. El valor por defecto es 1.

`beta` Si se da la matriz C, se multiplicará por este valor antes de ser sumada. El valor por defecto es 0, lo que significa que C no se suma, incluso estando presente. Por lo tanto, téngase cuidado en especificar un valor no nulo para *beta*.

`transpose_a`
Si toma el valor `true`, se utilizará la transpuesta de A, en lugar de la propia matriz A, en el producto. El valor por defecto es `false`.

`transpose_b`
: Si toma el valor `true`, se utilizará la transpuesta de B, en lugar de la propia matriz B, en el producto. El valor por defecto es `false`.

```
(%i1) load ("lapack")$
(%i2) A : matrix([1,2,3],[4,5,6],[7,8,9]);
                                  [ 1  2  3 ]
                                  [         ]
(%o2)                             [ 4  5  6 ]
                                  [         ]
                                  [ 7  8  9 ]
(%i3) B : matrix([-1,-2,-3],[-4,-5,-6],[-7,-8,-9]);
                                [ - 1  - 2  - 3 ]
                                [               ]
(%o3)                           [ - 4  - 5  - 6 ]
                                [               ]
                                [ - 7  - 8  - 9 ]
(%i4) C : matrix([3,2,1],[6,5,4],[9,8,7]);
                                  [ 3  2  1 ]
                                  [         ]
(%o4)                             [ 6  5  4 ]
                                  [         ]
                                  [ 9  8  7 ]
(%i5) dgemm(A,B);
                          [ - 30.0   - 36.0   - 42.0  ]
                          [                           ]
(%o5)                     [ - 66.0   - 81.0   - 96.0  ]
                          [                           ]
                          [ - 102.0  - 126.0  - 150.0 ]
(%i6) A . B;
                             [ - 30   - 36   - 42  ]
                             [                     ]
(%o6)                        [ - 66   - 81   - 96  ]
                             [                     ]
                             [ - 102  - 126  - 150 ]
(%i7) dgemm(A,B,transpose_a=true);
                          [ - 66.0  - 78.0   - 90.0  ]
                          [                          ]
(%o7)                     [ - 78.0  - 93.0   - 108.0 ]
                          [                          ]
                          [ - 90.0  - 108.0  - 126.0 ]
(%i8) transpose(A) . B;
                               [ - 66  - 78   - 90  ]
                               [                    ]
(%o8)                          [ - 78  - 93   - 108 ]
                               [                    ]
                               [ - 90  - 108  - 126 ]
```

```
(%i9) dgemm(A,B,c=C,beta=1);
                          [ - 27.0  - 34.0   - 41.0  ]
                          [                          ]
(%o9)                     [ - 60.0  - 76.0   - 92.0  ]
                          [                          ]
                          [ - 93.0  - 118.0  - 143.0 ]
(%i10) A . B + C;
                             [ - 27  - 34   - 41  ]
                             [                    ]
(%o10)                       [ - 60  - 76   - 92  ]
                             [                    ]
                             [ - 93  - 118  - 143 ]
(%i11) dgemm(A,B,c=C,beta=1, alpha=-1);
                             [ 33.0   38.0   43.0  ]
                             [                     ]
(%o11)                       [ 72.0   86.0   100.0 ]
                             [                     ]
                             [ 111.0  134.0  157.0 ]
(%i12) -A . B + C;
                                [ 33   38   43  ]
                                [               ]
(%o12)                          [ 72   86   100 ]
                                [               ]
                                [ 111  134  157 ]
```

60 lbfgs

60.1 Introducción a lbfgs

La función `lbfgs` implementa el llamado algoritmo L-BFGS [1] para resolver problemas de minimización sin restricciones mediante una técnica *cuasi-Newton con memoria limitada* (BFGS). El término memoria limitada procede del hecho de que se almacena una aproximación de rango bajo de la inversa de la matriz hessiana, en lugar de la matriz completa. El programa fue originalmente escrito en Fortran [2] por Jorge Nocedal, incorporando algunas funciones escritas originalmente por Jorge J. Moré y David J. Thuente, traducidas posteriormente a Lisp automáticamente con el programa `f2cl`. El paquete `lbfgs` contiene el código traducido, junto con una función interfaz que para controlar ciertos detalles.

Referencias:

[1] D. Liu and J. Nocedal. "On the limited memory BFGS method for large scale optimization". *Mathematical Programming B* 45:503–528 (1989)

[2] http://netlib.org/opt/lbfgs_um.shar

60.2 Funciones y variables para lbfgs

`lbfgs` (*FOM*, *X*, *X0*, *epsilon*, *iprint*) [Función]
`lbfgs` ([*FOM*, *grad*] *X*, *X0*, *epsilon*, *iprint*) [Function]

Encuentra una solución aproximada para el problema de minimización sin restricciones de la función objetivo *FOM* para la lista de variables *X*, partiendo de los estimadores iniciales *X0*, de tal manera que $norm(grad(FOM)) < epsilon * max(1, norm(X))$.

Si el argumento *grad* está presente, debe ser el gradiente de *FOM* respecto de las variables *X*. *grad* puede ser una lista o una función que devuelva una lista con igual número de elementos que *X*. Si el argumento no está presente, el gradiente se calcula automáticamente mediante derivación simbólica. Si *FOM* es una función, el gradiente *grad* debe ser suministrado por el usuario.

El algoritmo utilizado es una técnica *cuasi-Newton con memoria limitada* (BFGS) [1]. El término *memoria limitada* procede del hecho de que se almacena una aproximación de rango bajo de la inversa de la matriz hessiana, en lugar de la matriz completa. Cada iteración del algoritmo es una búsqueda a lo largo de una recta, cuya dirección se calcula a partir de la matriz inversa aproximada del hessiano. La función objetivo decrece siempre tras cada búsqueda exitosa a lo largo de la recta; además, casi siempre decrece también el módulo del gradiente de la función.

El argumento *iprint* controla los mensajes de progreso que envía la función `lbfgs`.

`iprint[1]`

iprint`[1]` controla la frecuencia con la que se emiten los mensajes.

`iprint[1] < 0`

No se envían mensajes.

`iprint[1] = 0`

Mensajes únicamente en la primera y última iteraciones.

`iprint[1] > 0`
: Imprime un mensaje cada *iprint*`[1]` iteraciones.

`iprint[2]`
: *iprint*`[2]` controla la cantidad de información contenida en los mensajes.

`iprint[2] = 0`
: Imprime contador de iteraciones, número de evaluaciones de *FOM*, valor de *FOM*, módulo del gradiente de *FOM* y amplitud del paso.

`iprint[2] = 1`
: Igual que *iprint*`[2] = 0`, incluyendo *X0* y el gradiente de *FOM* evaluado en *X0*.

`iprint[2] = 2`
: Igual que *iprint*`[2] = 1`, incluyendo los valores de X en cada iteración.

`iprint[2] = 3`
: Igual que *iprint*`[2] = 2`, incluyendo el gradiente de *FOM* en cada iteración.

Las columnas devueltas por `lbfgs` son las siguientes:

`I` Número de iteraciones. Se incremente tras cada búsqueda a lo largo de una recta.

`NFN` Número de evaluaciones de la función objetivo.

`FUNC` Valor de la función objetivo al final de cada iteración.

`GNORM` Módulo del gradiente de la función objetivo al final de cada iteración.

`STEPLENGTH`
: Un parámetro interno del algoritmo de búsqueda.

Para más información sobre el algoritmo se puede acudir a los comentarios en el código original en Fortran [2].

Véanse también `lbfgs_nfeval_max` y `lbfgs_ncorrections`.

Referencias:

[1] D. Liu and J. Nocedal. "On the limited memory BFGS method for large scale optimization". *Mathematical Programming B* 45:503–528 (1989)

[2] http://netlib.org/opt/lbfgs_um.shar

Ejemplos:

La misma función objetivo utilizada por FGCOMPUTE en el programa sdrive.f del paquete LBFGS de Netlib. Nótese que las variables en cuestión están subindicadas. La función objetivo tiene un mínimo exacto igual a cero en $u[k] = 1$, para $k = 1, ..., 8$.

```
(%i1) load ("lbfgs")$
(%i2) t1[j] := 1 - u[j];
(%o2)                      t1  := 1 - u
                             j         j
```

```
(%i3) t2[j] := 10*(u[j + 1] - u[j]^2);
                                                2
(%o3)                t2  := 10 (u      - u )
                       j         j + 1    j
(%i4) n : 8;
(%o4)                           8
(%i5) FOM : sum (t1[2*j - 1]^2 + t2[2*j - 1]^2, j, 1, n/2);
                   2 2           2              2 2           2
(%o5) 100 (u  - u )  + (1 - u )  + 100 (u  - u )  + (1 - u )
            8    7           7           6    5           5
                       2 2           2              2 2           2
        + 100 (u  - u )  + (1 - u )  + 100 (u  - u )  + (1 - u )
                4    3           3           2    1           1
(%i6) lbfgs (FOM, '[u[1],u[2],u[3],u[4],u[5],u[6],u[7],u[8]],
       [-1.2, 1, -1.2, 1, -1.2, 1, -1.2, 1], 1e-3, [1, 0]);
 ************************************************
   N=    8   NUMBER OF CORRECTIONS=25
        INITIAL VALUES
  F=  9.680000000000000D+01   GNORM=  4.657353755084533D+02
 ************************************************
```

I	NFN	FUNC	GNORM	STEPLENGTH
1	3	1.651479526340304D+01	4.324359291335977D+00	7.926153934390631D-04
2	4	1.650209316638371D+01	3.575788161060007D+00	1.000000000000000D+00
3	5	1.645461701312851D+01	6.230869903601577D+00	1.000000000000000D+00
4	6	1.636867301275588D+01	1.177589920974980D+01	1.000000000000000D+00
5	7	1.612153014409201D+01	2.292797147151288D+01	1.000000000000000D+00
6	8	1.569118407390628D+01	3.687447158775571D+01	1.000000000000000D+00
7	9	1.510361958398942D+01	4.501931728123680D+01	1.000000000000000D+00
8	10	1.391077875774294D+01	4.526061463810632D+01	1.000000000000000D+00
9	11	1.165625686278198D+01	2.748348965356917D+01	1.000000000000000D+00
10	12	9.859422687859137D+00	2.111494974231644D+01	1.000000000000000D+00
11	13	7.815442521732281D+00	6.110762325766556D+00	1.000000000000000D+00
12	15	7.346380905773160D+00	2.165281166714631D+01	1.285316401779533D-01
13	16	6.330460634066370D+00	1.401220851762050D+01	1.000000000000000D+00
14	17	5.238763939851439D+00	1.702473787613255D+01	1.000000000000000D+00
15	18	3.754016790406701D+00	7.981845727704576D+00	1.000000000000000D+00
16	20	3.001238402309352D+00	3.925482944716691D+00	2.333129631296807D-01
17	22	2.794390709718290D+00	8.243329982546473D+00	2.503577283782332D-01
18	23	2.563783562918759D+00	1.035413426521790D+01	1.000000000000000D+00
19	24	2.019429976377856D+00	1.065187312346769D+01	1.000000000000000D+00
20	25	1.428003167670903D+00	2.475962450826961D+00	1.000000000000000D+00
21	27	1.197874264861340D+00	8.441707983493810D+00	4.303451060808756D-01
22	28	9.023848941942773D-01	1.113189216635162D+01	1.000000000000000D+00
23	29	5.508226405863770D-01	2.380830600326308D+00	1.000000000000000D+00
24	31	3.902893258815567D-01	5.625595816584421D+00	4.834988416524465D-01
25	32	3.207542206990315D-01	1.149444645416472D+01	1.000000000000000D+00

```
26   33  1.874468266362791D-01  3.632482152880997D+00  1.000000000000000D+00
27   34  9.575763380706598D-02  4.816497446154354D+00  1.000000000000000D+00
28   35  4.085145107543406D-02  2.087009350166495D+00  1.000000000000000D+00
29   36  1.931106001379290D-02  3.886818608498966D+00  1.000000000000000D+00
30   37  6.894000721499670D-03  3.198505796342214D+00  1.000000000000000D+00
31   38  1.443296033051864D-03  1.590265471025043D+00  1.000000000000000D+00
32   39  1.571766603154336D-04  3.098257063980634D-01  1.000000000000000D+00
33   40  1.288011776581970D-05  1.207784183577257D-02  1.000000000000000D+00
34   41  1.806140173752971D-06  4.587890233385193D-02  1.000000000000000D+00
35   42  1.769004645459358D-07  1.790537375052208D-02  1.000000000000000D+00
36   43  3.312164100763217D-10  6.782068426119681D-04  1.000000000000000D+00

 THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
 IFLAG = 0
(%o6) [u  = 1.000005339816132, u  = 1.000009942840108,
        1                        2
u  = 1.000005339816132, u  = 1.000009942840108,
 3                       4
u  = 1.000005339816132, u  = 1.000009942840108,
 5                       6
u  = 1.000005339816132, u  = 1.000009942840108]
 7                       8
```

Un problema de regresión. La función objetivo es el cuadrado medio de la diferencia entre la predicción $F(X[i])$ y el valor observado $Y[i]$. La función F es monótona y acotada (llamada en ocasiones "sigmoidal"). En este ejemplo, `lbfgs` calcula valores aproximados para los parámetros de F y `plot2d` hace una representación gráfica comparativa de F junto con los datos observados.

```
(%i1) load ("lbfgs")$
(%i2) FOM : '((1/length(X))*sum((F(X[i]) - Y[i])^2, i, 1,
                                                  length(X)));
                                 2
                 sum((F(X ) - Y ) , i, 1, length(X))
                         i     i
(%o2)            ----------------------------------
                              length(X)
(%i3) X : [1, 2, 3, 4, 5];
(%o3)                      [1, 2, 3, 4, 5]
(%i4) Y : [0, 0.5, 1, 1.25, 1.5];
(%o4)                  [0, 0.5, 1, 1.25, 1.5]
(%i5) F(x) := A/(1 + exp(-B*(x - C)));
                                    A
(%o5)             F(x) := ----------------------
                          1 + exp((- B) (x - C))
(%i6) ''FOM;
                  A               2            A                 2
(%o6) ((----------------- - 1.5)  + (----------------- - 1.25)
```

```
                  - B (5 - C)                    - B (4 - C)
                 %e            + 1              %e            + 1
                     A             2              A               2
          + (----------------- - 1)  + (----------------- - 0.5)
               - B (3 - C)                  - B (2 - C)
              %e            + 1            %e            + 1
                        2
                       A
          + --------------------)/5
               - B (1 - C)     2
             (%e            + 1)
(%i7) estimates : lbfgs (FOM, '[A, B, C], [1, 1, 1], 1e-4, [1, 0]);
*************************************************
  N=    3   NUMBER OF CORRECTIONS=25
       INITIAL VALUES
 F=  1.348738534246918D-01   GNORM=  2.000215531936760D-01
*************************************************

I  NFN  FUNC                    GNORM                   STEPLENGTH

1    3  1.177820636622582D-01  9.893138394953992D-02  8.554435968992371D-01
2    6  2.302653892214013D-02  1.180098521565904D-01  2.100000000000000D+01
3    8  1.496348495303004D-02  9.611201567691624D-02  5.257340567840710D-01
4    9  7.900460841091138D-03  1.325041647391314D-02  1.000000000000000D+00
5   10  7.314495451266914D-03  1.510670810312226D-02  1.000000000000000D+00
6   11  6.750147275936668D-03  1.914964958023037D-02  1.000000000000000D+00
7   12  5.850716021108202D-03  1.028089194579382D-02  1.000000000000000D+00
8   13  5.778664230657800D-03  3.676866074532179D-04  1.000000000000000D+00
9   14  5.777818823650780D-03  3.010740179797108D-04  1.000000000000000D+00

 THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
 IFLAG = 0
(%o7) [A = 1.461933911464101, B = 1.601593973254801,
                                                C = 2.528933072164855]
(%i8) plot2d ([F(x), [discrete, X, Y]], [x, -1, 6]), ''estimates;
(%o8)
```

Especificando el gradiente de la función objetivo en lugar de calcularlo simbólicamente.

```
(%i1) load ("lbfgs")$
(%i2) F(a, b, c) := (a - 5)^2 + (b - 3)^4 + (c - 2)^6$
(%i3) define(F_grad(a, b, c),
              map (lambda ([x], diff (F(a, b, c), x)), [a, b, c]))$
(%i4) estimates : lbfgs ([F, F_grad],
                   [a, b, c], [0, 0, 0], 1e-4, [1, 0]);
*************************************************
  N=    3   NUMBER OF CORRECTIONS=25
```

```
                INITIAL VALUES
           F=  1.700000000000000D+02   GNORM=  2.205175729958953D+02
          ************************************************

 I  NFN  FUNC                    GNORM                   STEPLENGTH

 1    2  6.632967565917637D+01  6.498411132518770D+01  4.534785987412505D-03
 2    3  4.368890936228036D+01  3.784147651974131D+01  1.000000000000000D+00
 3    4  2.685298972775191D+01  1.640262125898520D+01  1.000000000000000D+00
 4    5  1.909064767659852D+01  9.733664001790506D+00  1.000000000000000D+00
 5    6  1.006493272061515D+01  6.344808151880209D+00  1.000000000000000D+00
 6    7  1.215263596054292D+00  2.204727876126877D+00  1.000000000000000D+00
 7    8  1.080252896385329D-02  1.431637116951845D-01  1.000000000000000D+00
 8    9  8.407195124830860D-03  1.126344579730008D-01  1.000000000000000D+00
 9   10  5.022091686198525D-03  7.750731829225275D-02  1.000000000000000D+00
10   11  2.277152808939775D-03  5.032810859286796D-02  1.000000000000000D+00
11   12  6.489384688303218D-04  1.932007150271009D-02  1.000000000000000D+00
12   13  2.075791943844547D-04  6.964319310814365D-03  1.000000000000000D+00
13   14  7.349472666162258D-05  4.017449067849554D-03  1.000000000000000D+00
14   15  2.293617477985238D-05  1.334590390856715D-03  1.000000000000000D+00
15   16  7.683645404048675D-06  6.011057038099202D-04  1.000000000000000D+00

          THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
          IFLAG = 0
         (%o4) [a = 5.000086823042934, b = 3.052395429705181,
                                                 c = 1.927980629919583]
```

`lbfgs_nfeval_max` [Variable]

Valor por defecto: 100

La variable `lbfgs_nfeval_max` almacena el número máximo de evaluaciones de la función objetivo en `lbfgs`. Cuando se alcanza el valor `lbfgs_nfeval_max`, `lbfgs` devuelve el resultado logrado en la última iteración exitosa.

`lbfgs_ncorrections` [Variable]

Valor por defecto: 25

La variable `lbfgs_ncorrections` almacena el número de correcciones aplicadas a la matriz inversa aproximada del hessiano, la cual es gestionada por `lbfgs`.

61 lindstedt

61.1 Funciones y variables para lindstedt

`Lindstedt` (*eq*,*pvar*,*torder*,*ic*) [Función]

Este es el primer paso hacia un programa para resolver ecuaciones de Lindstedt. Puede resolver problemas con condiciones iniciales, las cuales pueden ser constantes arbitrarias (no pueden usarse *%k1* ni *%k2*) donde las condiciones iniciales sobre las ecuaciones de perturbación son $z[i] = 0, z'[i] = 0$ para $i > 0$. El argumento *ic* es la lista de condiciones iniciales.

Ejemplo:

```
(%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
```

Antes de hacer uso de esta función ejecútense `load("makeOrders")` y `load("lindstedt")`.

62 linearalgebra

62.1 Introducción a linearalgebra

El paquete `linearalgebra` contiene una batería de funciones para álgebra lineal.
Ejemplo:

```
(%i1) M : matrix ([1, 2], [1, 2]);
                              [ 1  2 ]
(%o1)                         [      ]
                              [ 1  2 ]
(%i2) nullspace (M);
                                 [  1  ]
                                 [     ]
(%o2)                       span([   1 ])
                                 [ - - ]
                                 [   2 ]
(%i3) columnspace (M);
                                  [ 1 ]
(%o3)                          span([   ])
                                  [ 1 ]
(%i4) ptriangularize (M - z*ident(2), z);
                             [ 1   2 - z   ]
(%o4)                        [             ]
                             [           2 ]
                             [ 0  3 z - z  ]
(%i5) M : matrix ([1, 2, 3], [4, 5, 6], [7, 8, 9]) - z*ident(3);
                        [ 1 - z    2      3   ]
                        [                     ]
(%o5)                   [   4    5 - z    6   ]
                        [                     ]
                        [   7      8    9 - z ]
(%i6) MM : ptriangularize (M, z);
                 [ 4  5 - z            6           ]
                 [                                 ]
                 [                2                ]
                 [     66        z    102 z   132   ]
                 [ 0   --      - -- + ----- + ---   ]
(%o6)            [     49        7     49     49    ]
                 [                                 ]
                 [               3        2        ]
                 [           49 z    245 z    147 z ]
                 [ 0    0    ----- - ------ - ----- ]
                 [            264      88      44   ]
(%i7) algebraic : true;
(%o7)                          true
(%i8) tellrat (MM [3, 3]);
```

```
                        3       2
(%o8)                  [z  - 15 z  - 18 z]
(%i9) MM : ratsimp (MM);
               [ 4  5 - z           6          ]
               [                               ]
               [               2               ]
(%o9)          [     66      7 z  - 102 z - 132 ]
               [ 0   --    - ------------------ ]
               [     49             49         ]
               [                               ]
               [ 0    0            0           ]
(%i10) nullspace (MM);
                         [        1         ]
                         [                  ]
                         [   2              ]
                         [  z  - 14 z - 16  ]
                         [  --------------  ]
(%o10)             span([        8         ])
                         [                  ]
                         [    2             ]
                         [   z  - 18 z - 12 ]
                         [ - -------------- ]
                         [         12       ]
(%i11) M : matrix ([1, 2, 3, 4], [5, 6, 7, 8],
                   [9, 10, 11, 12], [13, 14, 15, 16]);
                        [ 1   2   3   4  ]
                        [                ]
                        [ 5   6   7   8  ]
(%o11)                  [                ]
                        [ 9   10  11  12 ]
                        [                ]
                        [ 13  14  15  16 ]
(%i12) columnspace (M);
                             [ 1  ]  [ 2  ]
                             [    ]  [    ]
                             [ 5  ]  [ 6  ]
(%o12)                 span([    ], [    ])
                             [ 9  ]  [ 10 ]
                             [    ]  [    ]
                             [ 13 ]  [ 14 ]
(%i13) apply ('orthogonal_complement, args (nullspace (transpose (M))));
                             [ 0 ]  [  1  ]
                             [   ]  [     ]
                             [ 1 ]  [  0  ]
(%o13)                 span([   ], [     ])
                             [ 2 ]  [ - 1 ]
                             [   ]  [     ]
```

```
[ 3 ]  [ - 2 ]
```

62.2 Funciones y variables para linearalgebra

`addmatrices` (*f*, *M_1*, ..., *M_n*) [Función]

Utiliza la función *f* como una función aditiva, devolviendo la suma de las matrices *M_1*, ..., *M_n*. La función *f* debe ser tal que acepte un número arbitrario de argumentos; en otras palabras, será una función n-aria de Maxima.

Ejemplos:

```
(%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*) [Función]

Devuelve el valor `true` si y solo si *M* es una matriz cuyos elementos son a su vez matrices.

`columnop` (*M*, *i*, *j*, *theta*) [Función]

Si *M* es una matriz, devuelve la matriz que resulta de hacer la operación columna `C_i <- C_i - theta * C_j`. Si *M* carece de cualquiera de las filas *i* o *j*, devuelve un mensaje de error.

`columnswap` (*M*, *i*, *j*) [Función]

Si *M* es una matriz, intercambia las columnas *i* y *j*. Si *M* carece de cualquiera de las filas *i* o *j*, devuelve un mensaje de error.

`columnspace` (*M*) [Función]

Si *M* es una matriz, devuelve `span (v_1, ..., v_n)`, donde el conjunto `{v_1, ..., v_n}` es la base del espacio generado por las columnas de *M*.

`copy` (*e*) [Función]

Devuelve una copia de la expresión *e* de Maxima. Aunque *e* puede ser cualquier expresión de Maxima, la función `copy` es especialmente útil cuando *e* es una lista o una matriz. Considérese el siguiente ejemplo:

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

Veamos el mismo ejemplo siendo ahora *mm* una copia de *m*

```
(%i6) m : [1,[2,3]]$
(%i7) mm : copy(m)$
```

```
(%i8) mm[2][1] : x$
(%i9) m;
(%o9)                        [1,[2,3]]
(%i10) mm;
(%o10)                       [1,[x,3]]
```

En esta ocasión, la asignación a *mm* no cambia el valor de *m*.

`cholesky` (*M*) [Función]
`cholesky` (*M*, *field*) [Función]

Devuelve la factorización de Cholesky de la matriz autoadjunta (o hermítica) *M*. El valor por defecto del segundo argumento es `generalring`. Para una descripción de los posibles valores para *field*, véase `lu_factor`.

`ctranspose` (*M*) [Función]

Devuelve la transpuesta compleja conjugada de la matriz *M*. La función `ctranspose` utiliza `matrix_element_transpose` para transponer cada elemento de la matriz.

`diag_matrix` (*d_1*, *d_2*, ..., *d_n*) [Función]

Devuelve una matriz diagonal con los elementos de la diagonal iguales a *d_1*, *d_2*, ..., *d_n*; cuando éstos son matrices, los elementos nulos de la matriz devuelta son matrices nulas de tamaño apropiado. Por ejemplo:

```
(%i1) diag_matrix(diag_matrix(1,2),diag_matrix(3,4));

                              [ [ 1  0 ]  [ 0  0 ] ]
                              [ [      ]  [      ] ]
                              [ [ 0  2 ]  [ 0  0 ] ]
(%o1)                         [                    ]
                              [ [ 0  0 ]  [ 3  0 ] ]
                              [ [      ]  [      ] ]
                              [ [ 0  0 ]  [ 0  4 ] ]
(%i2) diag_matrix(p,q);

                                     [ p  0 ]
(%o2)                                [      ]
                                     [ 0  q ]
```

`dotproduct` (*u*, *v*) [Función]

Devuelve el producto escalar de los vectores *u* y *v*. Equivale a `conjugate (transpose (u)) . v`. Los argumentos *u* y *v* deben ser vectores columna.

`eigens_by_jacobi` (*A*) [Función]
`eigens_by_jacobi` (*A*, *field_type*) [Función]

Calcula los valores y vectores propios de *A* por el método de las rotaciones de Jacobi. *A* debe ser una matriz simétrica (aunque no necesariamente definida o semidefinida positiva). El argumento *field_type* indica el tipo numérico sobre el que se realizan los cálculos, que puede ser tanto `floatfield` como `bigfloatfield`. En caso de que no se especifique *field_type*, su valor por defecto será `floatfield`.

Los elementos de A deben ser números o expresiones reducibles a números mediante la ejecución de `float` o `bfloat`, según sea el valor de *field_type*.

Ejemplos:

```
(%i1) S : matrix ([1/sqrt(2), 1/sqrt(2)], [- 1/sqrt(2), 1/sqrt(2)]);
                      [     1         1    ]
                      [  -------   ------- ]
                      [  sqrt(2)   sqrt(2) ]
(%o1)                 [                    ]
                      [      1        1    ]
                      [ - -------  ------- ]
                      [   sqrt(2)  sqrt(2) ]
(%i2) L : matrix ([sqrt(3), 0], [0, sqrt(5)]);
                       [ sqrt(3)     0    ]
(%o2)                  [                  ]
                       [    0     sqrt(5) ]
(%i3) M : S . L . transpose (S);
             [ sqrt(5)   sqrt(3)  sqrt(5)   sqrt(3) ]
             [ ------- + -------  ------- - ------- ]
             [    2         2        2         2    ]
(%o3)        [                                      ]
             [ sqrt(5)   sqrt(3)  sqrt(5)   sqrt(3) ]
             [ ------- - -------  ------- + ------- ]
             [    2         2        2         2    ]
(%i4) eigens_by_jacobi (M);
The largest percent change was 0.1454972243679
The largest percent change was 0.0
number of sweeps: 2
number of rotations: 1
(%o4) [[1.732050807568877, 2.23606797749979],
                         [  0.70710678118655   0.70710678118655 ]
                         [                                      ]]
                         [ - 0.70710678118655  0.70710678118655 ]
(%i5) float ([[sqrt(3), sqrt(5)], S]);
(%o5) [[1.732050807568877, 2.23606797749979],
                         [  0.70710678118655   0.70710678118655 ]
                         [                                      ]]
                         [ - 0.70710678118655  0.70710678118655 ]
(%i6) eigens_by_jacobi (M, bigfloatfield);
The largest percent change was 1.454972243679028b-1
The largest percent change was 0.0b0
number of sweeps: 2
number of rotations: 1
(%o6) [[1.732050807568877b0, 2.23606797749979b0],
                 [  7.071067811865475b-1   7.071067811865475b-1 ]
                 [                                              ]]
                 [ - 7.071067811865475b-1  7.071067811865475b-1 ]
```

`get_lu_factors` (*x*) [Función]

Cuando `x = lu_factor (A)`, entonces `get_lu_factors` devuelve una lista de la forma `[P, L, U]`, donde P es una matriz permutación, L es triangular inferior con unos en la diagonal y U es triangular superior, verificándose que `A = P L U`.

`hankel` (*col*) [Función]

`hankel` (*col*, *row*) [Función]

Devuelve la matriz de Hankel H. La primera columna de H coincide con *col*, excepto en el primer elemento, la última fila de H es *row*. El valor por defecto para *row* es el vector nulo con igual número de elementos que *col*.

`hessian` (*f*, *x*) [Función]

Devuelve la matriz hessiana de f con respecto de la lista de variables x. El elemento `(i, j)`-ésimo de la matriz hessiana es `diff(f, x[i], 1, x[j], 1)`.

Ejemplos:

```
(%i1) hessian (x * sin (y), [x, y]);
                       [   0       cos(y)   ]
(%o1)                  [                    ]
                       [ cos(y)  - x sin(y) ]
(%i2) depends (F, [a, b]);
(%o2)                         [F(a, b)]
(%i3) hessian (F, [a, b]);
                            [   2      2   ]
                            [  d F    d F  ]
                            [  ---   ----- ]
                            [    2   da db ]
                            [  da          ]
(%o3)                       [              ]
                            [   2      2   ]
                            [  d F    d F  ]
                            [ -----   ---  ]
                            [ da db     2  ]
                            [         db   ]
```

`hilbert_matrix` (*n*) [Función]

Devuelve la matriz de Hilbert n por n. Si n no es un entero positivo, emite un mensaje de error.

`identfor` (*M*) [Función]

`identfor` (*M*, *fld*) [Función]

Devuelve una matriz identidad con la misma forma que la matriz M. Los elementos de la diagonal de la matriz identidad son la identidad multiplicativa del campo *fld*; el valor por defecto para *fld* es *generalring*.

El primer argumento M debe ser una matriz cuadrada o no ser matriz en absoluto. Si M es una matriz, sus elementos pueden ser matrices cuadradas. La matriz puede tener bloques a cualquier nivel finito de profundidad.

Véase también `zerofor`

`invert_by_lu` (*M*, (*rng generalring*)) [Función]
Invierte la matriz *M* mediante la factorización LU, la cual se hace utilizando el anillo *rng*.

`jacobian` (*f*, *x*) [Función]
Devuelve la matriz jacobiana de la lista de funciones *f* respecto de la lista de variables *x*. El elemento `(i, j)`-ésimo de la matriz jacobiana es `diff(f[i], x[j])`.

Ejemplos:

```
(%i1) jacobian ([sin (u - v), sin (u * v)], [u, v]);
                  [ cos(v - u)  - cos(v - u) ]
(%o1)             [                          ]
                  [ v cos(u v)   u cos(u v)  ]
(%i2) depends ([F, G], [y, z]);
(%o2)                  [F(y, z), G(y, z)]
(%i3) jacobian ([F, G], [y, z]);
                             [ dF  dF ]
                             [ --  -- ]
                             [ dy  dz ]
(%o3)                        [        ]
                             [ dG  dG ]
                             [ --  -- ]
                             [ dy  dz ]
```

`kronecker_product` (*A*, *B*) [Función]
Devuelve el producto de Kroneckerde las matrices *A* y *B*.

`listp` (*e*, *p*) [Función]
`listp` (*e*) [Función]
Dado el argumento opcional *p*, devuelve `true` si *e* es una lista de Maxima y *p* toma el valor `true` al aplicarlo a cada elemento de la lista. Si a `listp` no se le suministra el argumento opcional, devuelve `true` si *e* es una lista de Maxima. En cualquier otro caso, el resultado es `false`.

`locate_matrix_entry` (*M*, *r_1*, *c_1*, *r_2*, *c_2*, *f*, *rel*) [Función]
El primer argumento debe ser una matriz, mientras que los argumentos desde *r_1* hasta *c_2* determinan la submatriz de *M* tomando las filas desde *r_1* hasta *r_2* y las columnas desde *c_1* hasta *c_2*.

La función `locate_matrix_entry` busca en la submatriz de *M* un elemento que satisfaga cierta propiedad. hay tres posibilidades:

(1) *rel* = `'bool` y *f* es un predicado:

Rastrea la submatriz de izquierda a derecha y de arriba hacia abajo, devolviendo el índice del primer elemento que satisface el predicado *f*; si ningún elemento lo satisface, el resultado es `false`.

(2) *rel* = `'max` y *f* una función real:

Rastrea la submatriz buscando el elemento que maximice *f*, devolviendo el índice correspondiente.

(3) `rel = 'min` y f una función real:

Rastrea la submatriz buscando el elemento que minimice f, devolviendo el índice correspondiente.

`lu_backsub` (*M*, *b*) [Función]

Si *M* = `lu_factor (A, field)`, entonces `lu_backsub (M, b)` resuelve el sistema de ecuaciones lineales *A x = b*.

`lu_factor` (*M*, *field*) [Función]

Devuelve una lista de la forma `[LU, perm, fld]`, o `[LU, perm, fld, lower-cnd upper-cnd]`, donde

- La matriz LU contiene la factorización de M de forma empaquetada, lo que significa tres cosas. En primer lugar, que las filas de LU están permutadas de acuerdo con la lista *perm*; por ejemplo, si *perm* es la lista `[3,2,1]`, la primera fila de la factorización LU es la tercera fila de la matriz LU. En segundo lugar, el factor triangular inferior de M es la parte triangular inferior de LU con los elementos de la diagonal sustituidos por unos. Por último, el factor triangular superior de M es la parte triangular superior de LU.
- Si el campo es `floatfield` o `complexfield`, los números *lower-cnd* y *upper-cnd* son las cotas inferior y superior del número de condición de la norma infinita de M. El número de condición no se puede estimar para todos los campos, en cuyo caso `lu_factor` devuelve una lista de dos elementos. Tanto la cota inferior como la superior pueden diferir de sus valores verdaderos. Véase también `mat_cond`.

El argumento M debe ser una matriz cuadrada.

El argumento opcional *fld* debe ser un símbolo que determine un anillo o un campo. Los anillos y campos predefinidos son:

a. `generalring` – el anillo de las expresiones de Maxima
b. `floatfield` – el campo de los números decimales en coma flotante de doble precisión
c. `complexfield` – el campo de los números complejos decimales en coma flotante de doble precisión
d. `crering` – el anillo de las expresiones canónicas racionales (*Canonical Rational Expression* o CRE) de Maxima
e. `rationalfield` – el campo de los números racionales
f. `runningerror` – controla los errores de redondeo de las operaciones en coma flotante
g. `noncommutingring` – el anillo de las expresiones de Maxima en las que el producto es el operador no conmutativo `"."`

Si el campo es `floatfield`, `complexfield` o `runningerror`, el algoritmo utiliza pivoteo parcial; para los demás campos, las filas se cambian cuando se necesita evitar pivotes nulos.

La suma aritmética en coma flotante no es asociativa, por lo que el significado de 'campo' no coincide exactamente con su definición matemática.

Un elemento del campo `runningerror` consiste en una lista de Maxima de la forma `[x,n]`, donde *x* es un número decimal en coma flotante y `n` un enetro. La diferencia relativa entre el valor real de `x` y `x` está aproximadamente acotado por el valor epsilon de la máquina multiplicado por `n`.

No es posible la definición de un nuevo campo por parte del usuario, a menos que éste tenga conocimientos de Common Lisp. Para hacerlo, el usuario debe definir funciones para las operaciones aritméticas y para convertir de la representación del campo a Maxima y al revés. Además, en los campos ordenados, donde se hace uso del pivoteo parcial, el usuario debe definir funciones para el módulo y para comparar números del campo. Después de lo anterior, tan solo queda definir una estructura Common Lisp `mring`. El fichero `mring` tiene muchos ejemplos.

Para calcular la factorización, la primera tarea consiste en convertir cada elemento de la matriz a un elemento del campo especificado. Si la conversión no es posible, la factorización se detiene con un mensaje de error. Los elementos del campo no necesitan ser expresiones de Maxima; por ejemplo, los elementos de `complexfield` son números complejos de Common Lisp. Tras la factorización, los elementos de la matriz deben convertirse nuevamente a expresiones de Maxima.

Véase también `get_lu_factors`.

Ejemplos:

```
(%i1) w[i,j] := random (1.0) + %i * random (1.0);
(%o1)          w     := random(1.) + %i random(1.)
                i, j
(%i2) showtime : true$
Evaluation took 0.00 seconds (0.00 elapsed)
(%i3) M : genmatrix (w, 100, 100)$
Evaluation took 7.40 seconds (8.23 elapsed)
(%i4) lu_factor (M, complexfield)$
Evaluation took 28.71 seconds (35.00 elapsed)
(%i5) lu_factor (M, generalring)$
Evaluation took 109.24 seconds (152.10 elapsed)
(%i6) showtime : false$

(%i7) M : matrix ([1 - z, 3], [3, 8 - z]);
                        [ 1 - z    3   ]
(%o7)                   [              ]
                        [   3    8 - z ]
(%i8) lu_factor (M, generalring);
          [ 1 - z         3        ]
          [                        ]
(%o8)    [[   3            9       ], [1, 2], generalring]
          [ -----  - z - ----- + 8 ]
          [ 1 - z        1 - z     ]
(%i9) get_lu_factors (%);
                  [   1    0 ]  [ 1 - z         3        ]
       [ 1  0 ]   [          ]  [                        ]
(%o9)  [[     ], [   3     ], [               9       ]]
```

```
                [ 0  1 ]  [ -----  1 ]  [   0    - z - ----- + 8 ]
                          [ 1 - z    ]  [              1 - z     ]
(%i10) %[1] . %[2] . %[3];
                                [ 1 - z    3   ]
(%o10)                          [              ]
                                [   3    8 - z ]
```

`mat_cond` (*M*, *1*) [Función]
`mat_cond` (*M*, *inf*) [Función]

Devuelve el número de condición de la *p*-norma de la matriz *M*. Los valores admisibles para *p* son 1 y *inf*. Esta función utiliza la factorización LU para invertir la matriz *M*, por lo que el tiempo de ejecución de `mat_cond` es proporcional al cubo del tamaño de la matriz; `lu_factor` determina las cotas inferior y superior para el número de condición de la norma infinita en un tiempo proporcional al cuadrado del tamaño de la matriz.

`mat_norm` (*M*, *1*) [Función]
`mat_norm` (*M*, *inf*) [Función]
`mat_norm` (*M*, *frobenius*) [Función]

Devuelve la *p*-norma de la matriz *M*. Los valores admisibles para *p* son 1, `inf` y `frobenius` (la norma matricial de Frobenius). La matriz *M* no debe contener bloques.

`matrixp` (*e*, *p*) [Función]
`matrixp` (*e*) [Función]

Dado el argumento opcional *p*, devuelve `true` si *e* es una matriz y *p* toma el valor `true` al aplicarlo a cada elemento de la matriz. Si a `matrixp` no se le suministra el argumento opcional, devuelve `true` si *e* es una matriz. En cualquier otro caso, el resultado es `false`.

Véase también `blockmatrixp`

`matrix_size` (*M*) [Función]

Devuelve una lista con el número de filas y columnas de la matriz *M*.

`mat_fullunblocker` (*M*) [Función]

Si *M* es una matriz de bloques, transforma la matriz llevando todos los elementos de los bloques al primer nivel. Si *M* es una matriz, devuelve *M*; en cualquier otro caso, envía un mensaje de error.

`mat_trace` (*M*) [Función]

Calcula la traza de la matriz *M*. Si *M* no es una matriz, devuelve una forma nominal. Si *M* es una matriz de bloques, `mat_trace(M)` calcula el mismo valor que `mat_trace(mat_unblocker(m))`.

`mat_unblocker` (*M*) [Función]

Si *M* es una matriz de bloques, deshace los bloques de un nivel. Si *M* es una matriz, `mat_unblocker (M)` devuelve *M*; en cualquier otro caso, envía un mensaje de error.

Si todos los elementos de *M* son matrices, `mat_unblocker (M)` devuelve una matriz sin bloques, pero si los elementos de *M* son a su vez matrices de bloques, `mat_unblocker (M)` devuelve una matriz con el nivel de bloques disminuido en uno.

En caso de trabajar con matrices de bloques, quizás sea conveniente darle a `matrix_element_mult` el valor `"."` y a `matrix_element_transpose` el valor `'transpose`. Véase también `mat_fullunblocker`.

Ejemplo:

```
(%i1) A : matrix ([1, 2], [3, 4]);
                              [ 1  2 ]
(%o1)                         [      ]
                              [ 3  4 ]
(%i2) B : matrix ([7, 8], [9, 10]);
                              [ 7  8  ]
(%o2)                         [       ]
                              [ 9  10 ]
(%i3) matrix ([A, B]);
                        [ [ 1  2 ]  [ 7  8  ] ]
(%o3)                   [ [      ]  [       ] ]
                        [ [ 3  4 ]  [ 9  10 ] ]
(%i4) mat_unblocker (%);
                            [ 1  2  7  8  ]
(%o4)                       [             ]
                            [ 3  4  9  10 ]
```

`nullspace` (*M*) [Función]

Si *M* es una matriz, devuelve `span (v_1, ..., v_n)`, siendo `{v_1, ..., v_n}` la base del espacio nulo de *M*. Si el espacio nulo contiene un único elemento, devuelve `span ()`.

`nullity` (*M*) [Función]

Si *M* es una matriz, devuelve la dimensión del espacio nulo de *M*.

`orthogonal_complement` (*v_1*, ..., *v_n*) [Función]

Devuelve `span (u_1, ..., u_m)`, siendo `{u_1, ..., u_m}` la base del complemento ortogonal del conjunto `(v_1, ..., v_n)`, cuyos elementos deben ser vectores columna.

`polynomialp` (*p*, *L*, *coeffp*, *exponp*) [Función]
`polynomialp` (*p*, *L*, *coeffp*) [Función]
`polynomialp` (*p*, *L*) [Función]

Devuelve `true` si *p* es un polinomio cuyas variables son las de la lista *L*, el predicado *coeffp* toma el valor `true` al aplicarlo a cada coeficiente y el predicado *exponp* también alcanza el valor `true` al ser aplicado a los exponentes de las variables listadas en *L*. En caso de necesitar que *exponp* no sea un predicado por defecto, se deberá especificar también el predicado *coeffp*, aunque aquí se desee su comportamiento por defecto.

La instrucción `polynomialp (p, L, coeffp)` equivale a `polynomialp (p, L, coeffp, 'nonnegintegerp)`, al tiempo que `polynomialp (p, L)` equivale a `polynomialp (p, L, 'constantp, 'nonnegintegerp)`.

No es necesario expandir el polinomio:

```
(%i1) polynomialp ((x + 1)*(x + 2), [x]);
(%o1)                         true
```

```
(%i2) polynomialp ((x + 1)*(x + 2)^a, [x]);
(%o2)                          false
```

Un ejemplo utilizando valores distintos a los utilizados por defecto en *coeffp* y en *exponp*:

```
(%i1) polynomialp ((x + 1)*(x + 2)^(3/2), [x],
                   numberp, numberp);
(%o1)                          true
(%i2) polynomialp ((x^(1/2) + 1)*(x + 2)^(3/2), [x],
                   numberp, numberp);
(%o2)                          true
```

Polinomios con dos variables:

```
(%i1) polynomialp (x^2 + 5*x*y + y^2, [x]);
(%o1)                          false
(%i2) polynomialp (x^2 + 5*x*y + y^2, [x, y]);
(%o2)                          true
```

`polytocompanion` (*p*, *x*) [Función]

Si *p* es un polinomio en *x*, devuelve la matriz compañera de *p*. Para un polinomio mónico *p* de grado *n* se tiene `p = (-1)^n charpoly (polytocompanion (p, x))`.

Si *p* no es un polinomio en *x*, se devuelve un mensaje de error.

`ptriangularize` (*M*, *v*) [Función]

Si *M* es una matriz en la que sus elementos son polinomios en *v*, devuelve una matriz *M2* tal que

1. *M2* es triangular superior,
2. `M2 = E_n ... E_1 M`, donde *E_1*, ..., *E_n* son matrices elementales cuyos elementos son polinomios en *v*,
3. `|det (M)| = |det (M2)|`,

Nota: esta función no comprueba si los elementos de la matriz son polinomios en *v*.

`rowop` (*M*, *i*, *j*, *theta*) [Función]

Si *M* es una matriz, devuelve la matriz que resulta de relizar la transformación `R_i <- R_i - theta * R_j` con las filas `R_i` y `R_j`. Si *M* no tiene estas filas, devuelve un mensaje de error.

`rank` (*M*) [Función]

Calcula el rango de la matriz *M*. El rango es la dimensión del espacio columna. Ejemplo:

```
(%i1) rank(matrix([1,2],[2,4]));
(%o1)                                1
(%i2) rank(matrix([1,b],[c,d]));
Proviso:  {d - b c # 0}
(%o2)                                2
```

`rowswap` (*M*, *i*, *j*) [Función]

Si *M* es una matriz, intercambia las filas *i* y *j*. Si *M* carece de estas filas, devuelve un mensaje de error.

`toeplitz` (*col*) [Función]
`toeplitz` (*col*, *row*) [Función]

Devuelve una matriz de Toeplitz T. La primera columna de T es *col*, excepto su primer elemento. La primera fila de T es *row*. El valor por defecto para *row* es el complejo conjugado de *col*. Ejemplo:

```
(%i1)  toeplitz([1,2,3],[x,y,z]);
                                    [ 1  y  z ]
                                    [         ]
(%o1)                               [ 2  1  y ]
                                    [         ]
                                    [ 3  2  1 ]
(%i2)  toeplitz([1,1+%i]);

                                  [   1     1 - %I ]
(%o2)                             [                ]
                                  [ %I + 1    1    ]
```

`vandermonde_matrix` ([*x_1*, ..., *x_n*]) [Función]

Devuelve una matriz n por n, cuya i-ésima fila es `[1, x_i, x_i^2, ... x_i^(n-1)]`.

`zerofor` (*M*) [Función]
`zerofor` (*M*, *fld*) [Función]

Devuelve la matriz nula con la misma estructura que la matriz M. Cada elemento de la matriz nula es la identidad aditiva del campo *fld*; el valor por defecto de *fld* es *generalring*.

El primer argumento de M debe ser una matriz cuadrada o no ser matriz en absoluto. Si M es una matriz, cada uno de sus elementos puede ser una matriz cuadrada, por lo que M puede ser una matriz de Maxima definida por bloques.

Véase también `identfor`.

`zeromatrixp` (*M*) [Función]

Si M no es una matriz definida por bloques, devuelve `true` si `is (equal (e, 0))` es verdadero para todo elemento e de M. Si M es una matriz por bloques, devuelve `true` si `zeromatrixp` devuelve a su vez `true` para cada elemento de e.

63 lsquares

63.1 Funciones y variables para lsquares

`lsquares_estimates` (*D*, *x*, *e*, *a*) [Función]
`lsquares_estimates` (*D*, *x*, *e*, *a*, *initial* = *L*, *tol* = *t*) [Función]

Estima los parámetros *a* que mejor se ajusten a la ecuación *e* de variables *x* y *a* a los datos *D* por el método de los mínimos cuadrados. La función `lsquares_estimates` busca primero una solución exacta, y si no la encuentra, buscará una aproximada.

El resultado es una lista de listas de ecuaciones de la forma `[a = ..., b = ..., c = ...]`. Cada elemento de la lista es un mínimo diferente de error cuadrático medio.

Los datos deben darse en formato matricial. Cada fila es un dato (el cual suele denominarse 'registro' o 'caso' en ciertos contextos), y las columnas contienen los valores para cada una de las variables. La lista de variables *x* asigna un nombre a cada una de las columnas de *D*, incluso a aquellas que no intervienen en el análisis. La lista *a* asigna nombres a los parámetros cuyas estimaciones se buscan. El argumento *e* es una expresión o ecuación de variables *x* y *a*; si *e* no es una ecuación (es decir, carece de igualdad), se trata como si fuese `e = 0`.

Se pueden dar argumentos adicionales a `lsquares_estimates` en forma de ecuaciones, las cuales se pasan tal cual a la función `lbfgs`, que es la que se encarga de calcular las estimaciones por el método numérico cuando no encuentra una solución exacta.

Cuando se pueda encontrar una solución exacta, mediante `solve`, los datos en *D* pueden contener valores no numéricos. Sin embargo, cuando no exista solución exacta, todos los elementos de *D* deben ser necesariamente numéricos, lo cual incluye constantes numéricas tales como `%pi` o `%e` y números literales (enteros, racionales y decimales en coma flotante, tanto los de doble precisión como los de precisión arbitraria). Los cálculos numéricos se realizan en doble precisión con aritmética de punto flotante, por lo que números de cualesquiera otro tipo son convenientemente convertidos antes de proceder con los cálculos.

Antes de utilizar esta función ejecútese `load("lsquares")`.

Véanse también `lsquares_estimates_exact`, `lsquares_estimates_approximate`, `lsquares_mse`, `lsquares_residuals` y `lsquares_residual_mse`.

Ejemplos:

Un problema con solución exacta.

```
(%i1) load ("lsquares")$
(%i2) M : matrix (
        [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                                  [ 1  1  1 ]
                                  [         ]
                                  [ 3       ]
                                  [ -  1  2 ]
                                  [ 2       ]
                                  [         ]
(%o2)                             [ 9       ]
```

```
                                      [ -  2  1 ]
                                      [ 4       ]
                                      [         ]
                                      [ 3  2  2 ]
                                      [         ]
                                      [ 2  2  1 ]
(%i3) lsquares_estimates (
        M, [z,x,y], (z+D)^2 = A*x+B*y+C, [A,B,C,D]);
                          59        27      10921        107
(%o3)           [[A = - --, B = - --, C = -----, D = - ---]]
                          16        16      1024         32
```

un problema para el que no se encuentra solución exacta, por lo que `lsquares_estimates` recurre a la aproximación numérica.

```
(%i1) load ("lsquares")$
(%i2) M : matrix ([1, 1], [2, 7/4], [3, 11/4], [4, 13/4]);
                                      [ 1  1  ]
                                      [       ]
                                      [    7  ]
                                      [ 2  -  ]
                                      [    4  ]
                                      [       ]
(%o2)                                 [    11 ]
                                      [ 3  -- ]
                                      [    4  ]
                                      [       ]
                                      [    13 ]
                                      [ 4  -- ]
                                      [    4  ]
(%i3) lsquares_estimates (
  M, [x,y], y=a*x^b+c, [a,b,c], initial=[3,3,3], iprint=[-1,0]);
(%o3) [[a = 1.387365874920637, b = .7110956639593767,
                                           c = - .4142705622439105]]
```

`lsquares_estimates_exact` (*MSE*, `a`) [Función]

Estima los valores de los parámetros `a` que minimizan el error cuadrático medio *MSE* mediante un sistema de ecuaciones que intentará resolver simbólicamente con `solve`. El error cuadrático medio es una expresión con parámetros `a`, como los devueltos por `lsquares_mse`.

El valor devuelto por la función es una lista de listas de ecuaciones de la forma `[a = ..., b = ..., c = ...]`. El resultado puede contener cero, uno o más elementos. Cuando la respuesta contiene más de una solución, todas ellas representan mínimos del error cuadrático medio.

Véanse también `lsquares_estimates`, `lsquares_estimates_approximate`, `lsquares_mse`, `lsquares_residuals` y `lsquares_residual_mse`.

Ejemplo:

```
(%i1) load ("lsquares")$
```

```
(%i2) M : matrix (
         [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                              [ 1  1  1 ]
                              [         ]
                              [ 3       ]
                              [ -  1  2 ]
                              [ 2       ]
                              [         ]
(%o2)                         [ 9       ]
                              [ -  2  1 ]
                              [ 4       ]
                              [         ]
                              [ 3  2  2 ]
                              [         ]
                              [ 2  2  1 ]
(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);
               5
              ====
              \                  2                          2
               >    ((D + M    )  - C - M     B - M     A)
              /              i, 1           i, 3      i, 2
              ====
              i = 1
(%o3)         ------------------------------------------
                                  5
(%i4) lsquares_estimates_exact (mse, [A, B, C, D]);
                     59        27      10921        107
(%o4)         [[A = - --, B = - --, C = -----, D = - ---]]
                     16        16      1024         32
```

`lsquares_estimates_approximate` (*MSE*, `a`, *initial* = *L*, *tol* = `t`) [Función]

Estima los valores de los parámetros *a* que minimizan el error cuadrático medio *MSE* mediante el algoritmo numérico `lbfgs`. El error cuadrático medio es una expresión con parámetros *a*, como los devueltos por `lsquares_mse`.

La solución devuelta por la función es un mínimo local (posiblemente global) del error cuadrático medio.

Por consistencia con `lsquares_estimates_exact`, el valor devuelto es una lista anidada con un único elemento, consistente en una lista de ecuaciones de la forma `[a = ..., b = ..., c = ...]`.

Los argumentos adicionales de `lsquares_estimates_approximate` se especifican como ecuaciones y se pasan de esta forma a la función `lbfgs`.

MSE debe devolver un número cuando a sus parámetros se les asignen valores numéricos, lo cual implica que los datos a partir de los cuales se ha generado *MSE* contengan únicamente constantes numéricas tales como `%pi` o `%e` y números literales (enteros, racionales y decimales en coma flotante, tanto los de doble precisión como los de precisión arbitraria). Los cálculos numéricos se realizan en doble precisión

con aritmética de punto flotante, por lo que números de cualesquiera otro tipo son convenientemente convertidos antes de proceder con los cálculos.

Antes de utilizar esta función ejecútese `load("lsquares")`.

Véanse también `lsquares_estimates`, `lsquares_estimates_exact`, `lsquares_mse`, `lsquares_residuals` y `lsquares_residual_mse`.

Ejemplo:

```
(%i1) load ("lsquares")$
(%i2) M : matrix (
         [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                              [ 1  1  1 ]
                              [         ]
                              [ 3       ]
                              [ -  1  2 ]
                              [ 2       ]
                              [         ]
(%o2)                         [ 9       ]
                              [ -  2  1 ]
                              [ 4       ]
                              [         ]
                              [ 3  2  2 ]
                              [         ]
                              [ 2  2  1 ]
(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);
                5
               ====
               \                 2                          2
                >    ((D + M    )  - C - M     B - M     A)
               /             i, 1           i, 3      i, 2
               ====
               i = 1
(%o3)          ------------------------------------------
                                  5
(%i4) lsquares_estimates_approximate (
                    mse, [A, B, C, D], iprint = [-1, 0]);
(%o4) [[A = - 3.67850494740174, B = - 1.683070351177813,
                   C = 10.63469950148635, D = - 3.340357993175206]]
```

`lsquares_mse` (*D*, *x*, *e*) [Función]

Devuelve el error medio cuadrático (MSE) para la ecuación *e* de variables *x* respecto de los datos *D*. El resultado devuelto es una suma, definida como

$$\frac{1}{n}\sum_{i=1}^{n}\left[\mathrm{lhs}\left(e_i\right)-\mathrm{rhs}\left(e_i\right)\right]^2,$$

siendo *n* el número de datos y `e[i]` es la ecuación *e* evaluada cuando a sus variables *x* se le asignan los valores asociados al dato `i`-ésimo *D*`[i]`.

Antes de utilizar esta función ejecútese `load("lsquares")`.

Ejemplo:

```
(%i1) load ("lsquares")$
(%i2) M : matrix (
         [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                              [ 1  1  1 ]
                              [         ]
                              [ 3       ]
                              [ -  1  2 ]
                              [ 2       ]
                              [         ]
(%o2)                         [ 9       ]
                              [ -  2  1 ]
                              [ 4       ]
                              [         ]
                              [ 3  2  2 ]
                              [         ]
                              [ 2  2  1 ]
(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);
           5
          ====
          \                 2                          2
           >    ((D + M    )  - C - M     B - M     A)
          /             i, 1           i, 3      i, 2
          ====
          i = 1
(%o3)     ------------------------------------------
                                5
(%i4) diff (mse, D);
          5
         ====
         \                              2
       4  >    (D + M    ) ((D + M    )  - C - M     B - M     A)
         /           i, 1          i, 1           i, 3      i, 2
         ====
         i = 1
(%o4) ------------------------------------------------------------
                                  5
(%i5) ''mse, nouns;
                2                 2          9 2                2
(%o5) (((D + 3)  - C - 2 B - 2 A)  + ((D + -)  - C - B - 2 A)
                                                4
            2                2          3 2                2
 + ((D + 2)  - C - B - 2 A)  + ((D + -)  - C - 2 B - A)
                                          2
            2              2
```

```
          + ((D + 1)  - C - B - A) )/5
```

`lsquares_residuals` (*D*, *x*, *e*, *a*) [Función]

Devuelve los residuos para la ecuación *e* de parámetros *a* y datos *D*.

D es una matriz, *x* una lista de variables y *e* es una ecuación o expresión general; si *e* no es una ecuación (es decir, carece de igualdad), se trata como si fuese `e = 0`. La lista *a* contiene ecuaciones que especifican valores para cualesquiera parámetros de *e* que no estén en *x*.

Los residuos se definen como

$$\mathrm{lhs}\,(e_i) - \mathrm{rhs}\,(e_i)\,,$$

siendo `e[i]` la ecuación *e* evaluada cuando las variables *x* toman los valores asociados al dato `i`-ésimo `D[i]`, y haciendo las asignaciones indicadas en *a* al resto de variables.

Antes de utilizar esta función ejecútese `load("lsquares")`.

Ejemplo:

```
(%i1) load ("lsquares")$
(%i2) M : matrix (
         [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                                    [ 1  1  1 ]
                                    [         ]
                                    [ 3       ]
                                    [ -  1  2 ]
                                    [ 2       ]
                                    [         ]
(%o2)                               [ 9       ]
                                    [ -  2  1 ]
                                    [ 4       ]
                                    [         ]
                                    [ 3  2  2 ]
                                    [         ]
                                    [ 2  2  1 ]
(%i3) a : lsquares_estimates (
          M, [z,x,y], (z+D)^2 = A*x+B*y+C, [A,B,C,D]);
                          59        27      10921        107
(%o3)           [[A = - --, B = - --, C = -----, D = - ---]]
                          16        16      1024         32
(%i4) lsquares_residuals (
          M, [z,x,y], (z+D)^2 = A*x+B*y+C, first(a));
                           13    13    13  13  13
(%o4)                     [--, - --, - --, --, --]
                           64    64    32  64  64
```

`lsquares_residual_mse` (*D*, *x*, *e*, *a*) [Función]

Devuelve el residuo del error cuadrático medio (MSE) de la ecuación *e* para los valores parámetricos *a* y datos *D*.

El residuo del error cuadrático medio (MSE) se define como

$$\frac{1}{n}\sum_{i=1}^{n}\left[\mathrm{lhs}\left(e_i\right)-\mathrm{rhs}\left(e_i\right)\right]^2,$$

siendo `e[i]` la ecuación e evaluada cuando las variables x toman los valores asociados al dato `i`-ésimo `D[i]`, y haciendo las asignaciones indicadas en a al resto de variables.

Antes de utilizar esta función ejecútese `load("lsquares")`.

Ejemplo:

```
(%i1) load ("lsquares")$
(%i2) M : matrix (
         [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                              [ 1  1  1 ]
                              [         ]
                              [ 3       ]
                              [ -  1  2 ]
                              [ 2       ]
                              [         ]
(%o2)                         [ 9       ]
                              [ -  2  1 ]
                              [ 4       ]
                              [         ]
                              [ 3  2  2 ]
                              [         ]
                              [ 2  2  1 ]
(%i3) a : lsquares_estimates (
            M, [z,x,y], (z+D)^2 = A*x+B*y+C, [A,B,C,D]);

                    59        27      10921        107
(%o3)     [[A = - --, B = - --, C = -----, D = - ---]]
                    16        16      1024         32
(%i4) lsquares_residual_mse (
            M, [z,x,y], (z + D)^2 = A*x + B*y + C, first (a));
                               169
(%o4)                          ----
                               2560
```

`plsquares` (*Mat*,*VarList*,*depvars*) [Función]
`plsquares` (*Mat*,*VarList*,*depvars*,*maxexpon*) [Función]
`plsquares` (*Mat*,*VarList*,*depvars*,*maxexpon*,*maxdegree*) [Función]

Ajuste de una función polinómica multivariante a una tabla de datos por el método de los *mínimos cuadrados*. *Mat* es la matriz con los datos empíricos, *VarList* es la lista con los nombres de las variables (una por cada columna de *Mat*, pero puede usarse `-` en lugar de los nombres de variables para ignorar las columnas de *Mat*), *depvars* es el nombre de la variable dependiente o una lista con uno o más nombres de variables dependientes (cuyos nombres deben estar también en *VarList*), *maxexpon* es un argumento opcional para indicar el máximo exponente para cada una de las

variables independientes (1 por defecto) y *maxdegree* es otro argumento opcional para el grado del polinomio (*maxexpon* por defecto); nótese que la suma de exponentes de cada término debe ser igual o menor que *maxdegree*. Si `maxdgree = 0` entonces no se aplicará ningún límite.

Si *depvars* es el nombre de una variable dependiente (no en una lista), `plsquares` devuelve el polinomio ajustado. Si *depvars* es una lista de una o más variables dependientes, `plsquares` devuelve una lista con los polinomios ajustados. Los coeficientes de determinación se muestran en su orden correspondiente para informar sobre la bondad del ajuste. Estos valores se almacenan también en la variable global *DETCOEF* en un formato de lista si *depvars* es a su vez una lista.

Un ejemplo sencillo de ajuste lineal multivariante:

```
(%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
```

El mismo ejemplo sin restricciones en el grado:

```
(%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
```

Cálculo del número de diagonales de un polígono de N lados

```
(%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
```

Cálculo del número de formas de colocar dos reinas en un tablero n x n de manera que no se amenacen.

```
(%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
```

Un ejemplo con seis variables dependientes:

```
(%i8) mtrx:matrix([0,0,0,0,0,1,1,1],[0,1,0,1,1,1,1,0,0],
                  [1,0,0,1,1,1,0,0],[1,1,1,1,1,0,0,0,1])$
(%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]
```

Antes de hacer uso de esta función ejecútese `load("plsquares")`.

64 makeOrders

64.1 Funciones y variables para makeOrders

`makeOrders` (*indvarlist*,*orderlist*) [Función]

Devuelve una lista con las potencias de las variables de un polinomio término a término.

```
(%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
```

donde `[0, 1]` se asocia al término b y `[2, 3]` a a^2b^3.

Antes de hacer uso de esta función ejecútese `load("makeOrders")`.

65 minpack

65.1 Introducción a minpack

`Minpack` es una traducción a Common Lisp (via `f2cl`) de la librería MINPACK escrita en Fortran, tal como se puede obtener de Netlib.

65.2 Funciones y variables para minpack

`minpack_lsquares` [Función]

`minpack_lsquares` (*flist*, *varlist*, *guess*)
`minpack_lsquares` (..., *'tolerance* = *tolerance*)
`minpack_lsquares` (..., *'jacobian* = *jacobian*)

Calcula el punto `x` que minimiza la suma de los cuadrados de las funciones de la lista *flist*. Las variables se escriben en la lista `varlist`. El argumento *guess* debe guardar una estimación inicial del punto óptimo.

Los argumentos opcionales *tolerance* y *jacobian* permiten mantener cierto control sobre el algoritmo; *tolerance* es el error relativo estimado que se desea en la suma de cuadrados, mientras que *jacobian* puede utilizarse para especificar el jacobiano. Si *jacobian* no se suministra, o se le da el valor `true`, el que ya tiene por defecto, el jacobiano se calcula a partir de *flist*. Si *jacobian* vale `false`, se utilizará una aproximación numérica.

`minpack_lsquares` devuelve una lista, siendo su primer elemento la solución estimada, el segundo la suma de cuadrados y el tercero indica la bondad del algoritmo, siendo sus posibles valores los siguientes:

`0` Número incorrecto de parámetros.

`1` El algoritmo estima que el error relativo de la suma de cuadrados es, como mucho, igual a `tolerance`.

`2` El algoritmo estima que el error relativo entre x y la solución es, como mucho, igual a `tolerance`.

`3` Las dos condiciones anteriores se cumplen.

`4` El vector formado por las funciones evaluadas en el punto x es ortogonal a las columnas del jacobiano dentro de la precisión de la máquina.

`5` El número de llamadas a las funciones ha alcanzado 100*(n+1), siendo n el número de variables.

`6` La tolerancia es demasiado pequeña, no siendo posible reducir más la suma de cuadrados.

`7` La tolerancia es demasiado pequeña, no siendo posible mejorar la solución aproximada x.

```
/* Problem 6: Powell singular function */
(%i1) powell(x1,x2,x3,x4) :=
         [x1+10*x2, sqrt(5)*(x3-x4), (x2-2*x3)^2,
```

```
                    sqrt(10)*(x1-x4)^2]$
(%i2) minpack_lsquares(powell(x1,x2,x3,x4), [x1,x2,x3,x4],
                       [3,-1,0,1]);
(%o2) [[1.652117596168394e-17, - 1.652117596168393e-18,
        2.643388153869468e-18, 2.643388153869468e-18],
       6.109327859207777e-34, 4]
/* Same problem but use numerical approximation to Jacobian */
(%i3) minpack_lsquares(powell(x1,x2,x3,x4), [x1,x2,x3,x4],
                       [3,-1,0,1], jacobian = false);
(%o3) [[5.060282149485331e-11, - 5.060282149491206e-12,
        2.179447843547218e-11, 2.179447843547218e-11],
       3.534491794847031e-21, 5]
```

`minpack_solve` [Función]

`minpack_solve` (*flist*, *varlist*, *guess*)
`minpack_solve` (..., *'tolerance* = *tolerance*)
`minpack_solve` (..., *'jacobian* = *jacobian*)

Resuelve un sistema de `n` ecuaciones con `n` incógnitas. Las `n` ecuaciones forman la lista *flist*, estando la lista *varlist* formada por las incógnitas. El argumento *guess* es una estimación inicial de la solución.

Los argumentos opcionales *tolerance* y *jacobian* permiten mantener cierto control sobre el algoritmo; *tolerance* es el error relativo estimado que se desea en la suma de cuadrados, mientras que *jacobian* puede utilizarse para especificar el jacobiano. Si *jacobian* no se suministra, o se le da el valor `true`, el que ya tiene por defecto, el jacobiano se calcula a partir de *flist*. Si *jacobian* vale `false`, se utilizará una aproximación numérica.

`minpack_solve` devuelve una lista, siendo su primer elemento la solución estimada, el segundo la suma de cuadrados y el tercero indica la bondad del algoritmo, siendo sus posibles valores los siguientes:

`0` Número incorrecto de parámetros.

`1` El algoritmo estima que el error relativo de la suma de cuadrados es, como mucho, igual a `tolerance`.

`2` El número de llamadas a las funciones ha alcanzado 100*(n+1), siendo n el número de incógnitas.

`3` La tolerancia es demasiado pequeña, no siendo posible reducir más la suma de cuadrados.

`4` El algoritmo no progresa adecuadamente.

66 mnewton

66.1 Funciones y variables para mnewton

`newtonepsilon` [Variable opcional]

Valor por defecto: `10.0^(-fpprec/2)`

Precisión que determina cuando la función `mnewton` ha conseguido una convergencia aceptable. Si `newtonepsilon` es un número decimal de precisión arbitraria (bigfloat), entonces `mnewton` realiza los cálculos en ese formato.

Véase también `mnewton`.

`newtonmaxiter` [Variable opcional]

Valor por defecto: `50`

Número máximo de iteraciones para la función `mnewton` en caso de que no se produzca convergencia, o de que ésta se haga muy lenta.

Véase también `mnewton`.

`mnewton` (*FuncList*,*VarList*,*GuessList*) [Función]

Resolución de sistemas de ecuaciones no lineales por el método de Newton. *FuncList* es la lista de ecuaciones a resolver, *VarList* es la lista con los nombres de las incógnitas y *GuessList* es la lista de aproximaciones iniciales.

La solución se devuelve en el mismo formato que lo hace la función `solve()`. Si no se le encuentra solución al sistema, se obtiene `[]` como respuesta.

Esta función se controla con las variables globales `newtonepsilon` y `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]]
```

La variable `newtonepsilon` controla la precisión de las aproximaciones. También controla si los cálculos se realizan con precisión doble o arbitraria (bigfloats).

```
(%i1) load("mnewton")$

(%i2) (fpprec : 25, newtonepsilon : bfloat(10^(-fpprec+5)))$

(%i3) mnewton([2*3^u-v/u-5, u+2^v-4], [u, v], [2, 2]);
(%o3) [[u = 1.066618389595406772591173b0,
                                  v = 1.552564766841786450100418b0]]
```

Antes de hacer uso de esta función ejecútese `load("mnewton")`. Véanse también `newtonepsilon` y `newtonmaxiter`.

67 numericalio

67.1 Introducción a numericalio

El paquete `numericalio` define funciones para leer y escribir ficheros de datos y flujos. Las funciones de entrada y salida en formato texto pueden leer y escribir números (enteros, decimales o decimales grandes), símbolos y cadenas. Las funciones de entrada y salida en formato binario sólo pueden leer y escribir números decimales.

Si ya existe una lista, matriz o array para almacenar los datos de entrada, las funciones de entrada de `numericalio` pueden escribir los datos directamente en estos objetos. En caso contrario, `numericalio` tratará de generar el objeto apropiado para almacenar estos datos.

67.1.1 Entrada y salida en formato texto

In plain-text input and output, it is assumed that each item to read or write is an atom:

En la entrada y salida de datos en formato texto se supone que cada dato es un átomo: un número entero, decimal, decimal grande, una cadena o un símbolo; no se admiten fracciones, números complejos o cualquier otra expresión no atómica. Estas funciones pueden llegar a realizar operaciones válidas con expresiones no atómicas, pero estos resultados no se documentan y están sujetos a cambios ulteriores.

Los átomos, tanto en los ficheros de entrada como en los de salida, tienen el mismo formato que en los ficheros por lotes de Maxima o en la consola interactiva. En particular, las cadenas deben encerrarse entre comillas dobles, la barra invertida `\` evita cualquier interpretación especial del carácter siguiente, y el símbolo de interrogación `?` se reconoce como el comienzo de un símbolo de Lisp. No se reconoce ningún carácter de continuación de línea interrumpida.

67.1.2 Separadores válidos para lectura

Las funciones para la entrada y salida de datos en formato texto tiene un argumento opcional, *separator_flag*, para indicar qué carácter se utiliza como separador.

Para la entrada de texto se reconocen los siguientes valores de la variable *separator_flag*: `comma` para los valores separados por comas, `pipe` para los valores separados por el carácter de la barra vertical `|`, `semicolon` para los valores separados por punto y coma `;`, y `space` para cuando los valores se separan por espacios o tabulaciones. Si el nombre del fichero tiene extensión `.csv` y no se especifica el argumento *separator_flag*, se tomará por defecto `comma`. Si el fichero tiene cualquier otra extensión diferente de `.csv` y no se especifica `separator_flag`, se usará por defecto `space`.

En la entrada de texto, varios espacios y tabulaciones sucesivos cuentan como un único separador. Sin embargo, varias comas, barras verticales o punto y comas sucesivos se interpretan que tienen el símbolo `false` entre ellos; por ejemplo, `1234,,Foo` se interpreta lo mismo que si fuese `1234,false,Foo`. En la salida, los átomos `false` deben escribirse explícitamente, por lo que la lista `[1234, false, Foo]` debe escribirse `1234,false,Foo`.

67.1.3 Separadores válidos para escritura

Para la entrada de texto se acepta `tab` como valor de *separator_flag* para datos separados por tabuladores, así como `comma`, `pipe`, `semicolon` y `space`.

En la escritura de texto, el átomo`false` se escribe tal cual y una lista `[1234, false, Foo]` se escribe `1234,false,Foo`.

67.1.4 Entrada y salida de decimales en formato binario

Las funciones de `numericalio` pueden leer y escribir números decimales en coma flotante de 8 bytes del estándar IEEE 754. Estos números se pueden escribir empezando por el byte menos significativo o por el más significativo, según lo indique la variable global `assume_external_byte_order`. Por defecto, `numericalio` los almacena con el byte más significativo primero.

Cualesquiera otros tipos de decimales son transformados a 8 bytes. El paquete `numericalio` no puede leer ni escribir datos binarios no numéricos.

Ciertos entornos Lisp no reconocen valores especiales del estándar IEEE 754 (más o menos infinito, valores no numéricos, valores no normales). El efecto que pueda producir la lectura de tales valores por parte de `numericalio` es imprevisible.

`numericalio` incluye funciones para abrir un flujo de lectura o escritura de flujos de bytes.

67.2 Funciones y variables para entrada y salida en formato texto

`read_matrix` (*S*) [Función]
`read_matrix` (*S*, *M*) [Función]
`read_matrix` (*S*, *separator_flag*) [Función]
`read_matrix` (*S*, *M*, *separator_flag*) [Función]

`read_matrix(`*S*`)` lee la fuente *S* y devuelve su contenido completo en forma de matriz. El tamaño de la matriz se deduce de los datos de entrada: cada fila del fichero forma una fila de la matriz. Si hay filas con diferente número de elementos, `read_matrix` emite un mensaje de error.

`read_matrix(`*S*`, `*M*`)` lee la fuente *S* y va almacenando su contenido en la matriz *M*, hasta que *M* esté llena o hasta que se consuma la fuente. Los datos se almacenan fila a fila. Los datos de entrada no necesitan tener el mismo número de filas y columnas que *M*.

La fuente *S* puede ser el nombre de un fichero o de un flujo.

Los valores aceptados para *separator_flag* son: `comma`, `pipe`, `semicolon` y `space`. Si no se especifica un valor para *separator_flag*, se supone que los datos están separados por espacios.

`read_array` (*S*, *A*) [Función]
`read_array` (*S*, *A*, *separator_flag*) [Función]

Guarda el contenido de la fuente *S* en el array *A*, hasta que *A* esté lleno o hasta que se consuma la fuente. Los datos se almacenan fila a fila. Los datos de entrada no necesitan tener el mismo número de filas y columnas que *A*.

La fuente *S* puede ser el nombre de un fichero o de un flujo.

Los valores aceptados para *separator_flag* son: `comma`, `pipe`, `semicolon` y `space`. Si no se especifica un valor para *separator_flag*, se supone que los datos están separados por espacios.

`read_hashed_array` (*S*, *A*) [Función]
`read_hashed_array` (*S*, *A*, *separator_flag*) [Función]

Lee la fuente *S* y devuelve su contenido completo en forma de array de claves. La fuente *S* puede ser el nombre de un fichero o de un flujo.

`read_hashed_array` interpreta el primer elemento de cada fila como una clave, asociando el resto de la fila, en formato de lista, a la clave. Por ejemplo, la secuencia `567 12 17 32 55` equivale a `A[567]: [12, 17, 32, 55]$`. Las filas no necesitan tener todas ellas el mismo número de elementos.

Los valores aceptados para *separator_flag* son: `comma`, `pipe`, `semicolon` y `space`. Si no se especifica un valor para *separator_flag*, se supone que los datos están separados por espacios.

`read_nested_list` (*S*) [Función]
`read_nested_list` (*S*, *separator_flag*) [Función]

Lee la fuente *S* y devuelve su contenido completo en forma de lista anidada. La fuente *S* puede ser el nombre de un fichero o de un flujo.

`read_nested_list` devuelve una lista que tiene una sublista por cada fila de entrada. Los filas de entrada no necesitan tener todas ellas el mismo número de elementos. Las filas en blanco no se ignoran, sino que se convierten en listas vacías

Los valores aceptados para *separator_flag* son: `comma`, `pipe`, `semicolon` y `space`. Si no se especifica un valor para *separator_flag*, se supone que los datos están separados por espacios.

`read_list` (*S*) [Función]
`read_list` (*S*, *L*) [Función]
`read_list` (*S*, *separator_flag*) [Función]
`read_list` (*S*, *L*, *separator_flag*) [Función]

`read_list(S)` lee la fuente *S* y devuelve su contenido como una lista simple.

`read_list(S, L)` guarda el contenido de la fuente *S* en la lista *L*, hasta que *L* esté llena o hasta que se consuma la fuente.

La fuente *S* puede ser el nombre de un fichero o de un flujo.

Los valores aceptados para *separator_flag* son: `comma`, `pipe`, `semicolon` y `space`. Si no se especifica un valor para *separator_flag*, se supone que los datos están separados por espacios.

`write_data` (*X*, *D*) [Función]
`write_data` (*X*, *D*, *separator_flag*) [Función]

Escribe el objeto *X* en el destino *D*.

`write_data` escribe una matriz fila a fila; cada línea de entrada se corresponde con una fila.

`write_data` escribe un array creado por `array` o `make_array` fila a fila, con una nueva línea al final de cada bloque de datos. Los bloques de mayores dimensiones se separan con líneas adicionales.

`write_data` escribe un array de claves con cada clave seguida de su lista asociada en una sola línea.

`write_data` escribe una lista anidada con una sublista por línea.

`write_data` escribe una lista simple en una única fila.

El destino *D* puede ser el nombre de un fichero o un flujo; en el primer caso, la variable global `file_output_append` controla si el fichero de salida es ampliado con la nueva información o si se borra antes; en el segundo caso, no se realiza ningún tipo de acción por parte de `write_data` después de que se hayan escrito los datos; en particular, el flujo se mantiene abierto.

Los valores aceptados para *separator_flag* son: `comma`, `pipe`, `semicolon` y `space`. Si no se especifica un valor para *separator_flag*, se supone que los datos están separados por espacios.

67.3 Funciones y variables para entrada y salida en formato binario

`assume_external_byte_order` (*byte_order_flag*) [Función]

Le indica a `numericalio` el orden de los bytes en que debe leer y escribir los datos. Los valores que reconoce *byte_order_flag* son dos: `lsb`, que indica que el byte menos significativo debe ser el primero, y `msb`, que indica que el byte más significativo es el que debe ir en primer lugar.

En caso de no hacer ninguna selección, `numericalio` interpreta que es el byte más significativo el que se debe leer o escribir primero.

`openr_binary` (*file_name*) [Función]

Devuelve un flujo de entrada de bytes no signados para la lectura del fichero de nombre *file_name*.

`openw_binary` (*file_name*) [Función]

Devuelve un flujo de entrada de bytes no signados para la escritura en el fichero de nombre *file_name*.

`opena_binary` (*file_name*) [Función]

Devuelve un flujo de entrada de bytes no signados para añadir datos al fichero de nombre *file_name*.

`read_binary_matrix` (*S*, *M*) [Función]

Lee números decimales en coma flotante de 8 bytes desde la fuente *S* y los va almacenando en la matriz *M*, bien hasta que *M* se llene, o bien hasta que la fuente se haya consumido. La matriz *M* se rellena fila a fila.

La fuente *S* puede ser el nombre de un fichero o un flujo.

El orden de los bytes de los datos procedentes de la fuente se especifican mediante `assume_external_byte_order`.

`read_binary_array` (*S*, *A*) [Función]

Lee números decimales en coma flotante de 8 bytes desde la fuente *S* y los va almacenando en el array *A*, bien hasta que *A* se llene, o bien hasta que la fuente se haya consumido. *A* debe ser un array creado por `array` o por `make_array`. El array *A* se rellena fila a fila.

La fuente *S* puede ser el nombre de un fichero o un flujo.

El orden de los bytes de los datos procedentes de la fuente se especifican mediante `assume_external_byte_order`.

`read_binary_list` (*S*) [Función]
`read_binary_list` (*S*, *L*) [Función]

`read_binary_list(S)` lee el contenido completo de la fuente de datos *S* como una secuencia de números decimales en coma flotante de 8 bytes en formato binario, devolviéndolos en forma de lista.

La fuente *S* puede ser el nombre de un fichero o un flujo.

`read_binary_list(S, L)` lee números decimales en coma flotante de 8 bytes en formato binario desde la fuente *S* y los almacena en la lista *L*, bien hasta que ésta esté llena, o bien hasta que se consuman los datos de la fuente.

El orden de los bytes de los datos procedentes de la fuente se especifican mediante `assume_external_byte_order`.

`write_binary_data` (*X*, *D*) [Función]

Escribe el objeto *X*, que contiene números decimales en coma flotante de 8 bytes del estándar IEEE 754, en el destino *D*. Cualesquiera otros tipos de decimales son transformados a 8 bytes. `write_binary_data` no puede escribir datos no numéricos.

El objeto *X* puede ser una lista, una lista anidada, una matriz, o un array creado con `array` o `make_array`; *X* no puede ser ni un array no declarado ni cualquier otro tipo de objeto distinto a los citados. `write_binary_data` escribe las listas anidadas, las matrices y los arrays fila a fila.

El destino *D* puede ser el nombre de un fichero o un flujo; en el primer caso, la variable global `file_output_append` controla si el fichero de salida es ampliado con la nueva información o si se borra antes; en el segundo caso, no se realiza ningún tipo de acción por parte de `write_binary_data` después de que se hayan escrito los datos; en particular, el flujo se mantiene abierto.

El orden de los bytes de los datos procedentes de la fuente se especifican mediante `assume_external_byte_order`.

68 opsubst

68.1 Funciones y variables para opsubst

`opsubst` (*f*,*g*,*e*) [Función]
`opsubst` (*g*=*f*,*e*) [Función]
`opsubst` ([*g1*=*f1*,*g2*=*f2*,..., *gn*=*fn*],*e*) [Función]

La función `opsubst` es similar a la función `subst`, excepto por el hecho de que `opsubst` tan solo hace sustituciones de operadores en las expresiones. En general, si f es un operador en la expresión e, lo cambia por g en la expresión e.

Para determinar el operador, `opsubst` asigna a `inflag` el valor `true`, lo cual significa que `opsubst` sustituye el operador interno de la expresión, no el mostrado en la salida formateada.

Ejemplo:

```
(%i1) load ("opsubst")$

(%i2) opsubst(f,g,g(g(x)));
(%o2)                      f(f(x))
(%i3) opsubst(f,g,g(g));
(%o3)                        f(g)
(%i4) opsubst(f,g[x],g[x](z));
(%o4)                        f(z)
(%i5) opsubst(g[x],f, f(z));
(%o5)                       g (z)
                             x
(%i6) opsubst(tan, sin, sin(sin));
(%o6)                      tan(sin)
(%i7) opsubst([f=g,g=h],f(x));
(%o7)                        h(x)
```

Internamente, Maxima no hace uso de los operadores de negación unaria, de división ni de la resta, por lo que:

```
(%i8) opsubst("+","-",a-b);
(%o8)                      a - b
(%i9) opsubst("f","-",-a);
(%o9)                       - a
(%i10) opsubst("^^","/",a/b);
                             a
(%o10)                       -
                             b
```

La representación interna de -a*b es *(-1,a,b), de modo que

```
(%i11) opsubst("[","*", -a*b);
(%o11)                   [- 1, a, b]
```

Si alguno de los operadores no es un símbolo de Maxima, se emitirá un mensaje de 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);
```

Sin embargo se permiten operadores subindicados:

```
(%i13) opsubst(g[5],f, f(x));
(%o13)                      g (x)
                             5
```

Antes de hacer uso de esta función ejecútese `load("opsubst")`.

69 orthopoly

69.1 Introducción a polinomios ortogonales

El paquete `orthopoly` contiene funciones para la evaluación simbólica y numérica de diversos tipos de polinomios ortogonales, como los de Chebyshev, Laguerre, Hermite, Jacobi, Legendre y ultraesféricos (Gegenbauer). Además, `orthopoly` soporta las funciones esféricas de Bessel, Hankel y armónicas.

Referencias:

- Abramowitz y Stegun, *Handbook of Mathematical Functions*, (1972, décima reimpresión, capítulo 22)
- Gradshteyn y Ryzhik, *Table of Integrals, Series y Products*, (1980, edición corregida y ampliada)
- Eugen Merzbacher, *Quantum Mechanics*, (1970, segunda edición)

El paquete `orthopoly`, junto con su documentación, fue escrito por Barton Willis de la Universidad de Nebraska en Kearney. El paquete se distribuye con la licencia GNU General Public License (GPL).

69.1.1 Iniciándose con orthopoly

`load ("orthopoly")` carga el paquete `orthopoly`.

Para obtener el polinomio de Legendre de tercer orden,

```
(%i1) legendre_p (3, x);
                        3             2
             5 (1 - x)    15 (1 - x)
(%o1)      - ---------- + ---------- - 6 (1 - x) + 1
                  2             2
```

Para expresarlo como una suma de potencias de x, aplíquese *ratsimp* o *rat* al resultado.

```
(%i2) [ratsimp (%), rat (%)];
                          3           3
                       5 x  - 3 x  5 x  - 3 x
(%o2)/R/              [----------, ----------]
                           2           2
```

De forma alternativa, conviértase el segundo argumento de to `legendre_p` (su variable "principal") a una expresión racional canónica (canonical rational expression, CRE)).

```
(%i1) legendre_p (3, rat (x));
                                3
                             5 x  - 3 x
(%o1)/R/                     ----------
                                2
```

Para la evaluación numérica, `orthopoly` hace uso del análisis de error de ejecución para estimar una cota superior del error. Por ejemplo,

```
(%i1) jacobi_p (150, 2, 3, 0.2);
(%o1) interval(- 0.062017037936715, 1.533267919277521E-11)
```

Los intervalos tienen la forma `interval (c, r)`, donde c es el centro y r el radio del intervalo. Puesto que Maxima no soporta aritmética de intervalos, en algunas situaciones, como en los gráficos, puede ser necesario ignorar el error y utilizar el centro del intervalo. Para conseguirlo conviene asignar a la variable `orthopoly_returns_intervals` el valor `false`.

```
(%i1) orthopoly_returns_intervals : false;
(%o1)                          false
(%i2) jacobi_p (150, 2, 3, 0.2);
(%o2)                  - 0.062017037936715
```

Véase la sección *Evaluación numérica* para más información.

La mayor parte de las funciones de `orthopoly` tienen una propiedad `gradef`; así,

```
(%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
```

La función `unit_step` del segundo ejemplo evita el error que aparecería al evaluar la expresión con n igual a 0.

```
(%i3) ev (%, n = 0);
(%o3)                           0
```

La propiedad "gradef" sólo se aplica a la variable principal; derivadas respecto de las otras variables darán lugar normalmente a mensajes de error; por ejemplo,

```
(%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);
```

Generalmente, las funciones de `orthopoly` se distribuyen sobre listas y matrices. Al objeto de que la evaluación se realice completamente, las variables opcionales `doallmxops` y `listarith` deben valer ambas `true`, que es el valor por defecto. Para ilustrar la distribución sobre matrices sirve el siguiente ejemplo

```
(%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       ]
```

En el segundo ejemplo, el elemento `i, j`-ésimo es `hermite (2, m[i,j])`, que no es lo mismo que calcular `-2 + 4 m . m`, según se ve en el siguiente ejemplo.

```
(%i4) -2 * matrix ([1, 0], [0, 1]) + 4 * m . m;
                    [ 4 x y - 2      0     ]
(%o4)               [                      ]
                    [     0      4 x y - 2 ]
```

Si se evalúa una función en un punto fuera de su dominio de definición, generalmente `orthopoly` devolverá la función sin evaluar. Por ejemplo,

```
(%i1) legendre_p (2/3, x);
(%o1)                          P   (x)
                                2/3
```

`orthopoly` da soporte a la traducción de expresiones al formato TeX y la representación bidimensional en el 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
```

69.1.2 Limitaciones

Cuando una expresión contenga varios polinomios ortogonales con órdenes simbólicos, es posible que aunque la expresión sea nula, Maxima sea incapaz de simplificarla a cero, por lo que si se divide por esta cantidad, aparecerán problemas. Por ejemplo, la siguiente expressión se anula para enteros n mayores que 1, no pudiendo Maxima reducirla a cero.

```
(%i1) (2*n - 1) * legendre_p (n - 1, x) * x - n * legendre_p (n, x)
                + (1 - n) * legendre_p (n - 2, x);
(%o1)  (2 n - 1) P     (x) x - n P (x) + (1 - n) P     (x)
                  n - 1           n               n - 2
```

Para un valor específico de n se puede reducir la expresión a cero.

```
(%i2) ev (% ,n = 10, ratsimp);
(%o2)                           0
```

Generalmente, la forma polinomial de un polinomio ortogonal no es la más apropiada para su evaluación numérica. Aquí un ejemplo.

```
(%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
```

Este resultado se puede mejorar expandiendo el polinomio y evaluando a continuación, lo que da una aproximación mejor.

```
(%i5) p : expand(p)$
(%i6) subst (0.2, x, p);
(%o6) 0.18413609766122982
```

Sin embargo esto no vale como regla general; la expansión del polinomio no siempre da como resultado una expresión más fácil de evaluar numéricamente. Sin duda, la mejor manera de hacer la evaluación numérica consiste en hacer que uno o más de los argumentos de la función sean decimales en coma flotante; de esta forma se utilizarán algoritmos decimales especializados para hacer la evaluación.

La función `float` de Maxima trabaja de forma indiscriminada; si se aplica `float` a una expresión que contenga un polinomio ortogonal con el grado u orden simbólico, éstos se pueden transformar en decimales y la expresión no ser evaluada de forma completa. Considérese

```
(%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
```

La expresión en `(%o3)` no da como resultado un decimal en coma flotante; `orthopoly` no reconoce decimales donde espera que haya enteros. De forma semejante, la evaluación numérica de la función `pochhammer` para órdenes que excedan `pochhammer_max_index` puede ser problemática; considérese

```
(%i1) x :  pochhammer (1, 10), pochhammer_max_index : 5;
(%o1)                          (1)
                                  10
```

Aplicando `float` no da para x un valor decimal

```
(%i2) float (x);
(%o2)                        (1.0)
                                  10.0
```

A fin de evaluar x como decimal, es necesario asignar a `pochhammer_max_index` en valor 11 o mayor y aplicar `float` a x.

```
(%i3) float (x), pochhammer_max_index : 11;
(%o3)                        3628800.0
```

El valor por defecto de `pochhammer_max_index` es 100; cámbiese este valor tras cargar el paquete `orthopoly`.

Por último, téngase en cuenta que las referencias bibliográficas no coinciden a la hora de definir los polinomios ortogonales; en `orthopoly` se han utilizado normalmente las convenciones seguidas por Abramowitz y Stegun.

Cuando se sospeche de un fallo en `orthopoly`, compruébense algunos casos especiales a fin de determinar si las definiciones de las que el usuario parte coinciden con las utilizadas por el paquete `orthopoly`. A veces las definiciones difieren por un factor de normalización; algunos autores utilizan versiones que hacen que las familias sean ortogonales en otros intervalos diferentes de $(-1, 1)$. Así por ejemplo, para definir un polinomio de Legendre ortogonal en $(0, 1)$ defínase

```
(%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
```

69.1.3 Evaluación numérica

La mayor parte de las funciones de `orthopoly` realizan análisis de errores en tiempo de ejecución para estimar el error en la evaluación decimal, a excepción de las funciones esféricas de Bessel y los polinomios asociados de Legendre de segunda especie. Para la evaluación numérica, las funciones esféricas de Bessel hacen uso de funciones SLATEC. No se lleva a cabo ningún método especial de evaluación numérica para los polinomios asociados de Legendre de segunda especie.

Es posible, aunque improbable, que el error obtenido en las evaluaciones numéricas exceda al error estimado.

Los intervalos tienen la forma `interval (c, r)`, siendo c el centro del intervalo y r su radio. El centro del intervalo puede ser un número complejo, pero el radio será siempre un número real positivo.

He aquí un ejemplo:

```
(%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
```

Se comprueba que el error es menor que el estimado

```
(%i4) is (abs (part (y0, 1) - y1) < part (y0, 2));
(%o4)                          true
```

En este ejemplo el error estimado es una cota superior para el error verdadero.

Maxima no da soporte a la aritmética de intervalos.

```
(%i1) legendre_p (7, 0.1) + legendre_p (8, 0.1);
(%o1) interval(0.18032072148437508, 3.1477135311021797E-15)
        + interval(- 0.19949294375000004, 3.3769353084291579E-15)
```

El usuario puede definir operadores aritméticos para los intervalos. Para definir la suma de intervalos se puede hacer

```
(%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)
```

Las rutinas especiales para cálculo numérico son llamadas cuando los argumentos son complejos. Por ejemplo,

```
(%i1) legendre_p (10, 2 + 3.0*%i);
(%o1) interval(- 3.876378825E+7 %i - 6.0787748E+7,
                                          1.2089173052721777E-6)
```

Compárese con el valor verdadero.

```
(%i1) float (expand (legendre_p (10, 2 + 3*%i)));
(%o1)          - 3.876378825E+7 %i - 6.0787748E+7
```

Además, cuando los argumentos son números decimales grandes (*big floats*), se realizan llamadas a las rutinas numéricas especiales; sin embargo, los decimales grandes se convierten previamente a doble precisión y de este tipo serán también los resultados.

```
(%i1) ultraspherical (150, 0.5b0, 0.9b0);
(%o1) interval(- 0.043009481257265, 3.3750051301228864E-14)
```

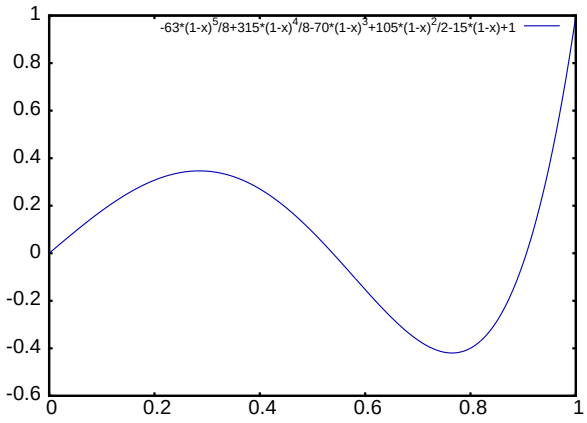
69.1.4 Gráficos y orthopoly

Para representar gráficamente expresiones que contengan polinomios ortogonales se deben hacer dos cosas:

1. Asignar a la variable opcional `orthopoly_returns_intervals` el valor `false`,
2. Comentar (con apóstrofo) las llamadas a las funciones de `orthopoly`.

Si las llamadas a las funciones no se comentan, Maxima las evalúa a polinomios antes de hacer el gráfico, por lo que el código especializado en el cálculo numérico no es llamado. Aquí hay un ejemplo de cómo se debe hacer para representar gráficamente una expresión que contiene un polinomio de Legendre:

```
(%i1) plot2d ('(legendre_p (5, x)), [x, 0, 1]),
      orthopoly_returns_intervals : false;
(%o1)
```



La expresión `legendre_p (5, x)` se comenta *completamente*, que no es lo mismo que comentar el nombre de la función, como en `'legendre_p (5, x)`.

69.1.5 Miscelánea de funciones

El paquete `orthopoly` define el símbolo de Pochhammer y la función de escalón unidad en sentencias gradef.

Para convertir los símbolos de Pochhammer en cocientes o funciones gamma, hágase uso de `makegamma`.

```
(%i1) makegamma (pochhammer (x, n));
                            gamma(x + n)
(%o1)                       ------------
                              gamma(x)
(%i2) makegamma (pochhammer (1/2, 1/2));
                                  1
(%o2)                           ---------
                                sqrt(%pi)
```

Las derivadas del símbolo de Pochhammer se dan en términos de la función `psi`.

```
(%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
```

Es necesario tener cuidado con la expresión en `(%o1)`, pues la diferencia de las funciones `psi` tiene polos cuando *x* = `-1, -2, .., -`*n*. Estos polos se cancelan con factores de `pochhammer (x, n)` haciendo que la derivada sea un polinomio de grado *n* `-1` si *n* es entero positivo.

El símbolo de Pochhammer se define para órdenes negativos a través de su representación como cociente de funciones gamma. Considérese

```
(%i1) q : makegamma (pochhammer (x, n));
                            gamma(x + n)
(%o1)                       ------------
                              gamma(x)
(%i2) sublis ([x=11/3, n= -6], q);
                                 729
```

```
(%o2)                          - ----
                                 2240
```

De forma alternativa, es posible llegar a este resultado directamente.

```
(%i1) pochhammer (11/3, -6);
                                 729
(%o1)                          - ----
                                 2240
```

La función de escalón unidad es continua por la izquierda; así,

```
(%i1) [unit_step (-1/10), unit_step (0), unit_step (1/10)];
(%o1)                       [0, 0, 1]
```

En caso de ser necesaria una función escalón unidad que no sea continua ni por la izquierda ni por la derecha en el origen, se puede definir haciendo uso de `signum`; por ejemplo,

```
(%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
```

No se debe redefinir la función `unit_step`, ya que parte del código de `orthopoly` requiere que la función escalón sea continua por la izquierda.

69.1.6 Algoritmos

En general, el paquete `orthopoly` gestiona la evaluación simbólica a través de la representación hipergeométrica de los polinomios ortogonales. Las funciones hipergeométricas se evalúan utilizando las funciones (no documentadas) `hypergeo11` y `hypergeo21`. Excepciones son las funciones de Bessel de índice semi-entero y las funciones asociadas de Legendre de segunda especie; las funciones de Bessel de índice semi-entero se evalúan utilizando una representación explícita, mientras que la función asociada de Legendre de segunda especie se evalúa recursivamente.

En cuanto a la evaluación numérica, la mayor parte de las funciones se convierten a su forma hipergeométrica, evaluándolas mediante recursión. Además, las excepciones son las funciones de Bessel de índice semi-entero y las funciones asociadas de Legendre de segunda especie. Las funciones de Bessel de índice semi-entero se evalúan numéricamente con código SLATEC.

69.2 Funciones y variables para polinomios ortogonales

`assoc_legendre_p` (*n*, *m*, *x*) [Función]

Función asociada de Legendre de primera especie de grado *n* y orden *m*.

Referencia: Abramowitz y Stegun, ecuaciones 22.5.37, página 779, 8.6.6 (segunda ecuación), página 334 y 8.2.5, página 333.

`assoc_legendre_q` (*n*, *m*, *x*) [Función]

Función asociada de Legendre de segunda especie de grado *n* y orden *m*.

Referencia: Abramowitz y Stegun, ecuaciones 8.5.3 y 8.1.8.

`chebyshev_t` (*n*, *x*) [Función]
Función de Chebyshev de primera especie.
Referencia: Abramowitz y Stegun, ecuación 22.5.47, página 779.

`chebyshev_u` (*n*, *x*) [Función]
Función de Chebyshev de segunda especie.
Referencia: Abramowitz y Stegun, ecuación 22.5.48, página 779.

`gen_laguerre` (*n*, `a`, *x*) [Función]
Polinomio de Laguerre generalizado de grado *n*.
Referencia: Abramowitz y Stegun, ecuación 22.5.54, página 780.

`hermite` (*n*, *x*) [Función]
Polinomio de Hermite.
Referencia: Abramowitz y Stegun, ecuación 22.5.55, página 780.

`intervalp` (*e*) [Función]
Devuelve `true` si la entrada es un intervalo y `false` en caso contrario.

`jacobi_p` (*n*, `a`, *b*, *x*) [Función]
Polinomio de Jacobi.
Los polinomios de Jacobi están definidos para todo *a* y *b*; sin embargo, el peso `(1 - x)^a (1 + x)^b` no es integrable para `a <= -1` o *b* `<= -1`.
Referencia: Abramowitz y Stegun, ecuación 22.5.42, página 779.

`laguerre` (*n*, *x*) [Función]
Polinomio de Laguerre.
Referencia: Abramowitz y Stegun, ecuaciones 22.5.16 y 22.5.54, página 780.

`legendre_p` (*n*, *x*) [Función]
Polinomio de Legendre de primera especie.
Referencia: Abramowitz y Stegun, ecuaciones 22.5.50 y 22.5.51, página 779.

`legendre_q` (*n*, *x*) [Función]
Polinomio de Legendre de segunda especie.
Referencia: Abramowitz y Stegun, ecuaciones 8.5.3 y 8.1.8.

`orthopoly_recur` (*f*, *args*) [Función]
Devuelve una relación recursiva para la familia de funciones ortogonales *f* con argumentos *args*. La recursión se hace con respecto al grado del polinomio.

```
(%i1) orthopoly_recur (legendre_p, [n, x]);
                     (2 n + 1) P (x) x - n P     (x)
                                n           n - 1
(%o1)   P     (x) = ------------------------------
         n + 1                   n + 1
```

El segundo argumento de `orthopoly_recur` debe ser una lista con el número correcto de argumentos para la función *f*; si no lo es, Maxima emite un mensaje de 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);
```

Además, si f no es el nombre de ninguna de las familias de polinomios ortogonales, se emite otro mensaje de error.

```
(%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 opcional]

Valor por defecto: `true`

Si `orthopoly_returns_intervals` vale `true`, los números decimales en coma flotante se retornan con el formato `interval (c, r)`, donde c es el centro del intervalo y r su radio. El centro puede ser un número complejo, en cuyo caso el intervalo es un disco en el plano complejo.

`orthopoly_weight (f, args)` [Función]

Devuelve una lista con tres elementos; el primer elemento es la fórmula del peso para la familia de polinomios ortogonales f con los argumentos dados por la lista *args*; el segundo y tercer elementos son los extremos inferior y superior del intervalo de ortogonalidad. Por ejemplo,

```
(%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
```

La variable principal de f debe ser un símbolo, en caso contrario Maxima emite un mensaje de error.

`pochhammer (n, x)` [Función]

Símbolo de Pochhammer. Para enteros no negativos n con `n <= pochhammer_max_index`, la expresión `pochhammer (x, n)` se evalúa como el producto `x (x + 1) (x + 2) ... (x + n - 1)` si `n > 0` y como 1 si `n = 0`. Para n negativo, `pochhammer (x, n)` se define como `(-1)^n / pochhammer (1 - x, -n)`. Así por ejemplo,

```
(%i1) pochhammer (x, 3);
(%o1)                   x (x + 1) (x + 2)
(%i2) pochhammer (x, -3);
                                 1
(%o2)               - -----------------------
                      (1 - x) (2 - x) (3 - x)
```

A fin de convertir el símbolo de Pochhammer en un cociente de funciones gamma (véase Abramowitz y Stegun, ecuación 6.1.22), hágase uso de `makegamma`. Por ejemplo,

```
(%i1) makegamma (pochhammer (x, n));
                            gamma(x + n)
```

```
(%o1)                          ------------
                                 gamma(x)
```

Si n es mayor que `pochhammer_max_index` o si n es simbólico, `pochhammer` devuelve una forma nominal.

```
(%i1) pochhammer (x, n);
(%o1)                            (x)
                                    n
```

`pochhammer_max_index` [Variable opcional]

Valor por defecto: 100

`pochhammer (n, x)` se evalúa como un producto si y sólo si `n <= pochhammer_max_index`.

Ejemplos:

```
(%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
```

Referencia: Abramowitz y Stegun, ecuación 6.1.16, página 256.

`spherical_bessel_j (n, x)` [Función]

Función de Bessel esférica de primera especie.

Referencia: Abramowitz y Stegun, ecuaciones 10.1.8, página 437 y 10.1.15, página 439.

`spherical_bessel_y (n, x)` [Función]

Función de Bessel esférica de segunda especie.

Referencia: Abramowitz y Stegun, ecuaciones 10.1.9, página 437 y 10.1.15, página 439.

`spherical_hankel1 (n, x)` [Función]

Función esférica de Hankel de primera especie.

Referencia: Abramowitz y Stegun, ecuación 10.1.36, página 439.

`spherical_hankel2 (n, x)` [Función]

Función esférica de Hankel de segunda especie.

Referencia: Abramowitz y Stegun, ecuación 10.1.17, página 439.

`spherical_harmonic (n, m, x, y)` [Función]

Función armónica esférica.

Referencia: Merzbacher 9.64.

`unit_step (x)` [Función]

Función de escalón unidad continua por la izquierda, definida de tal forma que `unit_step (x)` se anula para `x <= 0` y es igual a 1 para `x > 0`.

En caso de ser necesaria una función escalón unidad que tome el valor 1/2 en el origen, utilícese `(1 + signum (x))/2`.

`ultrasphericalultraspherical` (*n*, a, *x*) [Función]

Polinomio ultraesférico o de Gegenbauer.

Referencia: Abramowitz y Stegun, ecuación 22.5.46, página 779.

70 romberg

70.1 Funciones y variables para romberg

romberg (*expr*, *x*, a, *b*) [Función]
romberg (*F*, a, *b*) [Función]

Integra numéricamente por el método de Romberg.

La llamada romberg(*expr*, *x*, a, *b*) devuelve una estimación de la integral integrate(*expr*, *x*, a, *b*). El argumento *expr* debe ser una expresión reducible a un valor decimal en coma flotante cuando *x* es a su vez un número decimal.

La llamada romberg(*F*, a, *b*) devuelve una estimación de la integral integrate(*F*(x), x, a, *b*), siendo x el único argumento de *F*. El argumento *F* debe ser una función en Lisp o en Maxima que devuelva un valor decimal en coma flotante cuando *x* es a su vez un número decimal; *F* puede ser el nombre de una función de Maxima traducida o compilada.

La exactitud de romberg se controla con las variables globales rombergabs y rombergtol. La función romberg termina con éxito su cálculo cuando la diferencia absoluta entre sucesivas aproximaciones es menor que rombergabs, o cuando la diferencia relativa de sucesivas aproximaciones es menor que rombergtol. Así, cuando rombergabs vale 0.0 (su valor por defecto) sólo tiene efecto el test del error relativo basado en romberg.

La función romberg reduce a mitades sucesivas la amplitud del paso un máximo de rombergit veces antes de abandonar el cómputo; el número máximo de evaluaciones del integrando es, por consiguiente, igual a 2^rombergit. De no satisfacerse el criterio de error establecido por rombergabs y rombergtol, romberg devuelve un mensaje de error. La función romberg hace siempre al menos rombergmin iteraciones; se trata de una heurística para evitar la finalización prematura cuando el integrando oscila mucho.

La función romberg evalúa el integrando repetidamente tras asignarle a la variable de integración un valor específico. Este criterio permite anidar llamadas a romberg para calcular integrales múltiples. Sin embargo, los errores de cálculo no tienen en cuenta los errores de las integraciones anidadas, por lo que tales errores pueden subestimarse. Por otro lado, métodos especialmente desarrollados para integraciones múltiples pueden dar la misma exactitud con menos evaluaciones del integrando.

Para hacer uso de esta función ejecútese primero load("romberg").

Véase también QUADPACK, un conjunto de funciones para integración numérica.

Ejemplos:

Una integración unidimensional.

```
(%i1) load ("romberg");
(%o1)    /usr/share/maxima/5.11.0/share/numeric/romberg.lisp
(%i2) f(x) := 1/((x - 1)^2 + 1/100) + 1/((x - 2)^2 + 1/1000)
                + 1/((x - 3)^2 + 1/200);
                    1                1                1
(%o2) f(x) := -------------- + --------------- + --------------
```

```
                        2    1           2    1           2    1
                (x - 1)  + ---   (x - 2)  + ----   (x - 3)  + ---
                           100             1000             200
(%i3) rombergtol : 1e-6;
(%o3)                 9.9999999999999995E-7
(%i4) rombergit : 15;
(%o4)                          15
(%i5) estimate : romberg (f(x), x, -5, 5);
(%o5)                  173.6730736617464
(%i6) exact : integrate (f(x), x, -5, 5);
(%o6) 10 sqrt(10) atan(70 sqrt(10))
 + 10 sqrt(10) atan(30 sqrt(10)) + 10 sqrt(2) atan(80 sqrt(2))
 + 10 sqrt(2) atan(20 sqrt(2)) + 10 atan(60) + 10 atan(40)
(%i7) abs (estimate - exact) / exact, numer;
(%o7)               7.5527060865060088E-11
```

Una integración bidimensional, implementada mediante llamadas anidadas a `romberg`.

```
(%i1) load ("romberg");
(%o1)    /usr/share/maxima/5.11.0/share/numeric/romberg.lisp
(%i2) g(x, y) := x*y / (x + y);
                                    x y
(%o2)                  g(x, y) := -----
                                   x + y
(%i3) rombergtol : 1e-6;
(%o3)                 9.9999999999999995E-7
(%i4) estimate : romberg (romberg (g(x, y), y, 0, x/2), x, 1, 3);
(%o4)                  0.81930239628356
(%i5) assume (x > 0);
(%o5)                         [x > 0]
(%i6) integrate (integrate (g(x, y), y, 0, x/2), x, 1, 3);
                                      3
                                   2 log(-) - 1
                    9                  2        9
(%o6)       - 9 log(-) + 9 log(3) + ------------ + -
                    2                  6        2
(%i7) exact : radcan (%);
                    26 log(3) - 26 log(2) - 13
(%o7)             - --------------------------
                                 3
(%i8) abs (estimate - exact) / exact, numer;
(%o8)                1.3711979871851024E-10
```

`rombergabs` [Variable opcional]

Valor por defecto: 0.0

La exactitud de `romberg` se controla con las variables globales `rombergabs` y `rombergtol`. La función `romberg` termina con éxito su cálculo cuando la diferencia absoluta entre sucesivas aproximaciones es menor que `rombergabs`, o cuando la diferencia relativa de sucesivas aproximaciones es menor que `rombergtol`. Así,

cuando `rombergabs` vale 0.0 (su valor por defecto) sólo tiene efecto el test del error relativo basado en `romberg`.

Véanse también `rombergit` y `rombergmin`.

`rombergit` [Variable opcional]

Valor por defecto: 11

La función `romberg` reduce a mitades sucesivas la amplitud del paso un máximo de `rombergit` veces antes de abandonar el cómputo; el número máximo de evaluaciones del integrando es, por consiguiente, igual a `2^rombergit`. La función `romberg` hace siempre al menos `rombergmin` iteraciones; se trata de una heurística para evitar la finalización prematura cuando el integrando oscila mucho.

Véanse también `rombergabs` y `rombergtol`.

`rombergmin` [Variable opcional]

Valor por defecto: 0

La función `romberg` hace siempre al menos `rombergmin` iteraciones; se trata de una heurística para evitar la finalización prematura cuando el integrando oscila mucho.

Véanse también `rombergit`, `rombergabs` y `rombergtol`.

`rombergtol` [Variable opcional]

Valor por defecto: 1e-4

La exactitud de `romberg` se controla con las variables globales `rombergabs` y `rombergtol`. La función `romberg` termina con éxito su cálculo cuando la diferencia absoluta entre sucesivas aproximaciones es menor que `rombergabs`, o cuando la diferencia relativa de sucesivas aproximaciones es menor que `rombergtol`. Así, cuando `rombergabs` vale 0.0 (su valor por defecto) sólo tiene efecto el test del error relativo basado en `romberg`.

Véanse también `rombergit` y `rombergmin`.

71 simplex

71.1 Introducción a simplex

El paquete `simplex` utiliza el algoritmo simplex para programación lineal.

Ejemplo:

```
(%i1) load("simplex")$
(%i2) minimize_lp(x+y, [3*x+2*y>2, x+4*y>3]);
                        9        7       1
(%o2)                  [--, [y = --, x = -]]
                        10       10      5
```

71.2 Funciones y variables para simplex

`epsilon_lp` [Variable opcional]

Valor por defecto: `10^-8`

Error epsilon utilizado en los cálculos numéricos de `linear_program`.

Véase también `linear_program`.

`linear_program` (*A*, *b*, *c*) [Función]

La función `linear_program` es una implementación del algoritmo simplex. La instrucción `linear_program(A, b, c)` calcula un vector x tal que minimiza `c.x` bajo las restricciones `A.x = b` y `x >= 0`. El argumento A es una matriz y los argumentos b y c son listas.

La función `linear_program` devuelve una lista que contiene el vector solución x y el valor mínimo de `c.x`. Si el problema no está acotado, devuelve el mensaje `"Problem not bounded!"` y si el problema no es factible, devuelve el mensaje `"Problem not feasible!"`.

Para usar esta función, cárguese primero el paquete con la instrucción `load("simplex");`.

Ejemplo:

```
(%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
```

Véanse también `minimize_lp`, `scale_lp` y `epsilon_lp`.

`maximize_lp` (*obj*, *cond*, [*pos*]) [Función]

Maximiza la función objetivo lineal *obj* sujeta a ciertas restricciones lineales *cond*. Véase `minimize_lp` para una descripción detallada de los argumentos y de la respuesta dada por esta función.

`minimize_lp` (*obj*, *cond*, [*pos*]) [Función]

Minimiza la función objetivo lineal *obj* sujeta a ciertas restricciones lineales *cond*, siendo ésta una lista de ecuaciones o inecuaciones lineales. En las inecuaciones estrictas se reemplaza `>` por `>=` y `<` por `<=`. El argumento opcional *pos* es una lista de variables de decisión que se suponen positivas.

Si el mínimo existe, `minimize_lp` devuelve una lista que contiene el valor mínimo de la función objetivo y una lista de valores para las variables de decisión con los que se alcanza el mínimo. Si el problema no está acotado, devuelve el mensaje `"Problem not bounded!"` y si el problema no es factible, devuelve el mensaje `"Problem not feasible!"`.

Las variables de decisión no se suponen no negativas. Si todas las variables de decisión son no negativas, asígnese el valor `true` a la variable `nonegative_lp`. Si sólo algunas de las variables de decisión son positivas, lístense en el argumento opcional *pos*, lo cual es más eficiente que añadir restricciones.

La función `minimize_lp` utiliza el algoritmo simplex implementado en la función `linear_program` de Maxima.

Para usar esta función, cárguese primero el paquete con la instrucción `load("simplex");`.

Ejemplos:

```
(%i1) minimize_lp(x+y, [3*x+y=0, x+2*y>2]);
                        4       6        2
(%o1)                  [-, [y = -, x = - -]]
                        5       5        5
(%i2) minimize_lp(x+y, [3*x+y>0, x+2*y>2]), nonegative_lp=true;
(%o2)                  [1, [y = 1, x = 0]]
(%i3) minimize_lp(x+y, [3*x+y=0, x+2*y>2]), nonegative_lp=true;
(%o3)                 Problem not feasible!
(%i4) minimize_lp(x+y, [3*x+y>0]);
(%o4)                 Problem not bounded!
```

Véanse también `maximize_lp`, `nonegative_lp` y `epsilon_lp`.

`nonegative_lp` [Variable opcional]

Valor por defecto: `false`

Si `nonegative_lp` vale `true` todas las variables de decisión pasadas a `minimize_lp` y a `maximize_lp` se suponen positivas.

Véase también `minimize_lp`.

72 simplification

72.1 Introducción a simplification

El directorio `maxima/share/simplification` contiene programas que implementan algunas reglas y funciones para simplificar expresiones, así como ciertas funciones no relacionadas con la simplificación.

72.2 Paquete absimp

El paquete `absimp` contiene reglas para aplicar patrones que extienden el sistema de reglas nativo de Maxima para las funciones `abs` y `signum`, respetando las relaciones establecidas con la función `assume` o con declaraciones tales como `modedeclare (m, even, n, odd)` para enteros pares o impares.

En el paquete `absimp` se definen las funciones `unitramp` y `unitstep` en términos de `abs` y `signum`.

La instrucción `load ("absimp")` carga este paquete y `demo (absimp)` desarrolla una demostración sobre el uso del mismo.

Ejemplos:

```
(%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)
```

72.3 Paquete facexp

El paquete `facexp` contiene varias funciones que le aportan al usuario la posibilidad de estructurar expresiones controlando su expansión. Esta capacidad es especialmente útil cuando la expresión contiene variables con significado físico, ya que se suele dar el caso de que la forma más sencilla para estas expresiones se obtiene cuando se expanden respecto de estas variables y luego se factoriza respecto de sus coeficientes. Si bien es cierto que este procedimiento no es difícil de llevar a cabo con las funciones estándar de Maxima, pueden ser necesarios algunos retoques adicionales que sí pueden ser más difíciles de hacer.

La función `facsum` y sus formas relacionadas proporcionan un método para controlar la estructura de expresiones. La función `collectterms` puede usarse para añadir dos o más expresiones que ya hayan sido simplificadas de la forma indicada, sin necesidad de volver a simplificar la expresión completa. Esta función puede ser útil cuando las expresiones sean largas.

La instrucción `load ("facexp")` carga este paquete y `demo (facexp)` hace una demostración sobre su uso.

facsum (*expr*, *arg_1*, ..., *arg_n*) [Función]

Devuelve una expresión equivalente a *expr*, la cual depende de los argumentos *arg_1*, ..., *arg_n*, y éstos pueden ser de cualquiera de las formas aceptables para `ratvars`, o listas de estas formas. Si los argumentos no son listas, la forma devuelta se expande completamente con respecto de los argumentos, siendo los coeficientes de tales argumentos factorizados. Estos coeficientes no contienen a ninguno de los argumentos, excepto quizás de una forma no racional.

En caso de que cualquiera de los argumentos sea una lista, entonces todos ellos se combinan en una única lista, y en lugar de llamar a `factor` para los coeficientes de los argumentos, `facsum` se llama a sí misma utilizando esta nueva lista única como lista de argumentos.

Es posible que se quiera utilizar `facsum` con respecto a expresiones más complicadas, tales como `log (x + y)`. Estos argumentos son también admisibles.

En ocasiones puede ser necesario obtener cualquiera de las formas anteriores especificadas por sus operadores principales. Por ejemplo, se puede querer aplicar `facsum` con respecto a todos los `log`; en este caso, se puede incluir entre los argumentos bien los `log` específicos que se quieran tratar de esta manera, bien la expresión `operator (log)` o `'operator (log)`. Si se quiere aplicar `facsum` a *expr* con respecto a los operadores *op_1*, ..., *op_n*, se debe evaluar `facsum (expr, operator (op_1, ..., op_n))`. La forma `operator` puede aparecer también dentro de las listas de argumentos.

Además, dándole valores a las variables opcionales `facsum_combine` y `nextlayerfactor` se puede controlar el resultado de `facsum`.

nextlayerfactor [Variable global]

Valor por defecto: `false`

Si `nextlayerfactor` vale `true`, las llamadas recursivas de `facsum` se aplican a los factores de la forma factorizada de los coeficientes de los argumentos.

Si vale `false`, `facsum` se aplica a cada coeficiente como un todo cada vez que se efectúen llamadas recursivas a `facsum`.

La inclusión del átomo `nextlayerfactor` en la lista de argumentos de `facsum` tiene el mismo efecto que `nextlayerfactor: true`, pero *solamente* para el siguiente nivel de la expresión. Puesto que `nextlayerfactor` toma siempre uno de los valores `true` o `false`, debe aparecer comentado (comilla simple) cada vez que aparezca en la lista de argumentos de `facsum`.

facsum_combine [Variable global]

Valor por defecto: `true`

La variable `facsum_combine` controla la forma del resultado final devuelto por `facsum` si su argumento es un cociente de polinomios. Si `facsum_combine` vale `false`, el resultado será una suma completamente expandida, pero si vale `true`, la expresión devuelta es un cociente de polinomios.

factorfacsum (*expr*, *arg_1*, ... *arg_n*) [Función]

Devuelve una expresión equivalente a *expr* obtenida aplicando `facsum` a los factores de *expr*, de argumentos *arg_1*, ... *arg_n*. Si alguno de los factores de *expr* se eleva a una potencia, tanto el factor como el exponente se procesarán de esta manera.

`collectterms` (*expr*, *arg_1*, ..., *arg_n*) [Función]

Si algunas expresiones fueron ya simplificadas con `facsum`, `factorfacsum`, `factenexpand`, `facexpten` o `factorfacexpten`, debiendo ser luego sumadas, puede ser conveniente combinarlas utilizando la función `collecterms`, la cual admite como argumentos todos aquéllos que se puedan pasar a las anteriormente citadas funciones, con la excepción de `nextlayerfactor`, que no tiene efecto alguno sobre `collectterms`. La ventaja de `collectterms` es que devuelve una forma similar a la de `facsum`, pero debido a que suma expresiones que ya han sido previamente procesadas, no necesita repetir esta operación, lo cual resulta ser especialmente útil cuando las expresiones a sumar son muy grandes.

72.4 Paquete functs

`rempart` (*expr*, *n*) [Función]

Elimina la parte *n* de la expresión *expr*.

Si *n* es una lista de la forma `[1, m]`, entonces las partes desde *l* a *m* serán eliminadas.

Para hacer uso de esta función ejecutar `load("functs")`.

`wronskian` ([*f_1*, ..., *f_n*], *x*) [Función]

Devuelve la matriz wronskiana de las expresiones *f_1*, ..., *f_n* dependeientes de la variable *x*. El determinante de la matriz wronskiana es el determinante wronskiano de la lista de expresiones.

Para hacer uso de esta función ejecutar `load("functs")`.

Ejemplo:

```
(%i1) load("functs")$
(%i2) wronskian([f(x), g(x)],x);
(%o2) matrix([f(x),g(x)],['diff(f(x),x,1),'diff(g(x),x,1)])
```

`tracematrix` (*M*) [Función]

Devuelve la traza (suma de los elementos de la diagonal) de la matriz *M*.

Para hacer uso de esta función ejecutar `load("functs")`.

`rational` (*z*) [Función]

Multiplica el numerador y denominador de *z* por el complejo conjugado del denominador, racionalizando así el denominador. Devuelve la expresión canónica racional (canonical rational expression, CRE) si el argumento *z* es de esta forma, en caso contrario devuelve una expresión en formato común.

Para hacer uso de esta función ejecutar `load("functs")`.

`nonzeroandfreeof` (*x*, *expr*) [Función]

Devuelve `true` si *expr* es diferente de cero y `freeof (x, expr)` devuelve `true`. En caso contrario devuelve `false`.

Para hacer uso de esta función ejecutar `load("functs")`.

`linear` (*expr*, *x*) [Función]

Si *expr* es una expresión de la forma `a*x + b`, siendo *a* no nulo y los argumentos *a* y *b* no contienen a *x*, `linear` devuelve una lista con tres ecuaciones, una por cada variable *b*, *a* y *x*. Si no se cumple la condición anterior, `linear` devuelve `false`.

Para hacer uso de esta función ejecutar `load("functs")`.

Ejemplo:

```
(%i1) load ("antid");
(%o1)        /usr/share/maxima/5.29.1/share/integration/antid.mac
(%i2) linear ((1 - w)*(1 - x)*z, z);
(%o2)  [bargumentb = 0, aargumenta = (w - 1) x - w + 1, xargumentx = z]
(%i3) linear (cos(u - v) + cos(u + v), u);
(%o3)                                false
```

`gcdivide` (*p*, *q*) [Función]

Si la variable opcional `takegcd` vale `true`, que es su valor por defecto, `gcdivide` divide los polinomios *p* y *q* por su máximo común divisor y devuelve el cociente de los resultados. `gcdivide` hace una llamada a la función `ezgcd` para dividir los polinomios por su máximo común divisor.

Si `takegcd` vale `false`, `gcdivide` devuelve el cociente *p*/*q*.

Para hacer uso de esta función ejecutar `load("functs")`.

Véanse también `ezgcd`, `gcd`, `gcdex` y `poly_gcd`.

Ejemplos:

```
(%i1) load("functs")$

(%i2) p1:6*x^3+19*x^2+19*x+6;
                        3       2
(%o2)                6 x  + 19 x  + 19 x + 6
(%i3) p2:6*x^5+13*x^4+12*x^3+13*x^2+6*x;
                    5       4       3       2
(%o3)          6 x  + 13 x  + 12 x  + 13 x  + 6 x
(%i4) gcdivide(p1, p2);
                              x + 1
(%o4)                         ------
                               3
                              x  + x
(%i5) takegcd:false;
(%o5)                          false
(%i6) gcdivide(p1, p2);
                         3       2
                      6 x  + 19 x  + 19 x + 6
(%o6)          ---------------------------------
                    5       4       3       2
                6 x  + 13 x  + 12 x  + 13 x  + 6 x
(%i7) ratsimp(%);
                              x + 1
(%o7)                         ------
                               3
                              x  + x
```

`arithmetic (a, d, n)` [Función]
Devuelve el *n*-ésimo término de la progresión aritmética `a, a + d, a + 2*d, ..., a + (n - 1)*d`.

Para hacer uso de esta función ejecutar `load("functs")`.

`geometric (a, r, n)` [Función]
Devuelve el *n*-ésimo término de la progresión geométrica `a, a*r, a*r^2, ..., a*r^(n - 1)`.

Para hacer uso de esta función ejecutar `load("functs")`.

`harmonic (a, b, c, n)` [Función]
Devuelve el *n*-ésimo término de la progresión armónica `a/b, a/(b + c), a/(b + 2*c), ..., a/(b + (n - 1)*c)`.

Para hacer uso de esta función ejecutar `load("functs")`.

`arithsum (a, d, n)` [Función]
Devuelve la suma de la progresión aritmética desde hasta el *n*-ésimo término.

Para hacer uso de esta función ejecutar `load("functs")`.

`geosum (a, r, n)` [Función]
Devuelve la suma de la sucesión geométrica hasta el *n*-ésimo término. Si *n* es infinito (`inf`) la suma será finita sólo si el valor absoluto de *r* es menor que 1.

Para hacer uso de esta función ejecutar `load("functs")`.

`gaussprob (x)` [Función]
Devuelve la función de densidad de probabilidad, normal `%e^(-x^2/2) / sqrt(2*%pi)`.

Para hacer uso de esta función ejecutar `load("functs")`.

`gd (x)` [Función]
Devuelve la función de Gudermann, `2*atan(%e^x)-%pi/2`.

Para hacer uso de esta función ejecutar `load("functs")`.

`agd (x)` [Función]
Devuelve la inversa de la función de Gudermann, `log (tan (%pi/4 + x/2))`.

Para hacer uso de esta función ejecutar `load("functs")`.

`vers (x)` [Función]
Devuelve `1 - cos (x)`.

Para hacer uso de esta función ejecutar `load("functs")`.

`covers (x)` [Función]
Devuelve `1 - sin (x)`.

Para hacer uso de esta función ejecutar `load("functs")`.

`exsec (x)` [Función]
Devuelve `sec (x) - 1`.

Para hacer uso de esta función ejecutar `load("functs")`.

`hav` (*x*) [Función]

Devuelve `(1 - cos(x))/2`.

Para hacer uso de esta función ejecutar `load("functs")`.

`combination` (*n*, *r*) [Función]

Calcula el número de combinaciones de *n* objetos tomados de *r* en *r*.

Para hacer uso de esta función ejecutar `load("functs")`.

`permutation` (*n*, *r*) [Función]

Calcula el número de permutaciones de *r*, seleccionados de un conjunto de *n*.

Para hacer uso de esta función ejecutar `load("functs")`.

72.5 Paquete ineq

El paquete `ineq` contiene reglas de simplificación para desigualdades

Una sesión de ejemplo:

```
(%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
```

Debe tenerse cuidado con el uso de paréntesis que incluyan desigualdades; si se escribe `(A > B) + (C = 5)` el resultado es `A + C > B + 5`, pero `A > B + C = 5` es un error sintáctico y `(A > B + C) = 5` es una cosa completamente diferente.

Ejecútese `disprule (all)` para ver la lista completa de las reglas definidas.

Maxima preguntará al usuario cuando desconozca el signo de una cantidad que multiplica a una desigualdad.

Los fallos más comunes son:

```
eq: a > b;
2*eq;
% - eq;
```

Otro problema es el producto de una desigualdad por cero. Si se escribe `x*some_inequality` y Maxima pregunta por el signo de `x` y se responde que vale `zero` (o `z`), el programa devuelve `x*some_inequality` sin hacer uso de la información de que `x` es 0. En tal caso se debería escribir `ev (%, x: 0)`, ya que la base de datos sólo será utilizada para fines comparativos y no para evaluar `x`.

El usuario puede apreciar que las respuestas son más lentas al cargarse este paquete, ya que el simplificador deberá examinar más reglas que cuando no se hace uso del paquete, por lo que puede ser conveniente borrar estas reglas cuando ya no se haga uso de ellas. Ejecútese `kill (rules)` para eliminar todas las reglas (incluidas las definidas por el usuario); también es posible eliminar parte de ellas o utilizar `remrule` sobre una reglas específica.

Nótese que si se carga este paquete después de haber definido otras reglas de igual nombre,se borrarán las antiguas. Las reglas de este paquete son: `*rule1`, ..., `*rule8`, `+rule1`, ..., `+rule18`, debiéndose encerrar entre comillas el nombre de la reglas para referenciarse a ellas, como en `remrule ("+", "+rule1")` para eliminar la primera regla sobre `"+"`, o `disprule ("*rule2")` para mostrar la definición de la segunda regla multiplicativa.

72.6 Paquete rducon

`reduce_consts` (*expr*) [Función]

Sustituye subexpresiones constantes de *expr* por átomos, guardando la definición de todos ellos en la lista de ecuaciones `const_eqns` y devolviendo el expresión *expr* ya modificada. Se consideran partes constantes de *expr* aquellas que devuelven `true`

cuando se les aplica la función `constantp`, por lo que antes de llamar a `reduce_consts` se debe ejecutar

```
declare ([objetos a los que se quiera dar la propiedad de ser constantes], constant)$
```

para crear la base de datos de las cantidades constantes presentes en la expresión.

Si se pretende generar código Fortran después de estos cálculos simbólicos, una de las primeras secciones del código debe ser el cálculo de las constantes. Para generar este segmento de código hacer

```
map ('fortran, const_eqns)$
```

Junto a `const_eqns`, otras variables que afectan a `reduce_consts` son:

`const_prefix` (Valor por defecto: `xx`) es la cadena de caracteres utilizada como prefijo para todos los símbolos generados por `reduce_consts` para representar subexpresiones constantes.

`const_counter` (Valor por defecto: 1) es el índice entero utilizado para generar los símbolos que representen a las subexpresiones constantes encontradas por `reduce_consts`.

La instrucción `load ("rducon")` carga esta función y `demo (rducon)` hace una demostración sobre su uso.

72.7 Paquete scifac

`gcfac` (*expr*) [Función]

Es una función de factorización que intenta aplicar la misma heurística que los humanos cuando tratan de hacer las expresiones más simples, limitándose a la factorización de monomios. En caso de sumas, `gcfac` hace lo siguiente:

1. Factoriza los enteros.
2. Factoriza las potencias mayores de los términos que aparecen como coeficientes, independientemente de su complejidad.
3. Utiliza (1) y (2) en la factorización de pares de términos adyacentes.
4. Aplica estas técnicas repetida y recursivamente hasta que la expresión deje de sufrir cambios.

En general, el apartado (3) no hace una factorización óptima debido a la naturaleza combinatoria y compleja de encontrar cuál de todas las ordenaciones posibles de los pares da lugar a la expresión más compacta.

La instrucción `load ("scifac")` carga esta función y `demo (scifac)` hace una demostración sobre su uso.

72.8 Paquete sqdnst

`sqrtdenest` (*expr*) [Función]

Reduce expresiones en las que se encuentren raíces cuadradas anidadas, siempre que sea posible

Ejemplo:

```
(%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
```

A veces conviene aplicar `sqrtdenest` más de una vez, como en el caso `(19601-13860 sqrt(2))^(7/4)`.

La sentencia `load ("sqdnst")` carga esta función.

73 solve_rec

73.1 Introducción a solve_rec

El paquete `solve_rec` resuelve expresiones recurrentes lineales con coeficientes polinomiales.

Ejecútese `demo(solve_rec);` para ver una demostración sobre la utilización de este paquete.

Ejemplo:

```
(%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)
```

73.2 Funciones y variables para solve_rec

`reduce_order` (*rec*, *sol*, *var*) [Función]

Reduce el orden de la expresión recurrente lineal *rec* cuando se conoce una solución particular *sol*. La recurrencia reducida puede utilizarse para obterner más soluciones.

Ejemplo:

```
(%i3) rec: x[n+2] = x[n+1] + x[n]/n;
                                        x
                                         n
(%o3)               x      = x      + --
                     n + 2    n + 1   n
(%i4) solve_rec(rec, x[n]);
WARNING: found some hypergeometrical solutions!
(%o4)                    x  = %k  n
                          n     1
(%i5) reduce_order(rec, n, x[n]);
(%t5)                    x  = n %z
                          n       n

                            n - 1
                            ====
                            \
(%t6)                %z  =   >     %u
                       n    /        %j
                            ====
                            %j = 0

(%o6)             (- n - 2) %u     - %u
                                n + 1     n
(%i6) solve_rec((n+2)*%u[n+1] + %u[n], %u[n]);
```

```
                                          n
                                      %k  (- 1)
                                        1
(%o6)                      %u  = ----------
                             n    (n + 1)!
```

So the general solution is

```
                n - 1
                ====         j
                \       (- 1)
          %k  n  >     -------- + %k  n
            2   /     (j + 1)!     1
                ====
                j = 0
```

`simplify_products` [Variable opcional]

Valor por defecto: `true`

Si `simplify_products` vale `true`, `solve_rec` intentará simplificar los productos del resultado.

Véase también `solve_rec`.

`simplify_sum` (*expr*) [Función]

Intenta reducir todas las sumas que aparecen en *expr* a una forma cerrada.

Para utilizar esta función cárguese previamente el paquete `simplify_sum` ejecutando la instrucción `load("simplify_sum")`.

Ejemplo:

```
(%i1) load("simplify_sum")$
(%i2) sum(binomial(n+k,k)/2^k,k,1,n)+sum(binomial(2*n,2*k),k,1,n);
       n                          n
      ====                       ====
      \     binomial(n + k, k)   \
(%o2)  >    ------------------ +  >    binomial(2 n, 2 k)
      /              k           /
      ====          2            ====
      k = 1                      k = 1
(%i3) simplify_sum(%);

                          2 n - 1    n
(%o3)                    2        + 2  - 2
```

`solve_rec` (*eqn*, *var*, [*init*]) [Función]

Obtiene las soluciones hipergeométricas de la expresión recurrente *eqn* con coeficientes lineales en la variable *var*. Los argumentos opcionales *init* son condiciones iniciales.

La función `solve_rec` puede resolver expresiones recurrentes con coeficientes constantes, encuentra soluciones hipergeométricas de expresiones recurrentes lineales homogéneas con coeficientes polinomiales, obtiene soluciones racionales de expresiones

recurrentes lineales con coeficientes lineales y resuelve también expresiones recurrentes de Ricatti.

Nótese que el tiempo de ejecución del algoritmo para encontrar soluciones hipergeométricas es exponencial respecto del grado del coeficiente principal.

Para hacer uso de esta función ejecútese previamente `load("solve_rec");`.

Ejemplo de recurrencia lineal con coeficientes constantes:

```
(%i2) solve_rec(a[n]=a[n-1]+a[n-2]+n/2^n, a[n]);
                          n          n
           (sqrt(5) - 1)  %k  (- 1)
                            1           n
(%o2) a  = ------------------------ - ----
       n               n                 n
                      2                5 2
                                              n
                                    (sqrt(5) + 1)  %k
                                                     2    2
                                  + ------------------ - ----
                                             n              n
                                            2            5 2
```

Ejemplo de recurrencia lineal con coeficientes polinomiales:

```
(%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
```

Ejemplo de recurrencia de Ricatti:

```
(%i2) x*y[x+1]*y[x] - y[x+1]/(x+2) + y[x]/(x-1) = 0;
                             y         y
                              x + 1     x
(%o2)         x y  y      - ------ + ----- = 0
                 x  x + 1   x + 2    x - 1
(%i3) solve_rec(%, y[x], y[3]=5)$
(%i4) ratsimp(minfactorial(factcomb(%)));
                                  3
                             30 x  - 30 x
(%o4) y  = - ------------------------------------------------
       x        6      5       4       3       2
             5 x  - 3 x  - 25 x  + 15 x  + 20 x  - 12 x - 1584
```

Véanse también `solve_rec_rat`, `simplify_products` y `product_use_gamma`.

solve_rec_rat (*eqn*, *var*, [*init*]) [Función]

Calcula las soluciones racionales de las expresiones recurrentes lineales. Véase solve_rec para la descripción de sus argumentos.

Para hacer uso de esta función ejecútese previamente load("solve_rec");.

Ejemplo:

```
(%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)
```

Véase también solve_rec.

product_use_gamma [Variable opcional]

Valor por defecto:true

Si product_use_gamma vale true, solve_rec introduce la función gamma en la expresión del resultado cuando se simplifican productos.

Véanse también simplify_products y solve_rec.

summand_to_rec (*summand*, *k*, *n*) [Función]

Devuelve la expresión recurrente que satisface la suma

```
  inf
  ====
  \
   >     sumando
  /
  ====
 k = minf
```

donde el sumando es hipergeométrico en k y n.

Para hacer uso de esta función deben cargarse previamente los paquetes zeilberger y solve_rec mediante la ejecución de las sentencias load("solve_rec") y load("zeilberger").

```
(%i17) load("zeilberger")$
(%i18) summand: binom(3*k+1,k)*binom(3*(n-k),n-k)/(3*k+1)$
(%i19) summand_to_rec(summand, k, n);
Dependent equations eliminated:  (3 2)
(%o19) - 4 (n + 2) (2 n + 3) (2 n + 5) sm
                                          n + 2
                     2
 + 12 (2 n + 3) (9 n  + 27 n + 22) sm
                                     n + 1
```

```
 - 81 (n + 1) (3 n + 2) (3 n + 4) sm
                                    n
(%i21) sum(''summand, k, 0, n), n=0;
(%o21)                         1
(%i22) sum(''summand, k, 0, n), n=1;
(%o22)                         4
(%i23) product_use_gamma: false$
(%i24) solve_rec(%o19, sm[n], sm[0]=1, sm[1]=4);
              n - 1                n - 1
              /===\                /===\
               ! !                  ! !                  n
              ( ! !   (3 %j + 2)) ( ! !   (3 %j + 4)) 3
               ! !                  ! !
              %j = 0               %j = 0
(%o24) sm  = ----------------------------------------
         n            n - 1
                      /===\
                       ! !                 n
                      ( ! !   (2 %j + 3)) 2  n!
                       ! !
                      %j = 0
```

74 stats

74.1 Introducción a stats

El paquete `stats` contiene procedimientos clásicos sobre inferencia estadística y contraste de hipótesis.

Todas estas funciones devuelven un objeto Maxima de tipo `inference_result`, el cual contiene los resultados necesarios para hacer inferencias sobre la población y toma de decisiones.

La variable global `stats_numer` controla si los resultados deben darse en formato decimal o simbólico y racional; su valor por defecto es `true`, por lo que el formato de salida es decimal.

El paquete `descriptive` contiene algunas utilidades para manipular estructuras de datos (listas y matrices); por ejemplo para extraer submuestras. También contiene algunos ejemplos sobre cómo utilizar el paquete `numericalio` para leer datos de ficheros en texto plano. Véanse `descriptive` y `numericalio` para más detalles.

El paquete `stats` carga en memoria los paquetes `descriptive`, `distrib` y `inference_result`.

Para comentarios, errores o sugerencias, contáctese con el autor en

'mario ARROBA edu PUNTO xunta PUNTO es'.

74.2 Funciones y variables para inference_result

`inference_result` (*title*, *values*, *numbers*) [Función]

Construye un objeto `inference_result` del tipo devuelto por las funciones estadísticas. El argumento *title* es una cadena con el nombre del procedimiento; *values* es una lista con elementos de la forma `symbol = value` y *numbers* es una lista con enteros positivos desde uno hasta `length(values)`, que indican qué valores serán mostrados por defecto.

Ejemplo:

Este es un ejemplo que muestra los resultados asociados a un rectángulo. El título de este objeto es la cadena `"Rectangle"`, el cual almacena cinco resultados, a saber, `'base`, `'height`, `'diagonal`, `'area` y `'perimeter`, pero sólo muestra el primero, segundo, quinto y cuarto. El resultado `'diagonal` también se almacena en este objeto, pero no se muestra por defecto; para tener acceso a este valor, hágase uso de la función `take_inference`.

```
(%i1) load("inference_result")$
(%i2) b: 3$ h: 2$
(%i3) inference_result("Rectangle",
                       ['base=b,
                        'height=h,
                        'diagonal=sqrt(b^2+h^2),
                        'area=b*h,
                        'perimeter=2*(b+h)],
                       [1,2,5,4] );
```

```
                              |   Rectangle
                              |
                              |    base = 3
                              |
(%o3)                         |   height = 2
                              |
                              | perimeter = 10
                              |
                              |    area = 6
(%i4) take_inference('diagonal,%);
(%o4)                          sqrt(13)
```

Véase también `take_inference`.

`inferencep` (*obj*) [Función]

Devuelve `true` o `false`, dependiendo de que *obj* sea un objeto de tipo `inference_result` o no.

`items_inference` (*obj*) [Función]

Devuelve una lista con los nombres de los elementos almacenados en *obj*, el cual debe ser un objeto de tipo `inference_result`.

Ejemplo:

El objeto `inference_result` almacena dos valores, cuyos nombres son `'pi` y `'e`, pero sólo se muestra el segundo. La función `items_inference` devuelve los nombres de todos los elementos almacenados, independientemente de que sean mostrados o no.

```
(%i1) load("inference_result")$
(%i2) inference_result("Hi", ['pi=%pi,'e=%e],[2]);
                              |   Hi
(%o2)                         |
                              | e = %e
(%i3) items_inference(%);
(%o3)                          [pi, e]
```

`take_inference` (*n*, *obj*) [Función]
`take_inference` (*name*, *obj*) [Función]
`take_inference` (*list*, *obj*) [Función]

Si *n* es un entero positivo, devuelve el *n*-ésimo valor almacenado en *obj*; si el símbolo *name* es el nombre de uno de los elementos almacenados, también devuelve su valor. Si el primer elemento es una lista de números y/o símbolos, la función `take_inference` devuelve una lista con los resultados correspondientes.

Ejemplo:

Dado un objeto `inference_result`, la función `take_inference` es invocada para extraer cierta información almacenada en él.

```
(%i1) load("inference_result")$
(%i2) b: 3$ h: 2$
(%i3) sol: inference_result("Rectangle",
                            ['base=b,
```

```
                                    'height=h,
                                    'diagonal=sqrt(b^2+h^2),
                                    'area=b*h,
                                    'perimeter=2*(b+h)],
                                   [1,2,5,4] );
                              |   Rectangle
                              |
                              |    base = 3
                              |
(%o3)                         |   height = 2
                              |
                              | perimeter = 10
                              |
                              |    area = 6
(%i4) take_inference('base,sol);
(%o4)                              3
(%i5) take_inference(5,sol);
(%o5)                             10
(%i6) take_inference([1,'diagonal],sol);
(%o6)                         [3, sqrt(13)]
(%i7) take_inference(items_inference(sol),sol);
(%o7)                  [3, 2, sqrt(13), 6, 10]
```

Véanse también `inference_result` y `take_inference`.

74.3 Funciones y variables para stats

`stats_numer` [Variable opcional]

Valor por defecto: `true`

Cuando `stats_numer` vale `true`, las funciones de inferencia estadística devuelven sus resultados en formato decimal de coma flotante. Cuando vale `false`, los resultados se devuelven en formato simbólico y racional.

`test_mean` (*x*) [Función]
`test_mean` (*x*, *options* ...) [Función]

Es el test *t* de la media. El argumento *x* es una lista o matriz columna con los datos de una muestra unidimensional. También realiza el test asintótico basado en el *Teorema Central del límite* si se le asigna a la opción `'asymptotic` el valor `true`.

Opciones:

- `'mean`, valor por defecto `0`, es el valor de la media a contrastar.
- `'alternative`, valor por defecto `'twosided`, es la hipótesis alternativa; valores válidos son: `'twosided`, `'greater` y `'less`.
- `'dev`, valor por defecto `'unknown`, este es el valor de la desviación típica cuando se conoce; valores válidos son: `'unknown` o una expresión con valor positivo.
- `'conflevel`, valor por defecto `95/100`, nivel de confianza para el intervalo de confianza; debe ser una expresión que tome un valor en el intervalo (0,1).

- `'asymptotic`, valor por defecto `false`, indica si debe realizar el test exacto basado en la t de Student, o el asintótico basado en el *Teorema Central del límite*; valores válidos son `true` y `false`.

El resultado devuelto por la función `test_mean` es un objeto `inference_result` con los siguientes apartados:

1. `'mean_estimate`: la media muestral.
2. `'conf_level`: nivel de confianza seleccionado por el usuario.
3. `'conf_interval`: intervalo de confianza para la media poblacional.
4. `'method`: procedimiento de inferencia.
5. `'hypotheses`: hipótesis nula y alternativa a ser contrastada.
6. `'statistic`: valor del estadístico de contraste utilizado para probar la hipótesis.
7. `'distribution`: distribución del estadístico de contraste, junto con su(s) parámetro(s).
8. `'p_value`: p-valor del test.

Ejemplos:

Realiza el contraste exacto t con varianza desconocida. La hipótesis nula es H_0 : $mean = 50$, frente a la alternativa unilátera H_1 : $mean < 50$; de acuerdo con los resultados, no hay evidencia para rechazar H_0, pues el p-valor es muy grande.

```
(%i1) load("stats")$
(%i2) data: [78,64,35,45,45,75,43,74,42,42]$
(%i3) test_mean(data,'conflevel=0.9,'alternative='less,'mean=50);
          |                  MEAN TEST
          |
          |             mean_estimate = 54.3
          |
          |               conf_level = 0.9
          |
          | conf_interval = [minf, 61.51314273502712]
          |
(%o3)     |  method = Exact t-test. Unknown variance.
          |
          | hypotheses = H0: mean = 50 , H1: mean < 50
          |
          |       statistic = .8244705235071678
          |
          |       distribution = [student_t, 9]
          |
          |        p_value = .7845100411786889
```

En esta ocasión Maxima realiza un test asintótico. La hipótesis nula es H_0 : $equal(mean, 50)$ frente a la alternativa bilátera H_1 : $notequal(mean, 50)$; de acuerdo con los resultados, H_0 debe rechazarse en favor de la alternativa H_1, pues el p-valor es muy pequeño. Nótese que, tal como indica la componente `Method`, este procedimiento sólo puede aplicarse en muestras grandes.

```
(%i1) load("stats")$
```

```
(%i2) test_mean([36,118,52,87,35,256,56,178,57,57,89,34,25,98,35,
                 98,41,45,198,54,79,63,35,45,44,75,42,75,45,45,
                 45,51,123,54,151],
                 'asymptotic=true,'mean=50);
                |                       MEAN TEST
                |
                |           mean_estimate = 74.88571428571429
                |
                |                   conf_level = 0.95
                |
                | conf_interval = [57.72848600856194, 92.04294256286663]
                |
(%o2)           |    method = Large sample z-test. Unknown variance.
                |
                |       hypotheses = H0: mean = 50 , H1: mean # 50
                |
                |             statistic = 2.842831192874313
                |
                |             distribution = [normal, 0, 1]
                |
                |             p_value = .004471474652002261
```

`test_means_difference` (*x1*, *x2*) [Función]
`test_means_difference` (*x1*, *x2*, *options ...*) [Función]

Este es el test t para la diferencia de medias con muestras. Los argumentos *x1* y *x2* son listas o matrices columna que contienen dos muestras independientes. En caso de varianzas diferentes y desconocidas (véanse las opciones `'dev1`, `'dev2` y `'varequal` más abajo) los grados de libertad se calculan mediante la aproximación de Welch. También realiza el test asintótico basado en el *Teorema Central del límite* si se le asigna a la opción `'asymptotic` el valor `true`.

Opciones:

-
- `'alternative`, valor por defecto `'twosided`, es la hipótesis alternativa; valores válidos son: `'twosided`, `'greater` y `'less`.
- `'dev1`, valor por defecto `'unknown`, es el valor de la desviación típica de la muestra *x1* cuando se conoce; valores válidos son: `'unknown` o una expresión positiva.
- `'dev2`, valor por defecto `'unknown`, es el valor de la desviación típica de la muestra *x2* cuando se conoce; valores válidos son: `'unknown` o una expresión positiva.
- `'varequal`, valor por defecto `false`, indica si las varianzas deben considerarse iguales o no; esta opción sólo toma efecto cuando `'dev1` y/o `'dev2` tienen el valor `'unknown`.
- `'conflevel`, valor por defecto `95/100`, nivel de confianza para el intervalo de confianza; debe ser una expresión que tome un valor en el intervalo (0,1).
- `'asymptotic`, valor por defecto `false`, indica si debe realizar el test exacto basado en la t de Student, o el asintótico basado en el *Teorema Central del límite*; valores válidos son `true` y `false`.

El resultado devuelto por la función `test_means_difference` es un objeto `inference_result` con los siguientes apartados:

1. `'diff_estimate`: el estimador de la diferencia de medias.
2. `'conf_level`: nivel de confianza seleccionado por el usuario.
3. `'conf_interval`: intervalo de confianza para la diferencia de medias.
4. `'method`: procedimiento de inferencia.
5. `'hypotheses`: hipótesis nula y alternativa a ser contrastada.
6. `'statistic`: valor del estadístico de contraste utilizado para probar la hipótesis.
7. `'distribution`: distribución del estadístico de contraste, junto con su(s) parámetro(s).
8. `'p_value`: p-valor del test.

Ejemplos:

La igualdad de medias se contrasta con dos pequeñas muestras x y y, contra la alternativa $H_1 : m_1 > m_2$, siendo m_1 y m_2 las medias poblacionales; las varianzas son desconocidas y se supone que diferentes.

```
(%i1) load("stats")$
(%i2) x: [20.4,62.5,61.3,44.2,11.1,23.7]$
(%i3) y: [1.2,6.9,38.7,20.4,17.2]$
(%i4) test_means_difference(x,y,'alternative='greater);
             |              DIFFERENCE OF MEANS TEST
             |
             |         diff_estimate = 20.31999999999999
             |
             |                 conf_level = 0.95
             |
             |    conf_interval = [- .04597417812882298, inf]
             |
(%o4)        |        method = Exact t-test. Welch approx.
             |
             | hypotheses = H0: mean1 = mean2 , H1: mean1 > mean2
             |
             |           statistic = 1.838004300728477
             |
             |    distribution = [student_t, 8.62758740184604]
             |
             |            p_value = .05032746527991905
```

El mismo test que antes, pero ahora se suponen las varianzas iguales.

```
(%i1) load("stats")$
(%i2) x: [20.4,62.5,61.3,44.2,11.1,23.7]$
(%i3) y: matrix([1.2],[6.9],[38.7],[20.4],[17.2])$
(%i4) test_means_difference(x,y,
                            'alternative='greater,
                            'varequal=true);
             |              DIFFERENCE OF MEANS TEST
```

```
                    |
                    |         diff_estimate = 20.31999999999999
                    |
                    |                conf_level = 0.95
                    |
                    |     conf_interval = [- .7722627696897568, inf]
                    |
          (%o4)     |   method = Exact t-test. Unknown equal variances
                    |
                    | hypotheses = H0: mean1 = mean2 , H1: mean1 > mean2
                    |
                    |           statistic = 1.765996124515009
                    |
                    |           distribution = [student_t, 9]
                    |
                    |            p_value = .05560320992529344
```

`test_variance` (*x*) [Función]
`test_variance` (*x*, *options* ...) [Función]

Este es el test *chi^2* de la varianza. El argumento *x* es una lista o matriz columna con los datos de una muestra unidimensional extraída de una población normal.

Opciones:

- `'mean`, valor por defecto `'unknown`, es la media de la población, si se conoce.
- `'alternative`, valor por defecto `'twosided`, es la hipótesis alternativa; valores válidos son: `'twosided`, `'greater` y `'less`.
- `'variance`, valor por defecto `1`, este es el valor (positivo) de la varianza a contrastar.
- `'conflevel`, valor por defecto `95/100`, nivel de confianza para el intervalo de confianza; debe ser una expresión que tome un valor en el intervalo (0,1).

El resultado devuelto por la función `test_variance` es un objeto `inference_result` con los siguientes apartados:

1. `'var_estimate`: la varianza muestral.
2. `'conf_level`: nivel de confianza seleccionado por el usuario.
3. `'conf_interval`: intervalo de confianza para la varianza poblacional.
4. `'method`: procedimiento de inferencia.
5. `'hypotheses`: hipótesis nula y alternativa a ser contrastada.
6. `'statistic`: valor del estadístico de contraste utilizado para probar la hipótesis.
7. `'distribution`: distribución del estadístico de contraste, junto con su parámetro.
8. `'p_value`: *p*-valor del test.

Ejemplos:

Se contrasta si la varianza de una población de media desconocida es igual o mayor que 200.

```
(%i1) load("stats")$
```

```
(%i2) x: [203,229,215,220,223,233,208,228,209]$
(%i3) test_variance(x,'alternative='greater,'variance=200);
                |                   VARIANCE TEST
                |
                |              var_estimate = 110.75
                |
                |                conf_level = 0.95
                |
                |     conf_interval = [57.13433376937479, inf]
                |
(%o3)           | method = Variance Chi-square test. Unknown mean.
                |
                |    hypotheses = H0: var = 200 , H1: var > 200
                |
                |                 statistic = 4.43
                |
                |             distribution = [chi2, 8]
                |
                |           p_value = .8163948512777689
```

`test_variance_ratio` (*x1*, *x2*) [Función]
`test_variance_ratio` (*x1*, *x2*, *options* ...) [Función]

Este es el test F del cociente de las varianzas para dos poblaciones normales. Los argumentos *x1* y *x2* son listas o matrices columna que contienen los datos de dos muestras independientes.

Opciones:

- `'alternative`, valor por defecto `'twosided`, es la hipótesis alternativa; valores válidos son: `'twosided`, `'greater` y `'less`.
- `'mean1`, valor por defecto `'unknown`, es la media de la población de la que procede *x1* cuando se conoce.
- `'mean2`, valor por defecto `'unknown`, es la media de la población de la que procede *x2* cuando se conoce.
- `'conflevel`, valor por defecto `95/100`, nivel de confianza para el intervalo de confianza del cociente; debe ser una expresión que tome un valor en el intervalo (0,1).

El resultado devuelto por la función `test_variance_ratio` es un objeto `inference_result` con los siguientes resultados

1. `'ratio_estimate`: el cociente de varianzas muestral.
2. `'conf_level`: nivel de confianza seleccionado por el usuario.
3. `'conf_interval`: intervalo de confianza para el cociente de varianzas.
4. `'method`: procedimiento de inferencia.
5. `'hypotheses`: hipótesis nula y alternativa a ser contrastada.
6. `'statistic`: valor del estadístico de contraste utilizado para probar la hipótesis.
7. `'distribution`: distribución del estadístico de contraste, junto con sus parámetros.

8. `'p_value`: p-valor del test.

Ejemplos:

Se contrasta la igualdad de varianzas de dos poblaciones normales frente a la alternativa de que la primera es mayor que la segunda.

```
(%i1) load("stats")$
(%i2) x: [20.4,62.5,61.3,44.2,11.1,23.7]$
(%i3) y: [1.2,6.9,38.7,20.4,17.2]$
(%i4) test_variance_ratio(x,y,'alternative='greater);
                |              VARIANCE RATIO TEST
                |
                |       ratio_estimate = 2.316933391522034
                |
                |               conf_level = 0.95
                |
                |    conf_interval = [.3703504689507268, inf]
                |
(%o4)           | method = Variance ratio F-test. Unknown means.
                |
                | hypotheses = H0: var1 = var2 , H1: var1 > var2
                |
                |         statistic = 2.316933391522034
                |
                |            distribution = [f, 5, 4]
                |
                |          p_value = .2179269692254457
```

`test_proportion` (*x*, *n*) [Función]
`test_proportion` (*x*, *n*, *options* ...) [Función]

Inferencias sobre una proporción. El argumento x es el número de éxitos observados en n pruebas de Bernoulli con probabilidad desconocida.

Opciones:

- `'proportion`, valor por defecto `1/2`, es el valor de la probabilidad a contrastar.
- `'alternative`, valor por defecto `'twosided`, es la hipótesis alternativa; valores válidos son: `'twosided`, `'greater` y `'less`.
- `'conflevel`, valor por defecto `95/100`, nivel de confianza para el intervalo de confianza; debe ser una expresión que tome un valor en el intervalo (0,1).
- `'asymptotic`, valor por defecto `false`, indica si debe realizar el test exacto basado en la binomial, o el asintótico basado en el *Teorema Central del límite*; valores válidos son `true` y `false`.
- `'correct`, valor por defecto `true`, indica si se aplica o no la corrección de Yates.

El resultado devuelto por la función `test_proportion` es un objeto `inference_result` con los siguientes apartados:

1. `'sample_proportion`: proporción muestral.
2. `'conf_level`: nivel de confianza seleccionado.

3. 'conf_interval: intervalo de confianza de Wilson para la proporción.
4. 'method: procedimiento de inferencia.
5. 'hypotheses: hipótesis nula y alternativa a ser contrastada.
6. 'statistic: valor del estadístico de contraste utilizado para probar la hipótesis.
7. 'distribution: distribución del estadístico de contraste, junto con sus parámetros.
8. 'p_value: p-valor del test.

Ejemplos:

Realiza un contraste exacto. La hipótesis nula es $H_0 : p = 1/2$ y la alternativa unilátera es $H_1 : p < 1/2$.

```
(%i1) load("stats")$
(%i2) test_proportion(45, 103, alternative = less);
          |            PROPORTION TEST
          |
          | sample_proportion = .4368932038834951
          |
          |           conf_level = 0.95
          |
          | conf_interval = [0, 0.522714149150231]
          |
(%o2)     |     method = Exact binomial test.
          |
          | hypotheses = H0: p = 0.5 , H1: p < 0.5
          |
          |             statistic = 45
          |
          |  distribution = [binomial, 103, 0.5]
          |
          |      p_value = .1184509388901454
```

Un contraste asintótico bilátero. El nivel de confianza es 99/100.

```
(%i1) load("stats")$
(%i2) fpprintprec:7$
(%i3) test_proportion(45, 103,
                  conflevel = 99/100, asymptotic=true);
       |                PROPORTION TEST
       |
       |           sample_proportion = .43689
       |
       |                conf_level = 0.99
       |
       |        conf_interval = [.31422, .56749]
       |
(%o3)  | method = Asympthotic test with Yates correction.
       |
```

```
|     hypotheses = H0: p = 0.5 , H1: p # 0.5
|
|               statistic = .43689
|
|      distribution = [normal, 0.5, .048872]
|
|                p_value = .19662
```

`test_proportions_difference` (*x1*, *n1*, *x2*, *n2*) [Función]
`test_proportions_difference` (*x1*, *n1*, *x2*, *n2*, *options* ...) [Función]

Inferencias sobre la diferencia de dos proporciones. El argumento *x1* es el número de éxitos en *n1* experimentos de Bernoulli en la primera población y *x2* y *n2* son los valores correspondientes para la segunda población. Las muestras son independientes y el contraste es asintótico.

Opciones:

- `'alternative`, valor por defecto `'twosided`, es la hipótesis alternativa; valores válidos son:: `'twosided` (`p1 # p2`), `'greater` (`p1 > p2`) and `'less` (`p1 < p2`).
- `'conflevel`, valor por defecto `95/100`, nivel de confianza para el intervalo de confianza; debe ser una expresión que tome un valor en el intervalo (0,1).
- `'correct`, valor por defecto `true`, indica si se aplica o no la corrección de Yates.

El resultado devuelto por la función `test_proportions_difference` es un objeto `inference_result` con los siguientes apartados:

1. `'proportions`: lista con las dos proporciones muestrales.
2. `'conf_level`: nivel de confianza seleccionado.
3. `'conf_interval`: intervalo de confianza para la diferencia de proporciones `p1 - p2`.
4. `'method`: procedimiento de inferencia y mensaje de aviso en caso de que alguno de los tamaños muestrales sea menor de 10.
5. `'hypotheses`: hipótesis nula y alternativa a ser contrastada.
6. `'statistic`: valor del estadístico de contraste utilizado para probar la hipótesis.
7. `'distribution`: distribución del estadístico de contraste, junto con sus parámetros.
8. `'p_value`: *p*-valor del test.

Ejemplos:

Una máquina produce 10 piezas defectuosas en un lote de 250. Después de ciertas tareas de mantenimiento, produce 4 piezas defectuosas de un lote de 150. A fin de saber si la tarea de mantenimiento produjo alguna mejora, se contrasta la hipótesis nula `H0:p1=p2` contra la alternativa `H0:p1>p2`, donde `p1` y `p2` son las probabilidades de que un artículo producido por la máquina sea defectuoso, antes y después de la reparación. De acuerdo con el p valor, no hay evidencia suficiente para aceptar la alternativa.

```
(%i1) load("stats")$
(%i2) fpprintprec:7$
```

```
(%i3) test_proportions_difference(10, 250, 4, 150,
                                  alternative = greater);
      |       DIFFERENCE OF PROPORTIONS TEST
      |
      |       proportions = [0.04, .02666667]
      |
      |              conf_level = 0.95
      |
      |      conf_interval = [- .02172761, 1]
      |
(%o3) | method = Asymptotic test. Yates correction.
      |
      |   hypotheses = H0: p1 = p2 , H1: p1 > p2
      |
      |            statistic = .01333333
      |
      |    distribution = [normal, 0, .01898069]
      |
      |             p_value = .2411936
```

Desviación típica exacta de la distribución normal asintótica con datos desconocidos.

```
(%i1) load("stats")$
(%i2) stats_numer: false$
(%i3) sol: test_proportions_difference(x1,n1,x2,n2)$
(%i4) last(take_inference('distribution,sol));
               1    1                  x2 + x1
             (-- + --) (x2 + x1) (1 - -------)
               n2   n1                 n2 + n1
(%o4)    sqrt(---------------------------------)
                           n2 + n1
```

`test_sign` (*x*) [Función]

`test_sign` (*x*, *options* ...) [Función]

Este es el test no paramétrico de los signos para contrastes sobre la mediana de una población continua. El argumento *x* es una lista o matriz columna que contiene los datos de una muestra unidimensional.

Opciones:

- `'alternative`, valor por defecto `'twosided`, es la hipótesis alternativa; valores válidos son: `'twosided`, `'greater` y `'less`.
- `'median`, valor por defecto `0`, es el valor de la mediana a contrastar.

El resultado devuelto por la función `test_sign` es un objeto `inference_result` con los siguientes apartados:

1. `'med_estimate`: la mediana muestral.
2. `'method`: procedimiento de inferencia.
3. `'hypotheses`: hipótesis nula y alternativa a ser contrastada.
4. `'statistic`: valor del estadístico de contraste utilizado para probar la hipótesis.

5. `'distribution`: distribución del estadístico de contraste, junto con sus parámetros.
6. `'p_value`: p-valor del test.

Ejemplos:

Contrasta si la mediana de la población de la que se ha extraido la muestra es 6, frente a la alternativa $H_1 : median > 6$.

```
(%i1) load("stats")$
(%i2) x: [2,0.1,7,1.8,4,2.3,5.6,7.4,5.1,6.1,6]$
(%i3) test_sign(x,'median=6,'alternative='greater);
                |                   SIGN TEST
                |
                |              med_estimate = 5.1
                |
                |      method = Non parametric sign test.
                |
(%o3)           | hypotheses = H0: median = 6 , H1: median > 6
                |
                |                statistic = 7
                |
                |      distribution = [binomial, 10, 0.5]
                |
                |         p_value = .05468749999999989
```

`test_signed_rank` (*x*) [Función]
`test_signed_rank` (*x*, *options* ...) [Función]

Este el test de los rangos signados de Wilcoxon para hacer inferencias sobre la mediana de una población continua. El argumento *x* es una lista o matriz columna que contiene los datos de una muestra unidimensional. Realiza la aproximación normal si el tamaño muestral es mayor que 20, o si en la muestra aparece algún cero o hay empates.

Véanse también `pdf_rank_test` y `cdf_rank_test`.

Opciones:

- `'median`, valor por defecto `0`, es el valor de la mediana a ser contrastado.
- `'alternative`, valor por defecto `'twosided`, es la hipótesis alternativa; valores válidos son: `'twosided`, `'greater` y `'less`.

El resultado devuelto por la función `test_signed_rank` es un objeto `inference_result` con los siguientes apartados:

1. `'med_estimate`: la mediana muestral.
2. `'method`: procedimiento de inferencia.
3. `'hypotheses`: hipótesis nula y alternativa a ser contrastada.
4. `'statistic`: valor del estadístico de contraste utilizado para probar la hipótesis.
5. `'distribution`: distribución del estadístico de contraste, junto con su(s) parámetro(s).
6. `'p_value`: p-valor del test.

Ejemplos:

Contrasta la hipótesis nula $H_0 : median = 15$ frente a la alternativa $H_1 : median > 15$. Este test es exacto, puesto que no hay empates.

```
(%i1) load("stats")$
(%i2) x: [17.1,15.9,13.7,13.4,15.5,17.6]$
(%i3) test_signed_rank(x,median=15,alternative=greater);
                   |             SIGNED RANK TEST
                   |
                   |           med_estimate = 15.7
                   |
                   |           method = Exact test
                   |
(%o3)              | hypotheses = H0: med = 15 , H1: med > 15
                   |
                   |              statistic = 14
                   |
                   |     distribution = [signed_rank, 6]
                   |
                   |             p_value = 0.28125
```

Contrasta la hipótesis nula $H_0 : equal(median, 2.5)$ frente a la alternativa $H_1 : notequal(median, 2.5)$. Este es un test asintótico, debido a la presencia de empates.

```
(%i1) load("stats")$
(%i2) y:[1.9,2.3,2.6,1.9,1.6,3.3,4.2,4,2.4,2.9,1.5,3,2.9,4.2,3.1]$
(%i3) test_signed_rank(y,median=2.5);
                |                 SIGNED RANK TEST
                |
                |                med_estimate = 2.9
                |
                |          method = Asymptotic test. Ties
                |
(%o3)           |    hypotheses = H0: med = 2.5 , H1: med # 2.5
                |
                |                 statistic = 76.5
                |
                | distribution = [normal, 60.5, 17.58195097251724]
                |
                |           p_value = .3628097734643669
```

`test_rank_sum` (*x1*, *x2*) [Función]

`test_rank_sum` (*x1*, *x2*, *option*) [Función]

Este es el test de Wilcoxon-Mann-Whitney para comparar las medianas de dos poblaciones continuas. Los dos primeros argumentos *x1* y *x2* son listas o matrices columna con los datos de dos muestras independientes. Realiza la aproximación normal si alguna de las muestras tiene tamaño mayor que 10, o si hay empates.

Opción:

- `'alternative`, valor por defecto `'twosided`, es la hipótesis alternativa; valores válidos son: `'twosided`, `'greater` y `'less`.

El resultado devuelto por la función `test_rank_sum` es un objeto `inference_result` con los siguientes apartados:

1. `'method`: procedimiento de inferencia.
2. `'hypotheses`: hipótesis nula y alternativa a ser contrastada.
3. `'statistic`: valor del estadístico de contraste utilizado para probar la hipótesis.
4. `'distribution`: distribución del estadístico de contraste, junto con sus parámetros.
5. `'p_value`: p-valor del test.

Ejemplos:

Contrasta si dos poblaciones tiene medianas similares. Al ser los tamaños muestrales pequeños, se realiza el test exacto.

```
(%i1) load("stats")$
(%i2) x:[12,15,17,38,42,10,23,35,28]$
(%i3) y:[21,18,25,14,52,65,40,43]$
(%i4) test_rank_sum(x,y);
                 |                  RANK SUM TEST
                 |
                 |               method = Exact test
                 |
                 | hypotheses = H0: med1 = med2 , H1: med1 # med2
(%o4)            |
                 |                  statistic = 22
                 |
                 |         distribution = [rank_sum, 9, 8]
                 |
                 |           p_value = .1995886466474702
```

Ahora, con muestras mayores y empates, el procedimiento realiza la aproximación normal. La hipótesis alternativa es $H_1 : median1 < median2$.

```
(%i1) load("stats")$
(%i2) x: [39,42,35,13,10,23,15,20,17,27]$
(%i3) y: [20,52,66,19,41,32,44,25,14,39,43,35,19,56,27,15]$
(%i4) test_rank_sum(x,y,'alternative='less);
                |                  RANK SUM TEST
                |
                |           method = Asymptotic test. Ties
                |
                |  hypotheses = H0: med1 = med2 , H1: med1 < med2
(%o4)           |
                |                 statistic = 48.5
                |
                | distribution = [normal, 79.5, 18.95419580097078]
                |
                |            p_value = .05096985666598441
```

`test_normality (x)` [Función]

Test de Shapiro-Wilk para el contraste de normalidad. El argumento x es una lista de números, con tamaño muestral mayor que 2 y menor o igual que 5000; bajo cualesquiera otras condiciones, la función `test_normality` emite un mensaje de error.

Referencia:

[1] Algorithm AS R94, Applied Statistics (1995), vol.44, no.4, 547-551

El resultado devuelto por la función `test_normality` es un objeto `inference_result` con los siguientes apartados:

1. `'statistic`: valor del estadístico W.
2. `'p_value`: p-valor bajo la hipótesis de normalidad.

Ejemplos:

Contrasta la normalidad de una población a partir de una muestra de tamaño 9.

```
(%i1) load("stats")$
(%i2) x:[12,15,17,38,42,10,23,35,28]$
(%i3) test_normality(x);
                          |      SHAPIRO - WILK TEST
                          |
(%o3)                     | statistic = .9251055695162436
                          |
                          |  p_value = .4361763918860381
```

`linear_regression (x)` [Función]

`linear_regression (x option)` [Función]

Regresión lineal múltiple, $y_i = b0 + b1 * x_1 i + b2 * x_2 i + ... + bk * x_k i + u_i$, donde u_i son variables aleatorias independientes $N(0, sigma)$. El argumento x debe ser una matriz con más de una columna. La última columna se considera que son las respuestas (y_i).

Opción:

- `'conflevel`, valor por defecto `95/100`, nivel de confianza para los intervalos de confianza; debe ser una expresión que tome un valor en el intervalo (0,1).

El resultado devuelto por la función `linear_regression` es un objeto `inference_result` de Maxima con los siguientes campos:

1. `'b_estimation`: estimadores de los coeficientes de regresión.
2. `'b_covariances`: matriz de covarianzas de los estimadores de los coeficientes de regresión.
3. `b_conf_int`: intervalos de confianza para los coeficientes de regresión.
4. `b_statistics`: estadísticos para los contrastes de los coeficientes.
5. `b_p_values`: p-valores para los contrastes de los coeficientes.
6. `b_distribution`: distribución de probabilidad para los contrastes de los coeficientes.
7. `v_estimation`: estimador insesgado de la varianza.
8. `v_conf_int`: intervalo de confianza de la varianza.
9. `v_distribution`: distribución de probabilidad para el contraste de la varianza.

10. `residuals`: residuos.
11. `adc`: coeficiente de determinación ajustado.
12. `aic`: Criterio de información de Akaike.
13. `bic`: Criterio de información de Bayes.

Solamente los apartados 1, 4, 5, 6, 7, 8, 9 y 11, en este orden, se muestran por defecto. El resto permanecen ocultos hasta que el usuario haga uso de las funciones `items_inference` y `take_inference`.

Ejemplo:

Ajustando un modelo lineal a una muestra tridimensional. La última columna se considera que son las respuestas (y_i).

```
(%i2) load("stats")$
(%i3) X:matrix(
    [58,111,64],[84,131,78],[78,158,83],
    [81,147,88],[82,121,89],[102,165,99],
    [85,174,101],[102,169,102])$
(%i4) fpprintprec: 4$
(%i5) res: linear_regression(X);
                |       LINEAR REGRESSION MODEL
                |
                | b_estimation = [9.054, .5203, .2397]
                |
                | b_statistics = [.6051, 2.246, 1.74]
                |
                | b_p_values = [.5715, .07466, .1423]
                |
(%o5)           |   b_distribution = [student_t, 5]
                |
                |         v_estimation = 35.27
                |
                |     v_conf_int = [13.74, 212.2]
                |
                |      v_distribution = [chi2, 5]
                |
                |             adc = .7922
(%i6) items_inference(res);
(%o6) [b_estimation, b_covariances, b_conf_int, b_statistics,
b_p_values, b_distribution, v_estimation, v_conf_int,
v_distribution, residuals, adc, aic, bic]
(%i7) take_inference('b_covariances, res);
                  [  223.9    - 1.12   - .8532  ]
                  [                             ]
(%o7)             [ - 1.12    .05367   - .02305 ]
                  [                             ]
                  [ - .8532  - .02305   .01898  ]
(%i8) take_inference('bic, res);
```

```
(%o8)                          30.98
(%i9) load("draw")$
(%i10) draw2d(
    points_joined = true,
    grid = true,
    points(take_inference('residuals, res)) )$
```

74.4 Funciones y variables para distribuciones especiales

`pdf_signed_rank` (*x*, *n*) [Función]

Función de densidad de probabilidad de la distribución exacta del estadístico de contraste del test de los rangos signados. El argumento *x* es un número real y *n* un entero positivo.

Véase también `test_signed_rank`.

`cdf_signed_rank` (*x*, *n*) [Función]

Función de probabilidad acumulada de la distribución exacta del estadístico de contraste del test de los rangos signados. El argumento *x* es un número real y *n* un entero positivo.

Véase también `test_signed_rank`.

`pdf_rank_sum` (*x*, *n*, *m*) [Función]

Función de densidad de probabilidad de la distribución exacta del estadístico de contraste de Wilcoxon-Mann-Whitney. El argumento *x* es un número real y *n* y *m* son ambos enteros positivos.

Véase también `test_rank_sum`.

`cdf_rank_sum` (*x*, *n*, *m*) [Función]

Función de probabilidad acumulada de la distribución exacta del estadístico de contraste de Wilcoxon-Mann-Whitney. El argumento *x* es un número real y *n* y *m* son ambos enteros positivos.

Véase también `test_rank_sum`.

75 stirling

75.1 Funciones y variables para stirling

`stirling` (*z*,*n*) [Función]
`stirling` (*z*,*n*,*pred*) [Función]

Sustituye `gamma(x)` por la fórmula de Stirling $O(1/x^(2n-1))$. Si n no es un entero no negativo, emite un mensaje de error. Con el tercer argumento opcional `pred`, la fórmula de Stirling sólo se aplica si `pred` vale `true`.

Referencia: Abramowitz & Stegun, " Handbook of mathematical functions", 6.1.40.

Ejemplos:

```
(%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
```

La función `stirling` conoce la diferencia existente entre la variable 'gamma' y la función 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 aplicar la fórmula de Stirling sólo a aquellos términos que contengan la variable `k`, hágase uso del tercer argumento opcional; por ejemplo,

```
(%i7) makegamma(pochhammer(a,k)/pochhammer(b,k));
(%o7) (gamma(b)*gamma(k+a))/(gamma(a)*gamma(k+b))
```

```
(%i8) stirling(%,1, lambda([s], not(freeof(k,s))));
(%o8) (%e^(b-a)*gamma(b)*(k+a)^(k+a-1/2)*(k+b)^(-k-b+1/2))/gamma(a)
```

Los términos `gamma(a)` y `gamma(b)` no contienen a `k`, por lo que la fórmula de Stirling no ha sido aplicada a ellos.

Antes de hacer uso de esta función ejecútese `load("stirling")`.

76 stringproc

76.1 Introducción al procesamiento de cadenas

El paquete `stringproc` amplía las capacidades de Maxima para manipular cadenas de caracteres, al tiempo que añade algunas funciones útiles para la lectura y escritura de ficheros.

Para dudas y fallos, por favor contáctese con `volkervannek at gmail dot com`.

En Maxima, una cadena de caracteres se construye fácilmente escribiéndola entre comillas dobles, como en `"texto"`. La función `stringp` comprueba si el argumento es una cadena.

```
(%i1) m: "text";
(%o1)                          text
(%i2) stringp(m);
(%o2)                          true
```

Los caracteres se representan como cadenas de longitud unidad. No se tratan como caracteres Lisp. Se pueden chequear con la función `charp` (o con `lcharp` para los caracteres Lisp). La conversión de caracteres Lisp a caracteres Maxima se realiza con la función `cunlisp`.

```
(%i1) c: "e";
(%o1)                            e
(%i2) [charp(c),lcharp(c)];
(%o2)                       [true, false]
(%i3) supcase(c);
(%o3)                            E
(%i4) charp(%);
(%o4)                          true
```

Todos los caracteres devueltos por las funciones de `stringproc` son caracteres de Maxima. Puesto que los caracteres introducidos son cadenas de longitud igual a la unidad, se pueden utilizar las funciones de cadenas también para los caracteres, como se ha hecho con `supcase` en el anterior ejemplo.

Es importante tener en cuenta que el primer carácter en una cadena de Maxima ocupa la posición 1. Esto se ha diseñado así para mantener la compatibilidad con las listas de Maxima. Véanse las definiciones de `charat` y `charlist` para ver ejemplos.

Las funciones de cadena se utilizan frecuentemente cuando se trabaja con ficheros. El siguiente ejemplo muestra algunas de estas funciones en acción.

Ejemplo:

La función `openw` envía un flujo de salida hacia un fichero, entonces `printf` permitirá formatera la escritura en este fichero. Véase `printf` para más detalles.

```
(%i1) s: openw("E:/file.txt");
(%o1)                     #<output stream E:/file.txt>
(%i2) for n:0 thru 10 do printf( s, "~d ", fib(n) );
(%o2)                                  done
(%i3) printf( s, "~%~d ~f ~a ~a ~f ~e ~a~%",
              42,1.234,sqrt(2),%pi,1.0e-2,1.0e-2,1.0b-2 );
(%o3)                                  false
```

```
(%i4) close(s);
(%o4)                                true
```

Una vez cerrado el flujo, se podrá abrir nuevamente. La función `readline` devuelve el renglón entero como una única cadena. El paquete `stringproc` dispone de muchas funciones para manipular cadenas. La separación de palabras se puede hacer con `split` o `tokens`.

```
(%i5) s: openr("E:/file.txt");
(%o5)                     #<input stream E:/file.txt>
(%i6) readline(s);
(%o6)                     0 1 1 2 3 5 8 13 21 34 55
(%i7) line: readline(s);
(%o7)               42 1.234 sqrt(2) %pi 0.01 1.0E-2 1.0b-2
(%i8) list: tokens(line);
(%o8)           [42, 1.234, sqrt(2), %pi, 0.01, 1.0E-2, 1.0b-2]
(%i9) map( parsetoken, list );
(%o9)           [42, 1.234, false, false, 0.01, 0.01, false]
```

La función `parsetoken` sólo analiza sintácticamente números enteros y decimales. El análisis de símbolos y números decimales grandes (*big floats*) necesita `parse_string`, que se cargar automáticamente desde `eval_string.lisp`.

```
(%i5) s: openr("E:/file.txt");
(%o5)                     #<input stream E:/file.txt>
(%i6) readline(s);
(%o6)                     0 1 1 2 3 5 8 13 21 34 55
(%i7) line: readline(s);
(%o7)               42 1.234 sqrt(2) %pi 0.01 1.0E-2 1.0b-2
(%i8) list: tokens(line);
(%o8)           [42, 1.234, sqrt(2), %pi, 0.01, 1.0E-2, 1.0b-2]
(%i9) map( parse_string, list );
(%o9)            [42, 1.234, sqrt(2), %pi, 0.01, 0.01, 1.0b-2]
(%i10) float(%);
(%o10) [42.0, 1.234, 1.414213562373095, 3.141592653589793, 0.01,
                                                      0.01, 0.01]
(%i11) readline(s);
(%o11)                               false
(%i12) close(s)$
```

La función `readline` devuelve `false` cuando se alcanza el final del fichero.

76.2 Funciones y variables para entrada y salida

Ejemplo:

```
(%i1) s: openw("E:/file.txt");
(%o1)                     #<output stream E:/file.txt>
(%i2) control:
"~2tAn atom: ~20t~a~%~2tand a list: ~20t~{~r ~}~%~2t\
          and an integer: ~20t~d~%"$
(%i3) printf( s,control, 'true,[1,2,3],42 )$
(%o3)                                false
```

```
(%i4) close(s);
(%o4)                                true
(%i5) s: openr("E:/file.txt");
(%o5)                     #<input stream E:/file.txt>
(%i6) while stringp( tmp:readline(s) ) do print(tmp)$
  An atom:          true
  and a list:       one two three
  and an integer:   42
(%i7) close(s)$
```

`close` (*stream*) [Función]
Cierra el flujo de datos *stream* y devuelve `true` si *stream* había sido abierto.

`flength` (*stream*) [Función]
Devuelve el número de elementos en *stream*, el cual debe ser un flujo de datos desde o hacia un fichero.

`fposition` (*stream*) [Función]
`fposition` (*stream*, *pos*) [Función]
Devuelve la posición actual en el flujo de datos *stream* si no se utiliza *pos*. Si se utiliza *pos*, `fposition` fija la posición en *stream*. *stream* debe ser un flujo de datos desde o hacia un fichero y *pos* debe ser un entero positivo que hace corresponder al primer elemento de *stream* la posición 1.

`freshline` () [Función]
`freshline` (*stream*) [Función]
Escribe una nueva línea (en el flujo de datos *stream*) si la posición actual no corresponde al inicio de la línea.

Véase también `newline`.

`get_output_stream_string` (*stream*) [Función]
Devuelve una cadena con todos los caracteres presentes en *stream*, que debe ser un flujo de datos de salida abiero. Los caracteres devueltos son eliminados de *stream*.

Para un ejemplo, véase `make_string_output_stream` .

`make_string_input_stream` (*string*) [Función]
`make_string_input_stream` (*string*, *start*) [Función]
`make_string_input_stream` (*string*, *start*, *end*) [Función]
Devuelve un flujo de entrada que contiene partes de *string* junto con el carácter de final de fichero. Sin argumentos opcionales, el flujo contiene la cadena entera y se posiciona frente al primer carácter. Los argumentos *start* y *end* definen la subcadena contenida en el flujo. El primer carácter está disponible en la posición 1.

Ejemplo:

```
(%i1) istream : make_string_input_stream("text", 1, 4);
(%o1)              #<string-input stream from "text">
(%i2) (while (c : readchar(istream)) # false do sprint(c), newline())$
t e x
(%i3) close(istream)$
```

`make_string_output_stream ()` [Función]

Devuelve un flujo de salida que acepta caracteres. Los caracteres de este flujo podrán obtenerse con `get_output_stream_string`.

Ejemplo:

```
(%i1) ostream : make_string_output_stream();
(%o1)               #<string-output stream 09622ea0>
(%i2) printf(ostream, "foo")$

(%i3) printf(ostream, "bar")$

(%i4) string : get_output_stream_string(ostream);
(%o4)                            foobar
(%i5) printf(ostream, "baz")$

(%i6) string : get_output_stream_string(ostream);
(%o6)                              baz
(%i7) close(ostream)$
```

`newline ()` [Función]
`newline (`*stream*`)` [Función]

Escribe una nueva línea (en el flujo de datos *stream*).

Véase `sprint` para un ejemplo de uso de `newline()`.

Nótese que hay algunos casos en los que `newline` no trabaja según lo esperado.

`opena (`*file*`)` [Función]

Devuelve un flujo de datos al fichero *file*. Si se abre un fichero ya existente, `opena` añade elementos al final del fichero.

`openr (`*file*`)` [Función]

Devuelve un flujo de datos de entrada al fichero *file*. Si *file* no existe, será creado.

`openw (`*file*`)` [Función]

Devuelve un flujo de datos de salida al fichero *file*. Si *file* no existe, será creado. Si se abre un fichero ya existente, `openw` lo modifica borrando el contenido anterior.

`printf (`*dest*, *string*`)` [Función]
`printf (`*dest*, *string*, *expr_1*, ..., *expr_n*`)` [Función]

Genera una cadena de caracteres a partir de la cadena de control *string*, teniendo en cuenta que las tildes introducen directivas. El carácter que va después de la tilde, posiblemente precedido por parámetros y modificadores, especifica el tipo de formato que se desea. La mayor parte de las directivas usan uno o más elementos de los argumentos *expr_1*, ..., *expr_n* para crear la salida.

Si *dest* es un flujo o vale `true`, entonces `printf` devuelve `false`. En otro caso, `printf` devuelve una cadena conteniendo la salida.

`printf` da acceso a la función `format` de Common Lisp. El siguiente ejemplo muestra la relación entre estas dos funciones.

```
(%i1) printf(true, "R~dD~d~%", 2, 2);
```

```
R2D2
(%o1)                                  false
(%i2) :lisp (format t "R~dD~d~%" 2 2)
R2D2
NIL
```

La siguiente descripción es un simple resumen de las posibilidades de `printf`. La función `format` de Common Lisp se encuentra descrita en detalle en muchas referencias, como el manual libre "Common Lisp the Language" de Guy L. Steele; en particular, el capítulo 22.3.3.

```
~%       nueva línea
~&       línea de refresco
~t       tabulación
~$       moneda
~d       entero en base decimal
~b       entero en base binaria
~o       entero en base octal
~x       entero en base hexadecimal
~br      entero en base b
~r       deletrea un entero
~p       plural
~f       decimal en coma flotante
~e       notación científica
~g       ~f o ~e, dependiendo de la magnitud
~h       número decimal grande (bigfloat)
~a       utiliza la función string de Maxima
~s       como ~a, pero las cadenas se devuelven entre "comillas dobles"
~~       ~
~<       justificación, ~> termina
~(       conversor mayúscula/minúscula, ~) termina
~[       selección, ~] termina
~{       iteración, ~} termina
```

La directiva `~h` para números decimales grandes no pertenece al estándar de Lisp, por lo que se ilustra más abajo.

La directiva `~*` no está soportada.

Ejemplos:

Si *dest* es un flujo o vale `true`, entonces `printf` devuelve `false`. En otro caso, `printf` devuelve una cadena conteniendo la salida.

```
(%i1) printf( false, "~a ~a ~4f ~a ~@r",
              "String",sym,bound,sqrt(12),144), bound = 1.234;
(%o1)                 String sym 1.23 2*sqrt(3) CXLIV
(%i2) printf( false,"~{~a ~}",["one",2,"THREE"] );
(%o2)                           one 2 THREE
(%i3) printf(true,"~{~{~9,1f ~}~%~}",mat ),
          mat = args(matrix([1.1,2,3.33],[4,5,6],[7,8.88,9]))$
      1.1       2.0       3.3
```

```
      4.0       5.0       6.0
      7.0       8.9       9.0
(%i4) control: "~:(~r~) bird~p ~[is~;are~] singing."$
(%i5) printf( false,control, n,n,if n=1 then 1 else 2 ), n=2;
(%o5)                    Two birds are singing.
```

La directiva `~h` se ha introducido para formatear decimales grandes.

```
~w,d,e,x,o,p@H
 w : width
 d : decimal digits behind floating point
 e : minimal exponent digits
 x : preferred exponent
 o : overflow character
 p : padding character
 @ : display sign for positive numbers
```

```
(%i1) fpprec : 1000$
(%i2) printf(true, "|~h|~%", 2.b0^-64)$
|0.00000000000000000005421010862427522170037264004349708557128906250|
(%i3) fpprec : 26$
(%i4) printf(true, "|~h|~%", sqrt(2))$
|1.4142135623730950488016887|
(%i5) fpprec : 24$
(%i6) printf(true, "|~h|~%", sqrt(2))$
|1.41421356237309504880169|
(%i7) printf(true, "|~28h|~%", sqrt(2))$
|   1.41421356237309504880169|
(%i8) printf(true, "|~28,,,,,'*h|~%", sqrt(2))$
|***1.41421356237309504880169|
(%i9) printf(true, "|~,18h|~%", sqrt(2))$
|1.414213562373095049|
(%i10) printf(true, "|~,,,-3h|~%", sqrt(2))$
|1414.21356237309504880169b-3|
(%i11) printf(true, "|~,,2,-3h|~%", sqrt(2))$
|1414.21356237309504880169b-03|
(%i12) printf(true, "|~20h|~%", sqrt(2))$
|1.41421356237309504880169|
(%i13) printf(true, "|~20,,,,'+h|~%", sqrt(2))$
|++++++++++++++++++++|
```

`readchar` (*stream*) [Función]

Elimina y devuelve el primer carácter de *stream*. Si se ha alcanzado el final del fichero, `readchar` devuelve `false`.

Para un ejemplo, véase `make_string_input_stream`.

`readline` (*stream*) [Función]

Devuelve una cadena con los caracteres desde la posición actual en el flujo de datos *stream* hasta el final de la línea, o `false` si se ha alcanzado el final del fichero.

`sprint` (*expr_1*, ..., *expr_n*) [Función]

Evalúa y muestra sus argumentos uno tras otro en un renglón comenzando por su extremo izquierdo.

La función `newline()`, que se carga automáticamente desde `stringproc.lisp`, puede ser de utilidad si se quiere intercalar un salto de línea.

```
(%i1) for n:0 thru 19 do sprint( fib(n) )$
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
(%i2) for n:0 thru 22 do (
         sprint(fib(n)), if mod(n,10)=9 then newline() )$
0 1 1 2 3 5 8 13 21 34
55 89 144 233 377 610 987 1597 2584 4181
6765 10946 17711
```

76.3 Funciones y variables para caracteres

`alphacharp` (*char*) [Función]

Devuelve `true` si *char* es una carácter alfabético.

`alphanumericp` (*char*) [Función]

Devuelve `true` si *char* es una carácter alfabético o un dígito.

`ascii` (*int*) [Función]

Devuelve el carácter correspondiente al número ASCII *int*, debiendo ser $-1 < int < 256$.

```
(%i1) for n from 0 thru 255 do (
  tmp: ascii(n),
  if alphacharp(tmp) then sprint(tmp), if n=96 then newline() )$
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

`cequal` (*char_1*, *char_2*) [Función]

Devuelve `true` si *char_1* y *char_2* son el mismo carácter.

`cequalignore` (*char_1*, *char_2*) [Función]

Como `cequal`, pero ignora si las letras están en mayúsculas o minúsculas.

`cgreaterp` (*char_1*, *char_2*) [Función]

Devuelve `true` si el número ASCII de *char_1* es mayor que el de *char_2*.

`cgreaterpignore` (*char_1*, *char_2*) [Función]

Como `cgreaterp`, pero ignora si las letras están en mayúsculas o minúsculas.

`charp` (*obj*) [Función]

Devuelve `true` si *obj* es un carácter de Maxima.

`cint` (*char*) [Función]

Devuelve el número ASCII de *char*.

`clessp` (*char_1*, *char_2*) [Función]

Devuelve `true` si el número ASCII de *char_1* es menor que el de *char_2*.

clesspignore (*char_1*, *char_2*) [Función]
Como clessp, pero ignora si las letras están en mayúsculas o minúsculas.

constituent (*char*) [Función]
Devuelve true si *char* es un carácter gráfico y no el carácter espacio. Un carácter gráfico es el que se puede ver y con un espacio añadido; constituent está definido por Paul Graham, ANSI Common Lisp, 1996, page 67.

```
(%i1) for n from 0 thru 255 do (
tmp: ascii(n), if constituent(tmp) then sprint(tmp) )$
! " # % ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B
C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _ ` a b c
d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

cunlisp (*lisp_char*) [Función]
Convierte un carácter Lisp en uno de Maxima. El uso de esta función por parte del usuario no será necesario.

digitcharp (*char*) [Función]
Devuelve true si *char* es un dígito.

lcharp (*obj*) [Función]
Devuelve true si *obj* es un carácter de Lisp. El uso de esta función por parte del usuario no será necesario.

lowercasep (*char*) [Función]
Devuelve true si *char* es un carácter en minúscula.

newline [Variable]
El carácter de nueva línea.

space [Variable]
El carácter de espacio.

tab [Variable]
El carácter de tabulación.

uppercasep (*char*) [Función]
Devuelve true si *char* es un carácter en mayúscula.

76.4 Funciones y variables para cadenas

base64 (*string*) [Función]
Devuelve la representación en base 64 de *string* en formato de cadena de caracteres.

Ejemplo:

```
(%i1) base64 : base64("foo bar baz");
(%o1)                       Zm9vIGJhciBiYXo=
(%i2) string : base64_decode(base64);
(%o2)                          foo bar baz
```

`base64_decode` (*base64-string*) [Función]

Decodifica la cadena de caracteres *base64-string*, codificada en base 64, y devuelve la cadena original.

Para un ejemplo, véase `base64`.

`charat` (*string*, *n*) [Función]

Devuelve el *n*-ésimo carácter de *string*. Al primer carácter de *string* le corresponde *n* = 1.

```
(%i1) charat("Lisp",1);
(%o1)                           L
```

`charlist` (*string*) [Función]

Devuelve una lista con todos los caracteres de *string*.

```
(%i1) charlist("Lisp");
(%o1)                      [L, i, s, p]
(%i2) %[1];
(%o2)                           L
```

`eval_string` (*str*) [Función]

Analiza sintácticamente la cadena *str* como una expresión de Maxima y la evalúa. La cadena *str* puede terminar o no con cualquiera de los símbolos de final de sentencia (dólar `$` o punto y coma `;`). Sólo se analiza la primera expresión si hay más de una.

Se emitirá un mensaje de error si *str* no es una cadena.

Ejemplos:

```
(%i1) eval_string ("foo: 42; bar: foo^2 + baz");
(%o1)                         42
(%i2) eval_string ("(foo: 42, bar: foo^2 + baz)");
(%o2)                    baz + 1764
```

Véase también `parse_string`.

`md5sum` (*string*) [Función]

Devuelve, en formato de cadena de caracteres, el resultado de la suma de verificación md5 del argumento *string*. Para obtener el valor devuelto por la función como número entero, fijar la base numérica de entrada a 16 y a~nadir como prefijo el cero.

Ejemplo:

```
(%i1) string : md5sum("foo bar baz");
(%o1)                  ab07acbb1e496801937adfa772424bf7
(%i2) ibase : obase : 16.$

(%i3) integer : parse_string(sconcat(0, string));
(%o3)                 0ab07acbb1e496801937adfa772424bf7
```

`parse_string` (*str*) [Función]

Analiza sintácticamente la cadena *str* como una expresión de Maxima, pero no la evalúa. La cadena *str* puede terminar o no con cualquiera de los símbolos de final de sentencia (dólar `$` o punto y coma `;`). Sólo se analiza la primera expresión si hay más de una.

Se emitirá un mensaje de error si *str* no es una cadena.

Ejemplos:

```
(%i1) parse_string ("foo: 42; bar: foo^2 + baz");
(%o1)                     foo : 42
(%i2) parse_string ("(foo: 42, bar: foo^2 + baz)");
                                        2
(%o2)          (foo : 42, bar : foo  + baz)
```

Véase también `eval_string`.

`scopy` (*string*) [Función]

Devuelve una copia nueva de la cadena *string*.

`sdowncase` (*string*) [Función]
`sdowncase` (*string*, *start*) [Función]
`sdowncase` (*string*, *start*, *end*) [Función]

Convierte caracteres en minúscula a mayúscula. Véase también `supcase`.

`sequal` (*string_1*, *string_2*) [Función]

Devuelve `true` si *string_1* y *string_2* son dos cadenas de caracteres iguales.

`sequalignore` (*string_1*, *string_2*) [Función]

Igual que `sequal` pero no diferencia entre minúsculas y mayúsculas..

`sexplode` (*string*) [Función]

El nombre `sexplode` es un seudónimo de la función `charlist`.

`simplode` (*list*) [Función]
`simplode` (*list*, *delim*) [Función]

La función `simplode` admite como entrada una lista de expresiones para luego convertirla en una cadena de caracteres. Si no se utiliza la opción *delim* para indicar el delimitador, entonces `simplode` no hace uso de ninguno. El valor de *delim* puede ser cualquier cadena.

```
(%i1) simplode(["xx[",3,"]:",expand((x+y)^3)]);
(%o1)             xx[3]:y^3+3*x*y^2+3*x^2*y+x^3
(%i2) simplode( sexplode("stars")," * " );
(%o2)                    s * t * a * r * s
(%i3) simplode( ["One","more","coffee."]," " );
(%o3)                    One more coffee.
```

`sinsert` (*seq*, *string*, *pos*) [Función]

Devuelve la concatenación de las cadenas `substring (`*string*`, 1, `*pos*` - 1)`, *seq* y `substring (`*string*`,` *pos*`)`. Nótese que al primer carácter de *string* le corresponde la posición 1.

```
(%i1) s: "A submarine."$
(%i2) concat( substring(s,1,3),"yellow ",substring(s,3) );
(%o2)                  A yellow submarine.
(%i3) sinsert("hollow ",s,3);
(%o3)                  A hollow submarine.
```

`sinvertcase` (*string*) [Función]
`sinvertcase` (*string*, *start*) [Función]
`sinvertcase` (*string*, *start*, *end*) [Función]
Devuelve la misma cadena *string* pero con todos sus caracteres desde la posición *start* hasta *end* invertidos, esto es, las mayúsculas se convierten en minúsculas y éstas en mayúsculas. Si no se incluye el argumento *end*, se invierten todos los caracteres desde *start* hasta el final de la cadena.

```
(%i1) sinvertcase("sInvertCase");
(%o1)                      SiNVERTcASE
```

`slength` (*string*) [Función]
Devuelve el número de caracteres de *string*.

`smake` (*num*, *char*) [Función]
Construye una cadena de longitud *num* con todos sus caracteres iguales a *char*.

```
(%i1) smake(3,"w");
(%o1)                           www
```

`smismatch` (*string_1*, *string_2*) [Función]
`smismatch` (*string_1*, *string_2*, *test*) [Función]
Devuelve la posición del primer carácter de *string_1* distinto del correpondiente a *string_2*. La respuesta será `false` si no existe tal carácter. Por defecto, la función de comparación es `sequal`. Si se quiere ignorar la diferencia entre mayúsculas y minúsculas, hágase uso de `sequalignore` para el argumento *test*.

```
(%i1) smismatch("seven","seventh");
(%o1)                            6
```

`split` (*string*) [Función]
`split` (*string*, *delim*) [Función]
`split` (*string*, *delim*, *multiple*) [Función]
Devuelve la lista de todos los lexemas (*tokens*) de *string*. La función `split` utiliza *delim* como delimitador, y en caso de no ser utilizado este argumento, será utilizado el espacio en blanco como delimitador por defecto. El argumento *multiple* es una variable booleana con valor `true` por defecto. Los delimitadores múltiples se leen como uno solo, lo que resulta de utilidad si las tabulaciones son almacenadas como secuencias de espacios en blanco. Si a *multiple* se le asigna el valor `false`, se consirarárán todos los delimitadores.

```
(%i1) split("1.2   2.3   3.4   4.5");
(%o1)                  [1.2, 2.3, 3.4, 4.5]
(%i2) split("first;;third;fourth",";",false);
(%o2)               [first, , third, fourth]
```

`sposition` (*char*, *string*) [Función]
Devuelve la posición del primer carácter de *string* que coincide con *char*. Al primer carácter de *string* le corresponde la posición 1. Para cuando se quiera ignorar la diferencia entre mayúsculas y minúsculas, véase *ssearch*.

sremove (*seq*, *string*) [Función]
sremove (*seq*, *string*, *test*) [Función]
sremove (*seq*, *string*, *test*, *start*) [Función]
sremove (*seq*, *string*, *test*, *start*, *end*) [Función]

Devuelve la cadena *string* pero sin las subcadenas que coinciden con *seq*. La función de comparación por defecto es `sequal`. Si se quiere ignorar la diferencia entre mayúsculas y minúsculas, hágase uso de `sequalignore` para el argumento *test*. Utilícense *start* y *end* para acotar la búsqueda. Al primer carácter de *string* le corresponde la posición 1.

```
(%i1) sremove("n't","I don't like coffee.");
(%o1)                   I do like coffee.
(%i2) sremove ("DO ",%,'sequalignore);
(%o2)                    I like coffee.
```

sremovefirst (*seq*, *string*) [Función]
sremovefirst (*seq*, *string*, *test*) [Función]
sremovefirst (*seq*, *string*, *test*, *start*) [Función]
sremovefirst (*seq*, *string*, *test*, *start*, *end*) [Función]

Actúa de forma similar a la función `sremove`, pero sólo elimina la primera aparición de la subcadena `seq`.

sreverse (*string*) [Función]

Devuelve una cadena con todos los caracteres de *string* en orden inverso.

ssearch (*seq*, *string*) [Función]
ssearch (*seq*, *string*, *test*) [Función]
ssearch (*seq*, *string*, *test*, *start*) [Función]
ssearch (*seq*, *string*, *test*, *start*, *end*) [Función]

Devuelve la posición de la primera subcadena de *string* que coincide con la cadena *seq*. La función de comparación por defecto es `sequal`. Si se quiere ignorar la diferencia entre mayúsculas y minúsculas, hágase uso de `sequalignore` para el argumento *test*. Utilícense *start* y *end* para acotar la búsqueda. Al primer carácter de *string* le corresponde la posición 1.

```
(%i1) ssearch("~s","~{~S ~}~%",'sequalignore);
(%o1)                                  4
```

ssort (*string*) [Función]
ssort (*string*, *test*) [Función]

Devuelve una cadena con todos los caracteres de *string* en un orden tal que no haya dos caracteres sucesivos *c* y *d* que verifiquen que `test (c, d)` sea igual `false` y `test (d, c)` igual a `true`. La función de comparación *test* por defecto es *clessp*, siendo el conjunto de posibles valores para este argumento `{clessp, clesspignore, cgreaterp, cgreaterpignore, cequal, cequalignore}`.

```
(%i1) ssort("I don't like Mondays.");
(%o1)                    '.IMaddeiklnnoosty
(%i2) ssort("I don't like Mondays.",'cgreaterpignore);
(%o2)                 ytsoonnMlkIiedda.'
```

`ssubst` (*new*, *old*, *string*) [Función]
`ssubst` (*new*, *old*, *string*, *test*) [Función]
`ssubst` (*new*, *old*, *string*, *test*, *start*) [Función]
`ssubst` (*new*, *old*, *string*, *test*, *start*, *end*) [Función]

Devuelve una cadena similar a *string* pero en la que aquellas subcadenas coincidentes con *old* han sido sustituidas por *new*. Las subcadenas *old* y *new* no necesitan ser de la misma longitud. La función de comparación por defecto es `sequal`. Si se quiere ignorar la diferencia entre mayúsculas y minúsculas durante la búsqueda de *old*, hágase uso de `sequalignore` para el argumento *test*. Utilícense *start* y *end* para acotar la búsqueda. Al primer carácter de *string* le corresponde la posición 1.

```
(%i1) ssubst("like","hate","I hate Thai food. I hate green tea.");
(%o1)          I like Thai food. I like green tea.
(%i2) ssubst("Indian","thai",%,'sequalignore,8,12);
(%o2)         I like Indian food. I like green tea.
```

`ssubstfirst` (*new*, *old*, *string*) [Función]
`ssubstfirst` (*new*, *old*, *string*, *test*) [Función]
`ssubstfirst` (*new*, *old*, *string*, *test*, *start*) [Función]
`ssubstfirst` (*new*, *old*, *string*, *test*, *start*, *end*) [Función]

Actúa de forma similar a la función `subst`, pero sólo hace la sustitución en la primera coincidencia con *old*.

`strim` (*seq*,*string*) [Función]

Devuelve la cadena *string* pero recortando los caracteres de *seq* que tuviese en sus extremos.

```
(%i1) "/* comment */"$
(%i2) strim(" /*",%);
(%o2)                          comment
(%i3) slength(%);
(%o3)                             7
```

`striml` (*seq*, *string*) [Función]

Actúa de forma similar a `strim`, pero sólo recorta en el extremo final de *string*.

`strimr` (*seq*, *string*) [Función]

Actúa de forma similar a `strim`, pero sólo recorta en el extremo inicial de *string*.

`stringp` (*obj*) [Función]

Devuelve `true` si *obj* es una cadena. Véase un ejemplo en la introducción.

`substring` (*string*, *start*) [Función]
`substring` (*string*, *start*, *end*) [Función]

Devuelve la subcadena de *string* que comienza en la posición *start* y termina en la posición *end*. El carácter en la posición *end* no se incluye. En caso de no suministrarse el argumento *end*, la subcadena se extenderá hasta el final. Al primer carácter de *string* le corresponde la posición 1.

```
(%i1) substring("substring",4);
(%o1)                          string
```

```
(%i2) substring(%,4,6);
(%o2)                          in
```

supcase (*string*) [Función]
supcase (*string*, *start*) [Función]
supcase (*string*, *start*, *end*) [Función]

Devuelve la cadena *string* con todos sus caracteres entre las posiciones *start* y *end* en minúscula transformados a mayúscula. En caso de no suministrarse el argumento *end*, los cambios se extenderán hasta el final.

```
(%i1) supcase("english",1,2);
(%o1)                         English
```

tokens (*string*) [Función]
tokens (*string*, *test*) [Función]

Devuelve la lista de todos los lexemas (*tokens*) de *string*. Los lexemas son subcadenas cuyos caracteres satisfacen la condición *test*. Si no se suministra el argumento *test*, se utilizará la condición *constituent*, siendo el conjunto de las otras alternativas {constituent, alphacharp, digitcharp, lowercasep, uppercasep, charp, characterp, alphanumericp}.

```
(%i1) tokens("24 October 2005");
(%o1)                    [24, October, 2005]
(%i2) tokens("05-10-24",'digitcharp);
(%o2)                       [05, 10, 24]
(%i3) map(parse_string,%);
(%o3)                        [5, 10, 24]
```

77 to_poly_solve

77.1 Funciones y variables para to_poly_solve

Los paquetes `to_poly` y `to_poly_solve` son experimentales, siendo posible que las especificaciones de sus funciones puedan cambiar en el futuro, o que algunas de estas funciones puedan ser incorporadas a otras partes de Maxima.

Los paquetes `to_poly` y `to_poly_solve`, junto con su documentación, fue escrito por Barton Willis de la Universidad de Nebraska en Kearney.

`%and` [Operador]

El operador `%and` es una conjunción lógica. Maxima simplifica una expresión `%and` a `true`, `false` o a una expresión lógicamente equivalente, pero simplificada. El operador `%and` es asociativo, conmutativo e idempotente. Así, cuando `%and` devuelva una forma nominal, sus argumentos no serán redundantes; por ejemplo,

```
(%i1) a %and (a %and b);
(%o1)                        a %and b
```

Si uno de los argumentos de la conjunción es la negación de otro argumento, `%and` devuelve `false`:

```
(%i2) a %and (not a);
(%o2) false
```

Si cualquiera de los argumentos vale `false`, la conjunción devuelve `false`, incluso cuando haya algún otro argumento que sea una expresión no booleana; por ejemplo,

```
(%i2) a %and (not a);
(%o2)                          false
```

Los argumentos de la expresión `%and` que sean inecuaciones se reducen con la simplificación de Fourier; el método que se aplica dispone de un pre-procesador que convierte algunas, pero no todas, las inecuaciones no lineales a lineales. Por ejemplo, el método de simplificación de Fourier simplifica `abs(x) + 1 > 0` a `true`:

```
(%i4) (x < 1) %and (abs(x) + 1 > 0);
(%o4)                          x < 1
```

Notas

- La variable opcional `prederror` no altera la simplificación de las expresiones `%and`.
- Para evitar errores en la precedencia de las operaciones, se recomienda utilizar paréntesis en las expresiones combinadas en las que aparezcan los operadores `%and`, `%or` y `not` .
- Los operadores `and` y `or`, tal como están programados en Maxima, no conocen las propiedades asociativa ni conmutativa.

Limitaciones La conjunción `%and` simplifica inecuaciones *locamente*, no *globalmente*, lo que significa que conjunciones tales como

```
(%i5) (x < 1) %and (x > 1);
(%o5)                  (x > 1) %and (x < 1)
```

no simplifican a `false`. Además, las rutinas de eliminación de Fourier ignoran los hechos almacenados en la base de datos.

```
(%i6) assume(x > 5);
(%o6)                          [x > 5]
(%i7) (x > 1) %and (x > 2);
(%o7)                  (x > 1) %and (x > 2)
```

Por último, las inecuaciones no lineales que no se puedan reducir de manera sencilla a formas lineales, no se simplifican.

No está soportada la distributividad de `%and` respecto de `%or`, ni la negación respecto de `%and`.

Para hacer uso de este operador, ejecútese `load("to_poly_solve")`.

Véanse también `%or`, `%if`, `and`, `or` y `not`.

`%if` (*bool*, *a*, *b*) [Operador]

El operador `%if` es un condicional. La condición *bool* debe tomar un valor lógico; cuando sea `true`, se devolverá el segundo argumento, y cuando valga `false`, el segundo. En cualquier otro caso, se obtiene una forma nominal.

En Maxima, las expresiones con desigualdades o igualdades no adquieren valores lógicos; por ejemplo, $5 < 6$ no se simplifica a `true`, ni $5 = 6$ `false`. Sin embargo, en la condición de una sentencia `%if`, Maxima intenta determinar el valor lógico de la expresión de forma automática. Véase un ejemplo:

```
(%i1) f : %if(x # 1, 2, 8);
(%o1)                  %if(x - 1 # 0, 2, 8)
(%i2) [subst(x = -1,f), subst(x=1,f)];
(%o2)                          [2, 8]
```

Si en la condición aparece una inecuación, Maxima la reduce con una simplificación de Fourier.

Notas

- Si la condición no se reduce a un valor lógico, Maxima devuelve una forma nominal:

```
(%i3) %if(42,1,2);
(%o3)                        %if(42, 1, 2)
```

- El operador `if` de Maxima es n-ario, pero el operador `%if` no lo es.

Por último, las inecuaciones no lineales que no se puedan reducir de manera sencilla a formas lineales, no se simplifican.

Para hacer uso de este operador, ejecútese `load("to_poly_solve")`.

`%or` [Operador]

El operador `%or` es una disyunción lógica. Maxima simplifica una expresión `%or` a `true`, `false` o a una expresión lógicamente equivalente, pero simplificada. El operador `%or` es asociativo, conmutativo e idempotente. Así, cuando `%or` devuelva una forma nominal, sus argumentos no serán redundantes; por ejemplo,

```
(%i1) a %or (a %or b);
(%o1)                          a %or b
```

Si uno de los argumentos de la disyunción es la negación de otro argumento, `%or` devuelve `true`:

```
(%i2) a %or (not a);
(%o2)                          true
```

Si cualquiera de los argumentos vale `true`, la disyunción devuelve `true`, incluso cuando haya algún otro argumento que sea una expresión no booleana; por ejemplo,

```
(%i3) 42 %or true;
(%o3)                          true
```

Los argumentos de la expresión `%or` que sean inecuaciones se reducen con la simplificación de Fourier. Por ejemplo, el método de simplificación de Fourier simplifica `abs(x) + 1 > 0` a `true`:

```
(%i4) (x < 1) %or (abs(x) + 1 > 0);
(%o4)                          true
```

Notas

- La variable opcional `prederror` no altera la simplificación de las expresiones `%or`.
- Para evitar errores en la precedencia de las operaciones, se recomienda utilizar paréntesis en las expresiones combinadas en las que aparezcan los operadores `%and`, `%or` y `not` .
- Los operadores `and` y `or`, tal como están programados en Maxima, no conocen las propiedades asociativa ni conmutativa.

Limitaciones La conjunción `%or` simplifica inecuaciones *locamente*, no *globalmente*, lo que significa que disyunciones tales como

```
(%i1) (x < 1) %or (x >= 1);
(%o1) (x > 1) %or (x >= 1)
```

no simplifican a `true`. Además, las rutinas de eliminación de Fourier ignoran los hechos almacenados en la base de datos.

```
(%i2) assume(x > 5);
(%o2)                         [x > 5]
(%i3) (x > 1) %and (x > 2);
(%o3)                 (x > 1) %and (x > 2)
```

Por último, las inecuaciones no lineales que no se puedan reducir de manera sencilla a formas lineales, no se simplifican.

No está soportada la distributividad de `%or` respecto de `%and`, ni la negación respecto de `%or`.

Para hacer uso de este operador, ejecútese `load("to_poly_solve")`.

Véanse también `%and`, `%if`, `and`, `or` y `not`.

`complex_number_p` (*x*) [Función]

La función `complex_number_p` devuelve `true` si su argumento es de cualquiera de las formas `a + %i * b`, `a`, `%i b` o `%i`, donde `a` y `b` son racionales o decimales en coma flotante, de precisión doble o arbitraria (*bigfloats*); para cualesquiera otros argumentos, `complex_number_p` devuelve `false`.

Ejemplo:

```
(%i1) map('complex_number_p,[2/3, 2 + 1.5 * %i, %i]);
(%o1)                  [true, true, true]
(%i2) complex_number_p((2+%i)/(5-%i));
(%o2)                          false
(%i3) complex_number_p(cos(5 - 2 * %i));
(%o3)                          false
```

Véase también `isreal_p`.

Para hacer uso de esta función, ejecútese `load("to_poly_solve")`.

`compose_functions` (*l*) [Función]

La función `compose_functions(l)` devuelve una expresión lambda que es la composición de las funciones presentes en la lista *l*. Las funciones se aplican de derecha a izquierda.

Ejemplo:

```
(%i1) compose_functions([cos, exp]);
                                                %g151
(%o1)             lambda([%g151], cos(%e     ))
(%i2) %(x);
                                      x
(%o2)                         cos(%e )
```

Si la lista está vacía devuelve la función identidad:

```
(%i3) compose_functions([]);
(%o3)                 lambda([%g152], %g152)
(%i4)  %(x);
(%o4)                            x
```

Notas

- Cuando Maxima detecta que un miembro de la lista no es un símbolo o expresión lambda, la función `funmake` (no `compose_functions`) muestra un mensaje de error:

```
(%i5) compose_functions([a < b]);

funmake: first argument must be a symbol, subscripted symbol,
string, or lambda expression; found: a < b
#0: compose_functions(l=[a < b])(to_poly_solve.mac line 40)
 -- an error. To debug this try: debugmode(true);
```

- Para evitar conflictos de nombres, la variable independiente se determina con la función `new_variable`:

```
(%i6) compose_functions([%g0]);
(%o6)              lambda([%g154], %g0(%g154))
(%i7) compose_functions([%g0]);
(%o7)              lambda([%g155], %g0(%g155))
```

Aunque las variables dependientes sean diferentes, Maxima es capaz de determinar que las expresiones lambda son semánticamente equivalentes:

```
(%i8) is(equal(%o6,%o7));
```

```
(%o8)                                true
```

Para hacer uso de esta función, ejecútese `load("to_poly_solve")`.

`dfloat` (*x*) [Función]

La función `dfloat` es similar a `float`, pero `dfloat` aplica `rectform` cuando `float` no puede evaluar a un número decimal de coma flotante de doble precisión. Ejemplo:

```
(%i1) float(4.5^(1 + %i));
                                    %i + 1
(%o1)                          4.5
(%i2) dfloat(4.5^(1 + %i));
(%o2)        4.48998802962884 %i + .3000124893895671
```

Notas

- La forma rectangular de una expresión puede no ser la más adecuada para cálculos numéricos
- El identificador `float` es al mismo tiempo una variable opcional, cuyo valor por defecto es `false` y el nombre de una función.

Véanse también `float` y `bfloat`.

Para hacer uso de esta función, ejecútese `load("to_poly_solve")`.

`elim` (*l*, *x*) [Función]

La función `elim` elimina las variables que se indican en el conjunto o lista `x` del conjunto o lista de ecuaciones en `l`. Cada elemento de `x` debe ser un símbolo, mientras que los elementos de `l` pueden ser ecuaciones o expresiones que se suponen igualadas a cero.

La función `elim` devuelve una lista formada por dos listas; la primera está formada por las expresiones con las variables eliminadas y la segunda es la lista de pivotes o, en otras palabras, es la lista de expresiones que `elim` ha utilizado para proceder con la eliminación.

Ejemplo:

Eliminación entre ecuaciones lineales. Eliminando `x` e `y` se obtiene una única ecuación `2 z - 7 = 0`; las ecuaciones `y + 7 = 0` y `z - z + 1 = 1` se han utilizado como pivotes.

```
(%i1) elim(set(x + y + z = 1, x - y  - z = 8, x - z = 1),
           set(x,y));
(%o1)             [[2 z - 7], [y + 7, z - x + 1]]
```

Eliminando las tres variables de estas ecuaciones se triangulariza el sistema lineal:

```
(%i2) elim(set(x + y + z = 1, x - y  - z = 8, x - z = 1),
           set(x,y,z));
(%o2)           [[], [2 z - 7, y + 7, z - x + 1]]
```

Las ecuaciones no necesitan ser lineales:

```
(%i3) elim(set(x^2 - 2 * y^3 = 1,  x - y = 5), [x,y]);
                      3    2
(%o3)       [[], [2 y  - y  - 10 y - 24, y - x + 5]]
```

El usuario no puede controlar el orden en el que se eliminan las variables. El algoritmo utiliza una heurística con la que intenta escoger el mejor pivote y el mejor orden de eliminación.

Notas

- Al contrario que la función relacionada `eliminate`, la función `elim` no llama a la función `solve` cuando el número de ecuaciones iguala al de variables.
- La función `elim` trabaja aplicando resultantes; la variable opcional `resultant` determina qué algoritmo va a utilizar Maxima. Con `sqfr`, Maxima factoriza cada resultante y suprime ceros múltiples.
- `elim` triangulariza un conjunto de ecuaciones polinómicas no lineales; el conjunto solución del conjunto triangularizado puede ser mayor que el conjunto de soluciones del conjunto no triangularizado, por lo que las ecuaciones triangularizadas pueden tener soluciones falsas.

Véanse también `elim_allbut`, `eliminate_using`, `eliminate` y `resultant`.

Para hacer uso de esta función, ejecútese `load("to_poly")`.

`elim_allbut` (*l*, *x*) [Función]

Es similar a `elim`, excepto por el hecho de que elimina todas las variables que aparecen en la lista de ecuaciones `l` que no están en `x`.

Ejemplo:

```
(%i1) elim_allbut([x+y = 1, x - 5*y = 1],[]);
(%o1)                 [[], [y, y + x - 1]]
(%i2) elim_allbut([x+y = 1, x - 5*y = 1],[x]);
(%o2)                [[x - 1], [y + x - 1]]
```

Para hacer uso de esta función, ejecútese `load("to_poly")`.

Véanse también `elim`, `eliminate_using`, `eliminate` y `resultant`.

`eliminate_using` (*l*, *e*, *x*) [Función]

Elmina el símbolo `x` de la lista o conjunto de ecuaciones `l` haciendo uso del pivote `e`.

Ejemplos:

```
(%i1) eq : [x^2 - y^2 - z^3 , x*y - z^2 - 5, x - y + z];
                3    2    2     2
(%o1)      [- z  - y  + x , - z  + x y - 5, z - y + x]
(%i2) eliminate_using(eq,first(eq),z);
        3             2      2      3    2
(%o2) {y  + (1 - 3 x) y  + 3 x  y - x  - x ,
                        4    3  3       2  2             4
                       y  - x  y  + 13 x  y  - 75 x y + x  + 125}
(%i3) eliminate_using(eq,second(eq),z);
       2            2       4    3  3       2  2            4
(%o3) {y  - 3 x y + x  + 5, y  - x  y  + 13 x  y  - 75 x y + x
                                                          + 125}
(%i4) eliminate_using(eq, third(eq),z);
       2            2       3              2      2      3    2
(%o4) {y  - 3 x y + x  + 5, y  + (1 - 3 x) y  + 3 x  y - x  - x }
```

Para hacer uso de esta función, ejecútese `load("to_poly")`.

Véanse también `elim`, `elim_allbut`, `eliminate` y `resultant`.

`fourier_elim` ([*eq1*, *eq2*, ...], [*var1*, *var*, ...]) [Función]

La instrucción `fourier_elim([eq1,eq2,...], [var1,var2,...])` aplica el algoritmo de eliminación de Fourier para resolver el sistema de inecuaciones lineales `[eq1,eq2,...]` respecto de las variables `[var1,var2,...]`.

Ejemplos:

```
(%i1) fourier_elim([y-x < 5, x - y < 7, 10 < y],[x,y]);
(%o1)            [y - 5 < x, x < y + 7, 10 < y]
(%i2) fourier_elim([y-x < 5, x - y < 7, 10 < y],[y,x]);
(%o2)        [max(10, x - 7) < y, y < x + 5, 5 < x]
```

Eliminando primero respecto de x y luego respecto de y, se obtienen límites inferior y superior para x que dependen de y, y límites numéricos para y. Si se eliminan en orden inverso, se obtienen los límites de y en función de x, y los de x son números.

De ser necesario, `fourier_elim` devuelve una disyunción de listas de ecuaciones:

```
(%i3) fourier_elim([x # 6],[x]);
(%o3)                  [x < 6] or [6 < x]
```

Si no existe solución, `fourier_elim` devuelve `emptyset`, y si la solución son todos los reales, `fourier_elim` devuelve `universalset`:

```
(%i4) fourier_elim([x < 1, x > 1],[x]);
(%o4)                        emptyset
(%i5) fourier_elim([minf < x, x < inf],[x]);
(%o5)                      universalset
```

En caso de que las inecuaciones no sean lineales, `fourier_elim` devuelve una lista de inecuaciones simplificadas:

```
(%i6) fourier_elim([x^3 - 1 > 0],[x]);
                 2                                 2
(%o6) [1 < x, x  + x + 1 > 0] or [x < 1, - (x  + x + 1) > 0]
(%i7) fourier_elim([cos(x) < 1/2],[x]);
(%o7)                  [1 - 2 cos(x) > 0]
```

En lugar de una lista de inecuaciones, el primer argumento pasado a `fourier_elim` puede ser una conjunción o disyunción lógica.

```
(%i8) fourier_elim((x + y < 5) and (x - y >8),[x,y]);
                                                 3
(%o8)            [y + 8 < x, x < 5 - y, y < - -]
                                                 2
(%i9) fourier_elim(((x + y < 5) and x < 1) or  (x - y >8),[x,y]);
(%o9)          [y + 8 < x] or [x < min(1, 5 - y)]
```

La función `fourier_elim` soporta los operadores de desigualdad `<`, `<=`, `>`, `>=`, `#` y `=`.

La rutina de eliminación de Fourier dispone de un preprocesador que convierte algunas inecuaciones no lineales formadas con las funciones del valor absoluto, mínimo y máximo a inecuaciones lineales. Además, el preprocesador admite algunas expresiones que son productos o cocientes de términos lineales:

```
(%i10) fourier_elim([max(x,y) > 6, x # 8, abs(y-1) > 12],[x,y]);
```

```
(%o10) [6 < x, x < 8, y < - 11] or [8 < x, y < - 11]
 or [x < 8, 13 < y] or [x = y, 13 < y] or [8 < x, x < y, 13 < y]
 or [y < x, 13 < y]
(%i11) fourier_elim([(x+6)/(x-9) <= 6],[x]);
(%o11)           [x = 12] or [12 < x] or [x < 9]
(%i12) fourier_elim([x^2 - 1 # 0],[x]);
(%o12)      [- 1 < x, x < 1] or [1 < x] or [x < - 1]
```

Para hacer uso de esta función, ejecútese `load("fourier_elim")`.

`isreal_p` (*e*) [Función]

El predicado `isreal_p` devuelve `true` si `e` representa un número real y `false` si no representa un punto de la recta; en cualquier otro caso devuelve una forma nominal.

```
(%i1) map('isreal_p, [-1, 0, %i, %pi]);
(%o1)               [true, true, false, true]
```

Las variables de Maxima se interpretan como números reales:

```
(%i2) isreal_p(x);
(%o2)                          true
```

La función `isreal_p` consulta los hechos almacenados en la base de datos:

```
(%i3) declare(z,complex)$

(%i4) isreal_p(z);
(%o4)                       isreal_p(z)
```

Con frecuencia, `isreal_p` devuelve una forma nominal cuando debería devolver `false`; por ejemplo, la función logarítmica no toma valores reales en toda la recta real, por lo que `isreal_p(log(x))` debería devolver `false`, sin embargo:

```
(%i5) isreal_p(log(x));
(%o5)                    isreal_p(log(x))
```

Para hacer uso de esta función, ejecútese `load("to_poly_solve")`.

Véase también `complex_number_p`.

La función `isreal_p` es experimental; sus especificaciones pueden cambiar y su funcionalidad puede incorporarse a otras funciones de Maxima.

`new_variable` (*type*) [Función]

Devuelve un símbolo de la forma `%[z,n,r,c,g]k`, siendo `k` un número entero. Los valores admisibles para *type* son `integer`, `natural_number`, `real`, `natural_number` y `general`. Por número natural se entiende *entero negativo*, de manera que el ceero es un número natural.

Cuando *type* no es de ninguno de los tipos indicados más arriba, *type* toma por defecto el valor `general`. Para enteros, números naturales y números complejos, Maxima añade esta información a la base de datos de forma automática.

```
(%i1) map('new_variable,
          ['integer, 'natural_number, 'real, 'complex, 'general]);
(%o1)          [%z144, %n145, %r146, %c147, %g148]
(%i2) nicedummies(%);
(%o2)               [%z0, %n0, %r0, %c0, %g0]
```

```
(%i3) featurep(%z0, 'integer);
(%o3)                          true
(%i4) featurep(%n0, 'integer);
(%o4)                          true
(%i5) is(%n0 >= 0);
(%o5)                          true
(%i6) featurep(%c0, 'complex);
(%o6)                          true
```

Es recomendable que al argumento de `new_variable` se le aplique el operador de comilla simple para evitar su evaluación, de esta manera se evitan errores como el siguiente:

```
(%i7) integer : 12$

(%i8) new_variable(integer);
(%o8)                          %g149
(%i9) new_variable('integer);
(%o9)                          %z150
```

Para hacer uso de esta función, ejecútese `load("to_poly_solve")`.

Véase también `nicedummies`.

`nicedummies` [Función]

La función `nicedummies` reescribe los índices, comenzando por cero, de las variables de una expresión qua hayan sido introducidas por `new_variable`:

```
(%i1) new_variable('integer) + 52 * new_variable('integer);
(%o1)                    52 %z136 + %z135
(%i2) new_variable('integer) - new_variable('integer);
(%o2)                      %z137 - %z138
(%i3) nicedummies(%);
(%o3)                        %z0 - %z1
```

Para hacer uso de esta función, ejecútese `load("to_poly_solve")`.

Véase también `new_variable`.

`parg` (*x*) [Función]

La función `parg` es una versión con capacidades simplificadoras de la función de argumento complejo `carg`:

```
(%i1) map('parg,[1,1+%i,%i, -1 + %i, -1]);
                          %pi  %pi  3 %pi
(%o1)                [0, ---, ---, -----, %pi]
                           4    2     4
```

Si el argumento pasado a la función `parg` no es una constante, se devolverá una forma nominal:

```
(%i2) parg(x + %i * sqrt(x));
(%o2)                   parg(x + %i sqrt(x))
```

Si `sign` detecta que la entrada es un número real negativo o positivo, `parg` devuelve una forma no nominal aunque la entrada no sea una constante:

```
(%i3) parg(abs(x));
```

```
(%o3) 0
(%i4) parg(-x^2-1);
(%o4)                          %pi
```

La función `sign` suele ignorar las variables declaradas complejas (`declare(x,complex)`); en tales casos, `parg` puede retornar valores incorrectos:

```
(%i1) declare(x,complex)$

(%i2) parg(x^2 + 1);
(%o2) 0
```

Para hacer uso de esta función, ejecútese `load("to_poly_solve")`.

Véanse también `carg`, `isreal_p`.

`real_imagpart_to_conjugate (`*e*`)` [Función]

La función `real_imagpart_to_conjugate` reemplaza todas las llamadas a `realpart` y `imagpart` presentes en una expresión por llamadas a `conjugate`, obteniendo otra expresión equivalente:

```
(%i1) declare(x, complex)$

(%i2) real_imagpart_to_conjugate(realpart(x) +  imagpart(x) = 3);
          conjugate(x) + x   %i (x - conjugate(x))
(%o2)     ---------------- - ---------------------- = 3
                 2                      2
```

Para hacer uso de esta función, ejecútese `load("to_poly_solve")`.

`rectform_log_if_constant (`*e*`)` [Función]

La función `rectform_log_if_constant` convierte todos los términos de la forma `log(c)` a `rectform(log(c))`, siendo `c` una expresión constante o declarada como tal.

```
(%i1) rectform_log_if_constant(log(1-%i) - log(x - %i));
                                  log(2)   %i %pi
(%o1)            - log(x - %i) + ------ - ------
                                    2        4
(%i2) declare(a,constant, b,constant)$

(%i3) rectform_log_if_constant(log(a + %i*b));
                        2    2
                   log(b  + a )
(%o3)              ------------ + %i atan2(b, a)
                        2
```

Para hacer uso de esta función, ejecútese `load("to_poly_solve")`.

`simp_inequality (`*e*`)` [Función]

La función `simp_inequality` aplica ciertas simplificaciones a conjunciones y disyunciones de inecuaciones.

Limitaciones La función `simp_inequality` está limitada en al menos dos aspectos; en primer lugar, las simplificaciones son locales:

```
(%i1) simp_inequality((x > minf) %and (x < 0));
```

```
(%o2) (x>1) %and (x<1)
```

En segundo lugar, `simp_inequality` no tiene en cuenta los hechos de la base de datos:

```
(%i2) assume(x > 0)$

(%i3) simp_inequality(x > 0);
(%o3)                          x > 0
```

Para hacer uso de esta función, ejecútese `load("to_poly_solve")`.

`standardize_inverse_trig` (*e*) [Función]

Esta función aplica las identidades `cot(x) = atan(1/x)` y `acsc(x) = asin(1/x)` y similares con `asec`, `acoth` y `acsch`. Consúltese Abramowitz y Stegun, ecuaciones 4.4.6 a 4.4.8 y 4.6.4 a 4.6.6.

Para hacer uso de esta función, ejecútese `load("to_poly_solve")`.

`subst_parallel` (*l*, *e*) [Función]

Dada la ecuación o lista de ecuaciones *l* y la expresión *e*, sustituye *en paralelo* en *e* los miembros izquierdos de las ecuaciones por los derechos:

```
(%i1) load("to_poly_solve")$

(%i2) subst_parallel([x=y,y=x], [x,y]);
(%o2)                          [y, x]
```

Compárese el resultado anterior con las sustituciones hechas en serie:

```
(%i3) subst([x=y,y=x],[x,y]);
(%o3)                          [x, x]
```

La función `subst_parallel` es similar a `sublis`, excepto por el hecho de que `subst_parallel` permite la sustitución de expresiones no atómicas:

```
(%i4) subst_parallel([x^2 = a, y = b], x^2 * y);
(%o4)                           a b
(%i5) sublis([x^2 = a, y = b], x^2 * y);

                                                                      2
sublis: left-hand side of equation must be a symbol; found: x
 -- an error. To debug this try: debugmode(true);
```

Las sustituciones hechas por `subst_parallel` son literales, no semánticas, por lo que `subst_parallel` no reconoce que $x * y$ sea una subexpresión de $x^2 * y$:

```
(%i6) subst_parallel([x * y = a], x^2 * y);
                                 2
(%o6)                           x  y
```

La función `subst_parallel` realiza todas las sustituciones antes de proceder a la simplificación, lo que permite sustituciones en expresiones condicionales que podrín producir errores en caso de simplificar antes de sustituir:

```
(%i7) subst_parallel([x = 0], %if(x < 1, 5, log(x)));
(%o7)                            5
(%i8) subst([x = 0], %if(x < 1, 5, log(x)));
```

```
log: encountered log(0).
 -- an error. To debug this try: debugmode(true);
```

Para hacer uso de esta función, ejecútese `load("to_poly_solve_extra.lisp")`.

Véanse también `subst`, `sublis` y `ratsubst`.

`to_poly` (*e*, *l*) [Función]

La función `to_poly` intenta convertir la ecuación *e* en un sistema de polinomios, junto con restricciones en forma de desigualdades. Las soluciones del sistema polinómico que cumplan las restricciones son, a su vez, las soluciones de la ecuación *e*. Dicho de manera informal, `to_poly` intenta pasar a forma de polinomio la ecuación *e*; un ejemplo ayudará a aclarar su comportamiento:

```
(%i1) load("to_poly_solve")$

(%i2) to_poly(sqrt(x) = 3, [x]);
                                2
(%o2) [[%g130 - 3, x = %g130 ],
                        %pi                                     %pi
                    [- --- < parg(%g130), parg(%g130) <= ---], []]
                         2                                       2
```

Las condiciones `-%pi/2<parg(%g130),parg(%g130)<=%pi/2` dicen que `%g130` está en el rango de la función radical; cuando eso se cumpla, el conjunto de ecuaciones de `sqrt(x) = 3` coincide con el de `%g130-3,x=%g130^2`.

Para convertir a forma polinómica una expresión trigonométrica, es necesario introducir una sustitución no algebraica; tal sustitución se devuelve en la tercera lista de la respuesta de `to_poly`:

```
(%i3) to_poly(cos(x),[x]);
                  2                                    %i x
(%o3)    [[%g131  + 1], [2 %g131 # 0], [%g131 = %e    ]]
```

Los términos constantes no se transforman a polinomios a menos que el número uno se introduzca en la lista de variables:

```
(%i4) to_poly(x = sqrt(5),[x]);
(%o4)                [[x - sqrt(5)], [], []]
(%i5) to_poly(x = sqrt(5),[1,x]);
                               2
(%o5) [[x - %g132, 5 = %g132 ],
                        %pi                                     %pi
                    [- --- < parg(%g132), parg(%g132) <= ---], []]
                         2                                       2
```

Para generar un polinomio que tenga $sqrt(5) + sqrt(7)$ como raíz puede hacerse lo siguiente:

```
(%i6) first(elim_allbut(first(to_poly(x = sqrt(5) + sqrt(7),
                                      [1,x])), [x]));
                            4       2
(%o6)                     [x  - 24 x  + 4]
```

Para hacer uso de esta función, ejecútese `load("to_poly")`.

Véase también `to_poly_solve`.

`to_poly_solve` (*e*, *l*, [*options*]) [Función]

La función `to_poly_solve` intenta resolver las ecuaciones *e* de incógnitas *l*. El argumento *e* puede ser una única ecuación, o una lista o conjunto de ecuaciones; de forma similar, *l* puede ser un símbolo o una lista o conjunto de símbolos. Cuando uno de los elementos de *e* no sea una igualdad, como $x^2 - 1$, se supodrá que es igual a cero.

La estrategia básica de `to_poly_solve` consiste en convertir la entrada en un polinomio y luego invocar a la función `algsys`. Internamente, `to_poly_solve` asigna a `algexact` el valor `true`. Para cambiar el valor de `algexact`, debe anñadirse `'algexact=false` a la lista de argumentos de `to_poly_solve`.

Cuando `to_poly_solve` consigue determinar el conjunto de soluciones, cada miembro del conjunto de soluciones es una lista en un objeto `%union`:

```
(%i1) load("to_poly_solve")$

(%i2) to_poly_solve(x*(x-1) = 0, x);
(%o2)               %union([x = 0], [x = 1])
```

Cuando `to_poly_solve` es incapaz de determinar el conjunto de soluciones, devuelve una forma nominal de `%solve` y muestra un mensaje de aviso:

```
(%i3) to_poly_solve(x^k + 2* x + 1 = 0, x);

Nonalgebraic argument given to 'to_poly'
unable to solve
                             k
(%o3)            %solve([x  + 2 x + 1 = 0], [x])
```

A veces se puede obtener la solución haciendo una sustitución en `%solve`:

```
(%i4) subst(k = 2, %);
(%o4)                    %union([x = - 1])
```

Especialmente en el caso de las funciones trigonométricas, los resultados pueden incorporar números enteros arbitrarios de la forma `%zXXX`, siendo `XXX` un índice entero:

```
(%i5) to_poly_solve(sin(x) = 0, x);
(%o5)   %union([x = 2 %pi %z33 + %pi], [x = 2 %pi %z35])
```

Para inicializar los índices, hágase uso de `nicedummies`:

```
(%i6) nicedummies(%);
(%o6)    %union([x = 2 %pi %z0 + %pi], [x = 2 %pi %z1])
```

En ocasiones, se introducen números complejos arbitrarios de la forma `%cXXX`, o reales de la forma `%rXXX`. La función `nicedummies` inicializa estos identificadores a cero.

También a veces, la solución incorpora versiones simplificadas de los operadores lógicos `%and`, `%or` y `%if`, que representan, respectivamente, la conjunción, la disyunción y la implicación:

```
(%i7) sol : to_poly_solve(abs(x) = a, x);
(%o7) %union(%if(isnonnegative_p(a), [x = - a], %union()),
                        %if(isnonnegative_p(a), [x = a], %union()))
(%i8) subst(a = 42, sol);
```

```
(%o8)             %union([x = - 42], [x = 42])
(%i9) subst(a = -42, sol);
(%o9)                        %union()
```

El conjunto vacío se representa por `%union`.

La función `to_poly_solve` es capaz de resolver algunas ecuaciones con potencias racionales, potencias no racionales, valores absolutos, funciones trigonométricas y funciones del mínimo y del máximo. También puede resolver algunas ecuaciones resolubles en términos de la función W de Lambert:

```
(%i1) load("to_poly_solve")$

(%i2) to_poly_solve(set(max(x,y) = 5, x+y = 2), set(x,y));
(%o2)      %union([x = - 3, y = 5], [x = 5, y = - 3])
(%i3) to_poly_solve(abs(1-abs(1-x)) = 10,x);
(%o3)             %union([x = - 10], [x = 12])
(%i4) to_poly_solve(set(sqrt(x) + sqrt(y) = 5, x + y = 10),
                    set(x,y));
                     3/2              3/2
                    5    %i - 10     5    %i + 10
(%o4) %union([x = - ------------, y = ------------],
                         2                2
                               3/2                3/2
                              5    %i + 10       5    %i - 10
                       [x = ------------, y = - ------------])
                                   2                  2
(%i5) to_poly_solve(cos(x) * sin(x) = 1/2,x,
                    'simpfuncs = ['expand, 'nicedummies]);
                                        %pi
(%o5)              %union([x = %pi %z0 + ---])
                                         4
(%i6) to_poly_solve(x^(2*a) + x^a + 1,x);
                                       2 %i %pi %z81
                                       -------------
                                  1/a         a
                    (sqrt(3) %i - 1)    %e
(%o6) %union([x = -----------------------------------],
                                  1/a
                                 2
                                                    2 %i %pi %z83
                                                    -------------
                                              1/a         a
                          (- sqrt(3) %i - 1)    %e
                   [x = -----------------------------------])
                                               1/a
                                              2
(%i7) to_poly_solve(x * exp(x) = a, x);
(%o7)              %union([x = lambert_w(a)])
```

En el caso de inecuaciones lineales, `to_poly_solve` aplica automáticamente la eliminación de Fourier:

```
(%i8) to_poly_solve([x + y < 1, x - y >= 8], [x,y]);
                                 7
(%o8) %union([x = y + 8, y < - -],
                                 2
                                                                  7
                                   [y + 8 < x, x < 1 - y, y < - -])
                                                                  2
```

Los argumentos opcionales deben tener forma de ecuación; generalmente, el orden de estas opciones no reviste importancia.

- `simpfuncs = l`, siendo `l` una lista de funciones, aplica la composición de los elementos de `l` a cada solución:

```
(%i1) to_poly_solve(x^2=%i,x);
                                  1/4             1/4
(%o1)       %union([x = - (- 1)   ], [x = (- 1)   ])
(%i2) to_poly_solve(x^2= %i,x, 'simpfuncs = ['rectform]);
                       %i         1             %i         1
(%o2) %union([x = - ------- - -------], [x = ------- + -------])
                    sqrt(2)   sqrt(2)        sqrt(2)   sqrt(2)
```

  A veces, una simplificación puede anular una simplificación anterior:

```
(%i3) to_poly_solve(x^2=1,x);
(%o3)              %union([x = - 1], [x = 1])
(%i4) to_poly_solve(x^2= 1,x, 'simpfuncs = [polarform]);
                                           %i %pi
(%o4)            %union([x = 1], [x = %e      ])
```

  Maxima no comprueba que los elementos de la lista de funciones `l` sean todos simplificaciones:

```
(%i5) to_poly_solve(x^2 = %i,x, 'simpfuncs = [lambda([s],s^2)]);
(%o5)                   %union([x = %i])
```

  Para convertir cada solución a real de doble precisión hágase uso de `simpfunc = ['dfloat]`:

```
(%i6) to_poly_solve(x^3 +x + 1 = 0,x,
                    'simpfuncs = ['dfloat]), algexact : true;
(%o6) %union([x = - .6823278038280178],
[x = .3411639019140089 - 1.161541399997251 %i],
[x = 1.161541399997251 %i + .3411639019140089])
```

- Con la opción `use_grobner = true` se aplica la función `poly_reduced_grobner` a las ecuaciones antes de intentar resolverlas. En primer lugar, esta opción proporciona una manera de soslayar algunas debilidades de la función `algsys`:

```
(%i7) to_poly_solve([x^2+y^2=2^2,(x-1)^2+(y-1)^2=2^2],[x,y],
                    'use_grobner = true);
                    sqrt(7) - 1      sqrt(7) + 1
(%o7) %union([x = - -----------, y = -----------],
```

```
                                 2                2
                                        sqrt(7) + 1        sqrt(7) - 1
                                   [x = -----------, y = - -----------])
                                              2                  2
(%i8) to_poly_solve([x^2+y^2=2^2,(x-1)^2+(y-1)^2=2^2],[x,y]);
(%o8)                          %union()
```

- `maxdepth = k`, siendo `k` un positivo entero, controla el nivel de recursión. El valor por defecto es cinco. Cuando se excede el nivel de recursión se obtiene un mensaje de error:

```
(%i9) to_poly_solve(cos(x) = x,x, 'maxdepth = 2);

Unable to solve
Unable to solve
(%o9)        %solve([cos(x) = x], [x], maxdepth = 2)
```

- Con `parameters = l`, siendo `l` una lista de símbolos, el programa intenta encontrar una solución válida para todos los miembros de la lista `l`:

```
(%i10) to_poly_solve(a * x = x, x);
(%o10)                    %union([x = 0])
(%i11) to_poly_solve(a * x = x, x, 'parameters = [a]);
(%o11) %union(%if(a - 1 = 0, [x = %c111], %union()),
                                  %if(a - 1 # 0, [x = 0], %union()))
```

  En `(%o2)`, el programa introduce una variable ficticia; para reinicializarla, úsese la función `nicedummies`:

```
(%i12) nicedummies(%);
(%o12) %union(%if(a - 1 = 0, [x = %c0], %union()),
                                  %if(a - 1 # 0, [x = 0], %union()))
```

`to_poly_solve` utiliza información almacenada en el array `one_to_one_reduce` para resolver ecuaciones de la forma $f(a) = f(b)$. La asignación `one_to_one_reduce['f,'f] : lambda([a,b], a=b)` le dice a `to_poly_solve` que el conjunto de soluciones de $f(a) = f(b)$ es igual al conjunto de soluciones de $a = b$:

```
(%i13) one_to_one_reduce['f,'f] : lambda([a,b], a=b)$

(%i14) to_poly_solve(f(x^2-1) = f(0),x);
(%o14)             %union([x = - 1], [x = 1])
```

De forma más general, la asignación `one_to_one_reduce['f,'g] : lambda([a,b], w(a,b)=0` le indica a `to_poly_solve` que el cojunto de soluciones de $f(a) = f(b)$ es igual al conjunto de soluciones de $w(a, b) = 0$:

```
(%i15) one_to_one_reduce['f,'g] : lambda([a,b], a = 1 + b/2)$

(%i16) to_poly_solve(f(x) - g(x),x);
(%o16)                   %union([x = 2])
```

Además, `to_poly_solve` utiliza información almacenada en el array `function_inverse` para resolver ecuaciones de la forma $f(a) = b$. La asignación

`function_inverse['f] : lambda([s], g(s))` le dice a `to_poly_solve` que el conjunto de soluciones de $f(x) = b$ es igual al conjunto de soluciones de $x = g(b)$:

```
(%i17) function_inverse['Q] : lambda([s], P(s))$

(%i18) to_poly_solve(Q(x-1) = 2009,x);
(%o18)              %union([x = P(2009) + 1])
(%i19) function_inverse['G] : lambda([s], s+new_variable(integer));
(%o19)       lambda([s], s + new_variable(integer))
(%i20) to_poly_solve(G(x - a) = b,x);
(%o20)             %union([x = b + a + %z125])
```

Notas

- Las incógnitas a resolver no necesitan ser símbolos, lo cual es cierto cuando `fullratsubst` es capaz de hacer las sustituciones de forma apropiadas:

```
(%i1) to_poly_solve([x^2 + y^2 + x * y = 5, x * y = 8],
                    [x^2 + y^2, x * y]);
                                      2    2
(%o1)           %union([x y = 8, y  + x  = - 3])
```

- Cuando las ecuaciones involucran conjugados de complejos, el programa añade automáticamente las ecuaciones conjugadas:

```
(%i1) declare(x,complex)$

(%i2) to_poly_solve(x + (5 + %i) * conjugate(x) = 1, x);
                                     %i + 21
(%o2)              %union([x = - -----------])
                                   25 %i - 125
(%i3) declare(y,complex)$

(%i4) to_poly_solve(set(conjugate(x) - y = 42 + %i,
                        x + conjugate(y) = 0), set(x,y));
                           %i - 42        %i + 42
(%o4)        %union([x = - -------, y = - -------])
                                2              2
```

- Cuando las funciones involucran valores absolutos, `to_poly_solve` consulta los hechos de la base de datos para decidir si los argumentos de los valores absolutos son números complejos:

```
(%i1) to_poly_solve(abs(x) = 6, x);
(%o1)              %union([x = - 6], [x = 6])
(%i2) declare(z,complex)$

(%i3) to_poly_solve(abs(z) = 6, z);
(%o3) %union(%if((%c11 # 0) %and (%c11 conjugate(%c11) - 36 =
                                        0), [z = %c11], %union()))
```

  Esta es la única situación en la que `to_poly_solve` consulta la base de datos; si una incógnita se declara, por ejemplo, como entero, `to_poly_solve` lo ignora.

Para hacer uso de esta función, ejecútese `load("to_poly_solve")`.

Véase también `algexact`, `resultant`, `algebraic` y `to_poly`.

78 unit

78.1 Introducción a units

El paquete `unit` permite al usuario hacer cambios de unidades y llevar a cabo el análisis dimensional de las ecuaciones. La forma de operar de este paquete es radicalmente diferente de la del paquete original de Maxima; mientras que en el paquete original era tan solo una lista de definiciones, aquí se utiliza un conjunto de reglas que permiten seleccionar al usuario en qué unidades debe devolverse la expresión final.

Junto con el análisis dimensional, el paquete aporta una serie de herramientas para controlar las opciones de conversión y simplificación. Además de la conversión automática adaptable a las necesidades del usuario, el paquete `unit` permite hacer conversiones a la manera tradicional.

Nota: Cuando los factores de conversión no son exactos, Maxima los transformará a fracciones como consecuencia de la metodología utilizada para simplificar las unidades. Los mensajes de aviso concernientes a estas transformaciones están desactivados por defecto en el caso de las unidades (lo habitual es que estén activados en otros contextos) debido a que al ser una operación muy frecuente, serían un estorbo. El estado previo de la variable `ratprint` queda restaurado tras la conversión de las unidades, de manera que se mantendrá la opción seleccionada por el usuario; en caso de que éste necesite ver dichos avisos, podrá hacer la asignación `unitverbose:on` para reactivarlos desde el proceso de conversión de unidades.

El paquete `unit` se aloja en el directorio `share/contrib/unit` y se ajusta a las convenciones de Maxima para la carga de paquetes:

```
(%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.
```

Los avisos del tipo `WARNING` son normales y no deben interpretarse como errores; tan solo indican que el paquete `unit` está redefiniendo funciones que ya estaban definidas en Maxima. Esto es necesario para que las unidades se gestionen de forma correcta. El usuario debe tener en cuenta que si otros paquetes han cambiado las definiciones de estas funciones, tales cambios serán ignorados por el proceso de carga de `unit`.

El paquete `unit` también carga el fichero de Lisp `unit-functions.lisp`, el cual contiene las funciones Lisp necesarias.

El autor principal de este paquete es Clifford Yapp, quien ha recibido ayuda y asistencia, entre otros, de Barton Willis y Robert Dodier.

78.2 Funciones y variables para units

`setunits` (*list*) [Función]

El paquete `unit` no utiliza por defecto dimensiones derivadas, pero convierte todas las unidades a las siete fundamentales en unidades MKS.

```
(%i2) N;
                                      kg m
(%o2)                                 ----
                                        2
                                       s
(%i3) dyn;
                                    1      kg m
(%o3)                             (------) (----)
                                  100000     2
                                             s
(%i4) g;
                                     1
(%o4)                               (----) (kg)
                                    1000
(%i5) centigram*inch/minutes^2;
                                   127         kg m
(%o5)                        (-------------) (----)
                              1800000000000      2
                                                 s
```

Este es el comportamiento que se desea en ciertos casos. Si el usuario necesita utilizar otras unidades, habrá de utilizar la instrucción `setunits`:

```
(%i6) setunits([centigram,inch,minute]);
(%o6)                                done
(%i7) N;
                             1800000000000   %in cg
(%o7)                       (-------------) (------)
                                  127            2
                                               %min
(%i8) dyn;
                                 18000000   %in cg
(%o8)                           (--------) (------)
                                   127          2
                                              %min
(%i9) g;
(%o9)                              (100) (cg)
```

```
(%i10) centigram*inch/minutes^2;
                                      %in cg
(%o10)                                ------
                                          2
                                      %min
```

La especificación de las variables es relativamente flexible. Por ejemplo, si se quiere volver a utilizar kilogramos, metros y segundos como unidades por defecto, podemos hacer:

```
(%i11) setunits([kg,m,s]);
(%o11)                                done
(%i12) centigram*inch/minutes^2;
                                    127        kg m
(%o12)                        (-------------) (----)
                               1800000000000     2
                                                 s
```

Las unidades derivadas también se controlan con esta misma instrucción:

```
(%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
```

Téngase en cuenta que el paquete `unit` reconoce que la combinación de masa, longitud e inversa del cuadrado del tiempo da lugar a una fuerza, convirtiéndola a newtons. Esta es la forma general en la que trabaja Maxima. Si el usuario prefiere dinas a newtons, tan solo tendrá que hacer lo siguiente:

```
(%i22) setunits(dyn);
(%o22)                                done
(%i23) kg*m/s^2;
(%o23)                            (100000) (dyn)
(%i24) centigram*inch/minutes^2;
                                      127
(%o24)                            (--------) (dyn)
                                   18000000
```

Para desactivar una unidad se utiliza la instrucción `uforget`:

```
(%i26) uforget(dyn);
(%o26)                                false
```

```
(%i27) kg*m/s^2;
                                      kg m
(%o27)                                ----
                                        2
                                       s
(%i28) centigram*inch/minutes^2;
                                  127        kg m
(%o28)                       (-------------) (----)
                              1800000000000     2
                                               s
```

Esto también hubiese funcionado con `uforget(N)` o `uforget(%force)`.

Véase también `uforget`. Para hacer uso de esta función ejecútese `load("unit")`.

`uforget` (*list*) [Función]

Por defecto, el paquete `unit` convierte todas las unidades a las siete fundamentales del sistema MKS. Este comportamiento puede alterarse mediante la instrucción `setunits`. Después, si el usuario quiere restaurar el comportamiento por defecto podrá hacerlo para una dimensión determinada haciendo uso de la instrucción `uforget`:

```
(%i13) setunits([centigram,inch,minute]);
(%o13)                                done
(%i14) centigram*inch/minutes^2;
                                      %in cg
(%o14)                                ------
                                          2
                                      %min
(%i15) uforget([cg,%in,%min]);
(%o15)                      [false, false, false]
(%i16) centigram*inch/minutes^2;
                                  127        kg m
(%o16)                       (-------------) (----)
                              1800000000000     2
                                               s
```

La instrucción `uforget` opera sobre dimensiones, no sobre unidades, de modo que valdrá para cualquier unidad de una dimensión concreta. La propia dimensión es una argumento válido para esta función.

Véase también `setunits`. Para hacer uso de esta función ejecútese `load("unit")`.

`convert` (*expr*, *list*) [Función]

La función `convert` permite conversiones de una sola vez sin alterar el entorno global de ejecución. Acepta tanto un único argumento como una lista de unidades a utilizar en las conversiones. Cuando se realiza una llamada a `convert` se ignora el sistema global de evaluación, con el fin de evitar que el resultado deseado sea nuevamente transformado. Como consecuencia de esto, en los cálculos con decimales, los avisos de tipo `rat` se harán visibles si la variable global `ratprint` vale `true`. Otra propiedad de `convert` es que permite al usuario hacer conversiones al sistema fundamental de dimensiones incluso cuando el entorno ha sido ajustado para simplificar a una dimensión derivada.

```
(%i2) kg*m/s^2;
                                      kg m
(%o2)                                 ----
                                        2
                                       s
(%i3) convert(kg*m/s^2,[g,km,s]);
                                      g km
(%o3)                                 ----
                                        2
                                       s
(%i4) convert(kg*m/s^2,[g,inch,minute]);

'rat' replaced 39.37007874015748 by 5000/127 = 39.37007874015748
                              18000000000   %in g
(%o4)                        (-----------) (-----)
                                  127           2
                                             %min
(%i5) convert(kg*m/s^2,[N]);
(%o5)                                  N
(%i6) convert(kg*m^2/s^2,[N]);
(%o6)                                 m N
(%i7) setunits([N,J]);
(%o7)                                 done
(%i8) convert(kg*m^2/s^2,[N]);
(%o8)                                 m N
(%i9) convert(kg*m^2/s^2,[N,inch]);

'rat' replaced 39.37007874015748 by 5000/127 = 39.37007874015748
                                 5000
(%o9)                           (----) (%in N)
                                 127
(%i10) convert(kg*m^2/s^2,[J]);
(%o10)                                 J
(%i11) kg*m^2/s^2;
(%o11)                                 J
(%i12) setunits([g,inch,s]);
(%o12)                                done
(%i13) kg*m/s^2;
(%o13)                                 N
(%i14) uforget(N);
(%o14)                                false
(%i15) kg*m/s^2;
                                 5000000   %in g
(%o15)                          (-------) (-----)
                                   127       2
                                            s
```

```
(%i16) convert(kg*m/s^2,[g,inch,s]);

'rat' replaced 39.37007874015748 by 5000/127 = 39.37007874015748
                                  5000000   %in g
(%o16)                           (-------) (-----)
                                    127       2
                                             s
```

Véanse también `setunits` y `uforget`. Para hacer uso de esta función ejecútese `load("unit")`.

`usersetunits` [Variable opcional]

Valor por defecto: ninguno

En caso de que el usuario desee que el comportamiento por defecto del paquete `unit` sea distinto del descrito, puede hacer uso del fichero `maxima-init.mac` y de la variable global `usersetunits`. El paquete `unit` comprobará al ser cargado si se le ha dado a esta variable una lista de unidades; en caso afirmativo, aplicará `setunits` a las unidades de esta lista y las utilizará por defecto. Una llamada a la función `uforget` permitirá retornar al comportamiento establecido por defecto por el usuario. Por ejemplo, si en el archivo `maxima-init.mac` se tiene el siguiente código:

```
usersetunits : [N,J];
```

observaríamos el siguiente comportamiento:

```
(%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)
```

De no haber hecho uso de `usersetunits`, las entradas iniciales hubiesen sido convertidas a unidades MKS y cualquier llamada a `uforget` hubiese retornado también a MKS. Sin embargo, las preferencias establecidas por el usuario se respetan en ambos casos. Para eliminar las preferencias del usuario y volver a utilizar las establecidas por defecto por el paquete `unit`, debe utilizarse la instrucción `dontusedimension`. La función `uforget` puede restaurar nuevamente las preferencias del usuario, pero sólo si `usedimension` mantiene su valor. Alternativamente, `kill(usersetunits)` eliminará completamente cualquier vestigio de las preferencias del usuario durante la sesión actual. Véanse a continuación algunos ejemplos de aplicación de estas opciones:

```
(%i2) kg*m/s^2;
(%o2)                                     N
(%i3) kg*m^2/s^2;
(%o3)                                     J
(%i4) setunits([dyn,eV]);
(%o4)                                   done
(%i5) kg*m/s^2;
(%o5)                              (100000) (dyn)
(%i6) kg*m^2/s^2;
(%o6)                       (6241509596477042688) (eV)
(%i7) uforget([dyn,eV]);
(%o7)                            [false, false]
(%i8) kg*m/s^2;
(%o8)                                     N
```

```
(%i9) kg*m^2/s^2;
(%o9)                                  J
(%i10) dontusedimension(N);
(%o10)                               [%force]
(%i11) dontusedimension(J);
(%o11)                          [%energy, %force]
(%i12) kg*m/s^2;
                                       kg m
(%o12)                                 ----
                                         2
                                        s
(%i13) kg*m^2/s^2;
                                           2
                                       kg m
(%o13)                                 -----
                                         2
                                        s
(%i14) setunits([dyn,eV]);
(%o14)                                 done
(%i15) kg*m/s^2;
                                       kg m
(%o15)                                 ----
                                         2
                                        s
(%i16) kg*m^2/s^2;
                                           2
                                       kg m
(%o16)                                 -----
                                         2
                                        s
(%i17) uforget([dyn,eV]);
(%o17)                          [false, false]
(%i18) kg*m/s^2;
                                       kg m
(%o18)                                 ----
                                         2
                                        s
(%i19) kg*m^2/s^2;
                                           2
                                       kg m
(%o19)                                 -----
                                         2
                                        s
(%i20) usedimension(N);
Done.  To have Maxima simplify to this dimension, use
setunits([unit]) to select a unit.
(%o20)                                 true
```

```
(%i21) usedimension(J);
Done.  To have Maxima simplify to this dimension, use
setunits([unit]) to select a unit.
(%o21)                                true
(%i22) kg*m/s^2;
                                      kg m
(%o22)                                ----
                                        2
                                       s
(%i23) kg*m^2/s^2;
                                         2
                                      kg m
(%o23)                                -----
                                        2
                                       s
(%i24) setunits([dyn,eV]);
(%o24)                                done
(%i25) kg*m/s^2;
(%o25)                          (100000) (dyn)
(%i26) kg*m^2/s^2;
(%o26)                  (6241509596477042688) (eV)
(%i27) uforget([dyn,eV]);
(%o27)                           [false, false]
(%i28) kg*m/s^2;
(%o28)                                 N
(%i29) kg*m^2/s^2;
(%o29)                                 J
(%i30) kill(usersetunits);
(%o30)                                done
(%i31) uforget([dyn,eV]);
(%o31)                          [false, false]
(%i32) kg*m/s^2;
                                      kg m
(%o32)                                ----
                                        2
                                       s
(%i33) kg*m^2/s^2;
                                         2
                                      kg m
(%o33)                                -----
                                        2
                                       s
```

Desafortunadamente, esta amplia variedad de opciones puede resultar confusa en un primer momento, pero una vez se practica un poco con ellas, el usuario comprobará que tiene un control absoluto sobre su entorno de trabajo.

`metricexpandall (x)` [Función]

Reconstruye automáticamente las listas globales de unidades creando todas los múltiplos y submúltiplos métricos necesarios. El argumento numérico x se utiliza para especificar cuántos prefijos numéricos quiere utilizar el usuario. Los argumentos son los siguientes:

```
                0 - none. Only base units
                1 - kilo, centi, milli
(por defecto)   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 - todos
```

Normalmente, Maxima no definirá el juego completo de múltiplos y submúltiplos, lo que implica un número muy grande de unidades, pero `metricexpandall` puede utilizarse para reconstruir la lista. La variable fundamental del paquete `unit` es `%unitexpand`.

`%unitexpand` [Variable opcional]

Valor por defecto: `2`

Es el valor suministrado a `metricexpandall` durante la carga del paquete `unit`.

79 zeilberger

79.1 Introducción a zeilberger

El paquete `zeilberger` implementa el algoritmo de Zeilberger para la suma hipergeométrica definida y el algoritmo de Gosper para la suma hipergeométrica indefinida. Además, hace uso del método de optimización por filtrado desarrollado por Axel Riese.

El autor de este paquete es Fabrizio Caruso.

Antes de hacer uso de las funciones aquí definidas, ejecútese la sentencia `load ("zeilberger")`.

79.1.1 El problema de la suma indefinida

El paquete `zeilberger` implementa el algoritmo de Gosper para la suma hipergeométrica indefinida. Dado el término general hipergeométrico F_k de índice k, se plantea el problema de encontrar su antidiferencia hipergeométrica, esto es, el término hipergeométrico tal que

$$F_k = f_{k+1} - f_k.$$

79.1.2 El problema de la suma definida

El paquete `zeilberger` implementa el algoritmo de Zeilberger para la suma hipergeométrica definida. Dados el término hipergeométrico propio $F_(n, k)$, de índices n y k, y el entero positivo d, se plantea el problema de encontrar una expresión recurrente lineal de orden d con coeficientes polinomiales en n y una función racional R en n y k tales que

$a_0 F_(n, k) + ... + a_d F_(n + d), k = Delta_K(R(n, k) F_(n, k))$

donde $Delta_k$ es el k-ésimo operador diferencia hacia adelante, esto es, $Delta_k(t_k) := t_(k + 1) - t_k$.

79.1.3 Niveles de información

Hay versiones extendidas de los nombres de las instrucciones, que se construyen añadiendo uno de los siguientes prefijos:

`Summary` Tan solo muestra un sumario al final

`Verbose` Alguna información en los niveles intermedios

`VeryVerbose`
Más información

`Extra` Aún más información, incluida alguna sobre el sistema lineal en el algoritmo de Zeilberger.

Por ejemplo: `GosperVerbose`, `parGosperVeryVerbose`, `ZeilbergerExtra`, `AntiDifferenceSummary`.

79.2 Funciones y variables para zeilberger

`AntiDifference` (*F_k*, *k*) [Función]

Returns the hypergeometric anti-difference of F_k, if it exists. Otherwise `AntiDifference` returns `no_hyp_antidifference`.

`Gosper` (*F_k*, *k*) [Función]

Devuelve, si existe, el elemento racional asociado a *F_k*, esto es, la función racional que verifica

$F_k = R(k+1)\ F_{k+1} - R(k)\ F_k$,

En caso de no existir este elemento, `Gosper` devuelve `no_hyp_sol`.

`GosperSum` (*F_k*, *k*, *a*, *b*) [Función]

Devuelve la suma de los términos F_k desde $k = a$ hasta $k = b$ si F_k tiene una antidiferencia hipergeométrica. En caso contrario, `GosperSum` devuelve `nongosper_summable`.

Ejemplos:

```
(%i1) load ("zeilberger")$
(%i2) GosperSum ((-1)^k*k / (4*k^2 - 1), k, 1, n);
Dependent equations eliminated:  (1)
                            3       n + 1
                       (n + -) (- 1)
                            2                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)                    NON_GOSPER_SUMMABLE
```

`parGosper` (*`F_{n,k}`*, *`k`*, *`n`*, *`d`*) [Función]

Intenta calcular una recurrecia de orden d para $F\_\{n,k\}$.

El algoritmo devuelve una secuencia $[s_1, s_2, ..., s_m]$ de soluciones, cada una de las cuales tiene la forma

$[R(n,k), [a_0, a_1, ..., a_d]]$.

La función `parGosper` devuelve `[]` si no encuentra ninguna recurrencia.

`Zeilberger` (*`F_{n,k}`*, *`k`*, *`n`*) [Función]

Intenta calcular la suma hipergeométrica indefinida de $F\_\{n,k\}$.

La función `Zeilberger` invoca en primer lugar a `Gosper`, y en caso de no encontrar una solución, llama después a `parGosper` con los órdenes 1, 2, 3, ..., hasta `max_ord`. Si `Zeilberger` encuentra una solución antes de alcanzar `max_ord`, se detiene su ejecución y devuelve el resultado.

El algoritmo devuelve una secuencia $[s_1, s_2, ..., s_m]$ de soluciones, cada una de las cuales tiene la forma

$[R(n,k), [a_0, a_1, ..., a_d]]$.

La función `Zeilberger` devuelve `[]` si no encuentra ninguna solución.

La función `Zeilberger` llama a `Gosper` sólo si `Gosper_in_Zeilberger` tiene el valor `true`.

`max_ord` [Variable opcional]

Valor por defecto: 5

`max_ord` es el máximo orden de recurrencia que ensayará la función `Zeilberger`.

`simplified_output` [Variable opcional]

Valor por defecto: `false`

Si `simplified_output` vale `true`, las funciones del paquete `zeilberger` tratan de presentar las soluciones simplificadas.

`linear_solver` [Variable opcional]

Valor por defecto: `linsolve`

La variable `linear_solver` guarda el nombre de la función que se utilizará para resolver el sistema de ecuaciones del algoritmo de Zeilberger.

`warnings` [Variable opcional]

Valor por defecto: `true`

Si `warnings` vale `true`, las funciones del paquete `zeilberger` emiten mensajes de aviso durante su ejecución.

`Gosper_in_Zeilberger` [Variable opcional]

Valor por defecto: `true`

Si `Gosper_in_Zeilberger` vale `true`, la función `Zeilberger` llama a la función `Gosper` antes de llamar a `parGosper`. En caso contrario, `Zeilberger` invoca inmediatamente a `parGosper`.

`trivial_solutions` [Variable opcional]

Valor por defecto: `true`

Si `trivial_solutions` vale `true`, la función `Zeilberger` devuelve soluciones triviales.

`mod_test` [Variable opcional]

Valor por defecto: `false`

Si `mod_test` vale `true`, la función `parGosper` ejecuta una prueba modular para descartar sistemas sin soluciones.

`modular_linear_solver` [Variable opcional]

Valor por defecto: `linsolve`

La variable `modular_linear_solver` guarda el nombre de la función que deberá ser llamada por la prueba modular de `parGosper` para resolver sistemas lineales.

`ev_point` [Variable opcional]

Valor por defecto: `big_primes[10]`

La variable `ev_point` guarda el valor para el que debe evaluarse n durante la ejecución de la prueba modular de `parGosper`.

`mod_big_prime` [Variable opcional]

Valor por defecto: `big_primes[1]`

La variable `mod_big_prime` guarda el módulo utilizado por la prueba modular de `parGosper`.

`mod_threshold` [Variable opcional]

Valor por defecto: 4

La variable `mod_threshold` es el máximo orden que ensaya la prueba modular de `parGosper`.

Apéndice A Índice de Funciones y Variables

!

#

%

,

*

+

—

.

/

:

<

=

>

?

[

]

^

_

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