

Ver. 5.19

Manual de Maxima

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.

Notas de la traducción española.

La versión española del manual de Maxima la mantienen Mario Rodríguez y Jaime Villate; para comentarios, sugerencias y colaboraciones, contáctese mediante cualquiera de las siguientes direcciones de correo electrónico:

- mario ARROBA edu PUNTO xunta PUNTO es
- villate ARROBA fe PUNTO up PUNTO pt

Resumen del Contenido

1	Introducción a Maxima	1
2	Detección e informe de fallos	5
3	Ayuda	7
4	Línea de Comandos	15
5	Operadores	31
6	Expresiones	63
7	Simplificación	99
8	Gráficos	107
9	Lectura y escritura	129
10	Aritmética de punto flotante	159
11	Contextos	161
12	Polinomios	167
13	Constantes	189
14	Logaritmos	193
15	Trigonometría	197
16	Funciones Especiales	205
17	Funciones elípticas	225
18	Límites	231
19	Diferenciación	233
20	Integración	245
21	Ecuaciones	265
22	Ecuaciones Diferenciales	283
23	Métodos numéricos	287
24	Arrays	297
25	Matrices y Álgebra Lineal	305
26	Afines	329
27	itensor	333
28	ctensor	367
29	atensor	393
30	Series	397
31	Teoría de Números	411
32	Simetrías	419
33	Grupos	435
34	Entorno de Ejecución	437
35	Misclánea de opciones	441

36	Reglas y patrones	449
37	Listas	467
38	Conjuntos	475
39	Definición de Funciones	503
40	Programación	531
41	Depurado	543
42	augmented_lagrangian	551
43	bode	553
44	contrib_ode	555
45	descriptive	563
46	diag	585
47	distrib	593
48	draw	629
49	dynamics	687
50	ezunits	697
51	f90	713
52	ggf	715
53	graphs	717
54	grobner	745
55	impdiff	753
56	interpol	755
57	lapack	761
58	lbfgs	765
59	lindstedt	771
60	linearalgebra	773
61	lsquares	787
62	makeOrders	797
63	mnewton	799
64	numericalio	801
65	opsubst	807
66	orthopoly	809
67	plotdf	821
68	romberg	827
69	simplex	831
70	simplification	833
71	solve_rec	843
72	stats	849
73	stirling	867

74	stringproc	869
75	unit	881
76	zeilberger	891
77	Índice de Funciones y Variables	895
A	Índice de Funciones y Variables	897

Tabla de Contenidos

1	Introducción a Maxima	1
2	Detección e informe de fallos	5
2.1	Funciones y variables para la detección e informe de fallos ..	5
3	Ayuda	7
3.1	Lisp y Maxima.....	7
3.2	Recolector de basura	8
3.3	Documentación	8
3.4	Funciones y variables para la ayuda.....	9
4	Línea de Comandos	15
4.1	Introducción a la Línea de Comandos	15
4.2	Funciones y variables para la Línea de Comandos	19
5	Operadores	31
5.1	n-arios	31
5.2	no-fijos	31
5.3	postfijos	31
5.4	prefijos	31
5.5	Operadores aritméticos	32
5.6	Operadores relacionales	35
5.7	Operadores generales	36
6	Expresiones	63
6.1	Introducción a las expresiones	63
6.2	Expresiones complejas	63
6.3	Nombres y verbos	64
6.4	Identificadores	65
6.5	Cadenas de caracteres	66
6.6	Desigualdades	66
6.7	Sintaxis	67
6.8	Funciones y variables para expresiones	69
7	Simplificación	99
7.1	Funciones y variables para simplificación	99
8	Gráficos	107
8.1	Funciones y variables para gráficos	107

9	Lectura y escritura	129
9.1	Comentarios	129
9.2	Archivos	129
9.3	Funciones y variables para lectura y escritura	129
10	Aritmética de punto flotante	159
10.1	Funciones y variables para la aritmética de punto flotante	159
11	Contextos	161
11.1	Funciones y variables para Contextos	161
12	Polinomios	167
12.1	Introducción a los polinomios	167
12.2	Funciones y variables para polinomios	167
13	Constantes	189
13.1	Funciones y variables para Constantes	189
14	Logaritmos	193
14.1	Funciones y variables para logaritmos	193
15	Trigonometría	197
15.1	Introducción a la trigonometría	197
15.2	Funciones y variables para trigonometría	197
16	Funciones Especiales	205
16.1	Introducción a las funciones especiales	205
16.2	Funciones de Bessel	205
16.3	Funciones de Airy	207
16.4	Funciones Gamma y factorial	208
16.5	Integral exponencial	219
16.6	Función de error	220
16.7	Funciones de Struve	221
16.8	Funciones hipergeométricas	221
16.9	Funciones de cilindro parabólico	221
16.10	Funciones y variables para las funciones especiales	222
17	Funciones elípticas	225
17.1	Introducción a las funciones e integrales elípticas	225
17.2	Funciones y variables para funciones elípticas	226
17.3	Funciones y variables para integrales elípticas	228
18	Límites	231
18.1	Funciones y variables para límites	231

19 Diferenciación	233
19.1 Funciones y variables para la diferenciación	233
20 Integración	245
20.1 Introducción a la integración	245
20.2 Funciones y variables para integración	245
20.3 Introducción a QUADPACK	254
20.3.1 Perspectiva general	254
20.4 Funciones y variables para QUADPACK	255
21 Ecuaciones	265
21.1 Funciones y variable para las ecuaciones	265
22 Ecuaciones Diferenciales	283
22.1 Introducción a las ecuaciones diferenciales	283
22.2 Funciones y variables para ecuaciones diferenciales.	283
23 Métodos numéricos	287
23.1 Introducción a los métodos numéricos	287
23.2 Series de Fourier	287
23.3 Funciones y variables para los métodos numéricos	287
23.4 Funciones y variables para las series de Fourier	293
24 Arrays	297
24.1 Funciones y variables para Arrays	297
25 Matrices y Álgebra Lineal	305
25.1 Introducción a las matrices y el álgebra lineal	305
25.1.1 Operador punto	305
25.1.2 Vectores	305
25.1.3 Paquete eigen	305
25.2 Funciones y variables para las matrices y el álgebra lineal	306
26 Afines	329
26.1 Funciones y variables para Afines	329

27	itensor	333
27.1	Introducción a itensor	333
27.1.1	Notación tensorial	334
27.1.2	Manipulación indexada de tensores	334
27.2	Funciones y variables para itensor	337
27.2.1	Trabajando con objetos indexados	337
27.2.2	Simetrías de tensores	346
27.2.3	Cálculo tensorial indexado	347
27.2.4	Tensores en espacios curvos	352
27.2.5	Sistemas de referencia móviles	355
27.2.6	Torsión y no metricidad	358
27.2.7	Álgebra exterior	360
27.2.8	Exportando expresiones en TeX	364
27.2.9	Interactuando con ctensor	364
27.2.10	Palabras reservadas	365
28	ctensor	367
28.1	Introducción a ctensor	367
28.2	Funciones y variables para ctensor	369
28.2.1	Inicialización y preparación	369
28.2.2	Los tensores del espacio curvo	371
28.2.3	Desarrollo de Taylor	374
28.2.4	Campos del sistema de referencia	377
28.2.5	Clasificación algebraica	377
28.2.6	Torsión y no metricidad	379
28.2.7	Otras funcionalidades	380
28.2.8	Utilidades	383
28.2.9	Variables utilizadas por ctensor	387
28.2.10	Nombres reservados	391
29	atensor	393
29.1	Introducción a atensor	393
29.2	Funciones y variables para atensor	394
30	Series	397
30.1	Introducción a las series	397
30.2	Funciones y variables para las series	397
30.3	Series de Poisson	408
31	Teoría de Números	411
31.1	Funciones y variables para teoría de números	411
32	Simetrías	419
32.1	Funciones y variables para simetrías	419

33	Grupos	435
33.1	Funciones y variables para grupos	435
34	Entorno de Ejecución	437
34.1	Introducción al entorno de ejecución	437
34.2	Interrupciones	437
34.3	Funciones y variables para el entorno de ejecución	437
35	Misclánea de opciones	441
35.1	Introducción a la misclánea de opciones	441
35.2	Share	441
35.3	Funciones y variables para la misclánea de opciones	441
36	Reglas y patrones	449
36.1	Introducción a reglas y patrones	449
36.2	Funciones y variables sobre reglas y patrones	449
37	Listas	467
37.1	Introducción a las listas	467
37.2	Funciones y variables para listas	467
38	Conjuntos	475
38.1	Introducción a los conjuntos	475
38.1.1	Utilización	475
38.1.2	Iteraciones con elementos	477
38.1.3	Fallos	478
38.1.4	Autores	479
38.2	Funciones y variables para los conjuntos	479
39	Definición de Funciones	503
39.1	Introducción a la definición de funciones	503
39.2	Funciones	503
39.2.1	Funciones ordinarias	503
39.2.2	Funciones array	504
39.3	Macros	505
39.4	Funciones y variables para la definición de funciones	508
40	Programación	531
40.1	Introducción a la programación	531
40.2	Funciones y variables para la programación	531
41	Depurado	543
41.1	Depuración del código fuente	543
41.2	Claves de depuración	544
41.3	Funciones y variables para depurado	546

42	augmented_lagrangian	551
42.1	Funciones y variables para augmented_lagrangian	551
43	bode	553
43.1	Funciones y variables para bode	553
44	contrib_ode	555
44.1	Introducción a contrib_ode	555
44.2	Funciones y variables para contrib_ode	557
44.3	Posibles mejoras a contrib_ode	560
44.4	Pruebas realizadas con contrib_ode	560
44.5	Referencias para contrib_ode	560
45	descriptive	563
45.1	Introducción a descriptive	563
45.2	Funciones y variables para el tratamiento de datos	565
45.3	Funciones y variables de valores descriptivos	568
45.4	Funciones y variables de valores descriptivos multivariantes	575
45.5	Funciones y variables para gráficos estadísticos	579
46	diag	585
46.1	Funciones y variables para diag	585
47	distrib	593
47.1	Introducción a distrib	593
47.2	Funciones y variables para distribuciones continuas	595
47.3	Funciones y variables para distribuciones discretas	620
48	draw	629
48.1	Introducción a draw	629
48.2	Funciones y variables para draw	629
48.3	Funciones y variables para picture	681
48.4	Funciones y variables para worldmap	683
49	dynamics	687
49.1	Introducción a dynamics	687
49.2	Funciones y variables para dynamics	687
50	ezunits	697
50.1	Introducción a ezunits	697
50.2	Introducción a physical_constants	698
50.3	Funciones y variables para ezunits	700

51	f90	713
51.1	Funciones y variables para f90.....	713
52	ggf	715
52.1	Funciones y variables para ggf.....	715
53	graphs	717
53.1	Introducción a graphs	717
53.2	Funciones y variables para graphs	717
53.2.1	Construyendo grafos	717
53.2.2	Propiedades de los grafos.....	722
53.2.3	Modificación de grafos	737
53.2.4	Lectura y escritura de ficheros	740
53.2.5	Visualización	741
54	grobner	745
54.1	Introducción a grobner	745
54.1.1	Notas sobre el paquete grobner	745
54.1.2	Implementaciones de órdenes admisibles de monomios	745
54.2	Funciones y variables para grobner	746
54.2.1	VARIABLES Opcionales	746
54.2.2	Operadores simples	747
54.2.3	Otras funciones	748
54.2.4	Postprocesamiento estándar de bases de Groebner	749
55	impdiff.....	753
55.1	Funciones y variables para impdiff.....	753
56	interpol	755
56.1	Introducción a interpol	755
56.2	Funciones y variables para interpol	755
57	lapack	761
57.1	Introducción a lapack.....	761
57.2	Funciones y variables para lapack	761
58	lbfgs	765
58.1	Introducción a lbfgs	765
58.2	Funciones y variables para lbfgs	765
59	lindstedt	771
59.1	Funciones y variables para lindstedt	771

60	linearalgebra	773
60.1	Introducción a linearalgebra	773
60.2	Funciones y variables para linearalgebra	775
61	lsquares	787
61.1	Funciones y variables para lsquares	787
62	makeOrders	797
62.1	Funciones y variables para makeOrders	797
63	mnewton	799
63.1	Funciones y variables para mnewton	799
64	numericalio	801
64.1	Introducción a numericalio	801
64.1.1	Entrada y salida en formato texto	801
64.1.2	Separadores válidos para lectura	801
64.1.3	Separadores válidos para escritura	802
64.1.4	Entrada y salida de decimales en formato binario	802
64.2	Funciones y variables para entrada y salida en formato texto	802
64.3	Funciones y variables para entrada y salida en formato binario	804
65	opsubst	807
65.1	Funciones y variables para opsubst	807
66	orthopoly	809
66.1	Introducción a polinomios ortogonales	809
66.1.1	Iniciándose con orthopoly	809
66.1.2	Limitaciones	811
66.1.3	Evaluación numérica	813
66.1.4	Gráficos y orthopoly	814
66.1.5	Miscelánea de funciones	815
66.1.6	Algoritmos	816
66.2	Funciones y variables para polinomios ortogonales	816
67	plotdf	821
67.1	Introducción a plotdf	821
67.2	Funciones y variables para plotdf	821
68	romberg	827
68.1	Funciones y variables para romberg	827

69	simplex	831
69.1	Introducción a simplex	831
69.2	Funciones y variables para simplex	831
70	simplification	833
70.1	Introducción a simplification	833
70.2	Funciones y variables para simplification	833
70.2.1	Paquete absimp	833
70.2.2	Paquete faceexp	833
70.2.3	Paquete functs	835
70.2.4	Paquete ineq	838
70.2.5	Paquete rducon	839
70.2.6	Paquete scifac	840
70.2.7	Paquete sqdnst	840
71	solve_rec	843
71.1	Introducción a solve_rec	843
71.2	Funciones y variables para solve_rec	843
72	stats	849
72.1	Introducción a stats	849
72.2	Funciones y variables para inference_result	849
72.3	Funciones y variables para stats	851
72.4	Funciones y variables para distribuciones especiales	866
73	stirling	867
73.1	Funciones y variables para stirling	867
74	stringproc	869
74.1	Introducción al procesamiento de cadenas	869
74.2	Funciones y variables para entrada y salida	870
74.3	Funciones y variables para caracteres	873
74.4	Funciones y variables para cadenas	875
75	unit	881
75.1	Introducción a units	881
75.2	Funciones y variables para units	882
76	zeilberger	891
76.1	Introducción a zeilberger	891
76.1.0.1	El problema de la suma indefinida	891
76.1.0.2	El problema de la suma definida	891
76.1.1	Niveles de información	891
76.2	Funciones y variables para zeilberger	892

77 Índice de Funciones y Variables	895
Apéndice A Índice de Funciones y Variables	897

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 buscar en las páginas info. La instrucción `describe` mostrará el comando o comandos que contengan una cierta cadena. 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);                                true
(%o1)
(%i2) integerp (1);                                true
(%o2)
(%i3) integerp (-17);                             true
(%o3)
(%i4) integerp (0.0);                            false
(%o4)
(%i5) integerp (1.0);                            false
(%o5)
(%i6) integerp (%pi);                           false
(%o6)
(%i7) integerp (n);                            false
(%o7)
(%i8) declare (n, integer);                      done
(%o8)
(%i9) integerp (n);                            false
(%o9)

-- 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 un resultado de forma posterior, usted puede asignar dicho resultado a una variable o referirse a él por medio de la etiqueta asociada (%i* o %o*). Adicionalmente puede usar % 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);
```

```
(%o3)          6 (y + x)
```

Maxima manipula sin ningún problema números complejos y constantes numéricas:

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

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

Bajo un sistema X window, (es decir que poseea interfaz gráfica), Maxima puede generar gráficas de una o más funciones:

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

2 2

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

2 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`.

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 (llámeselo 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

Imprime un resumen de los parámetros que se usaron para construir la versión de Maxima que se está usando.

`build_info` retorna una cadena vacía "".

3 Ayuda

3.1 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 símbolo 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 caracteres 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 ordinarios (sin una puntuación especial), deben tener nombres tipo Lisp que comiencen con el signo pesos \$.

Maxima es case-sensitive, distingue entre letras minúsculas y mayúsculas en identificadores, mientras que Lisp no. 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 identificador Lisp esté en mayúscula, minúscula o una combinación de ambas, no afecta en nada. Por ejemplo, los identificadores Lisp `$foo`, `$FOO`, y `$Foo`, todos corresponden al identificador Maxima `foo`.
2. Un identificador Lisp el cual se encuentre todo en mayúscula o todo en minúscula y encerrado entre barras verticales corresponde a un identificador Maxima con el caso contrario. Esto es, de mayúsculas cambia a minúsculas y de minúsculas cambia a mayúsculas. E.g., el identificador Lisp `|$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.
```

3.2 Recolector de basura

La computación simbólica tiende a crear una buena cantidad de basura, y un manejo efectivo de esto puede ser crucial para el término exitoso de algunos programas.

Bajo GCL (GNU Common Lisp), en los sistemas UNIX donde la llamada al sistema `mprotect` esta disponible (incluyendo SUN OS 4.0 y algunas variantes de BSD) un recolector de basura estratificado está disponible. Estos límites de colección para memoria virtual, han sido escritos recientemente. Mire la documentación de GCL bajo ALLOCATE y GBC. En el nivel lisp haga (`(setq si::*notify-gbc* t)`) eso le ayudará a determinar cuales áreas necesitan más espacio.

3.3 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.

3.4 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, %gammagreek, gamma, gammagreek, gammalim, gamma_expand,■
gamma_imag, gamma_incomplete, gamma_incomplete_generalized,
gamma_incomplete_regularized, gamma_radius, Gamma, log_gamma, makegamma,■
gamma_incomplete_generalized_regularized]

```

demo (archivo)

Función

Evaluá 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 evalúa 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)

-
(%i7)      exp1 :  $\frac{z}{y+x} + \frac{x}{2(y+x)}$ 
(%o7)       $\frac{z}{y+x} + \frac{x}{2(y+x)}$ 

-
(%i8)      combine(exp1)
(%o8)       $\frac{z}{y+x} + \frac{x}{2(y+x)}$ 

-
(%i9)      rncombine(%)
(%o9)       $\frac{2z+x}{2(y+x)}$ 

-
(%i10)     exp2 :  $\frac{d^3 + c^3 + b^2 + a^2}{3^3 3^2 2^2}$ 
(%o10)     $\frac{d^3 + c^3 + b^2 + a^2}{3^3 3^2 2^2}$ 

```

```

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

```

```

(%i12)          rncombine(exp2)
2 d + 2 c + 3 b + 3 a
-----
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)

```

```

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 (<i>topic</i>)	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 contiene 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)
(%o4)                                lambda([x, y, z], z^2 + y^2 + x^2)
(%i5) %(1,2,a)
(%o5)                                a^2 + 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

,

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)
(%i2) aa^2;
(%o2)
(%i3) 'aa^2;
(%o3)
(%i4) '';
(%o4)
```

1024

1048576

²

aa

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)
(%i2) x1: 7;
(%o2)
(%i3) integrate (x^2, x, x0, x1);
(%o3)
```

5

7

218

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 hayan en la expresión.

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

1024

19

51

bb + sqrt(aa)

51

La comilla simple no evita la simplificación.

```
(%i1) sin (17 * %pi) + cos (17 * %pi);
(%o1)
(%i2) ?(sin (17 * %pi) + cos (17 * %pi));
(%o2)
```

- 1

- 1

"

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)  ''%;           - 273
(%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, el operador comilla-comilla hace que el operador pase de ser un nombre a ser un verbo, a menos que ya sea un verbo.

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

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

(%t8)          foo_2b(x) := %
```

```
(%o8) [%t7, %t8]
(%i8) F : lambda ([u], diff (sin (u), u));
(%o8)           lambda([u], diff(sin(u), u))
(%i9) G : lambda ([u], ''(diff (sin (u), u)));
(%o9)           lambda([u], cos(u))
(%i10) '(sum (a[k], k, 1, 3) + sum (b[k], k, 1, 3));
(%o10)      sum(b , k, 1, 3) + sum(a , k, 1, 3)
                           k
                           3   3   2   2   1   1
(%i11) ''(sum (a[k], k, 1, 3)) + ''(sum (b[k], k, 1, 3));
(%o11)      b + a + b + a + b + a
                           3   3   2   2   1   1
```

4.2 Funciones y variables para la Línea de Comandos

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.

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.

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.
- **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 evalúan 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*. Note 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`, `float` y `pred` 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);
(%o1)          cos(y) + sin(x) + -- (sin(w)) + (w + 1)
                           dw
(%i2) ev (% , sin, expand, diff, x=2, y=1);
(%o2)          cos(w) + w^2 + 2 w + cos(1) + 1.909297426825682
```

Una sintaxis alternativa de alto nivel ha sido proveida para `ev`, por medio de la cual uno puede escribir solamente sus argumentos, sin el comando `ev()`. Esto es, una forma sencilla de escritura:

`expr, arg_1, ..., arg_n`

Esto no es permitido como parte de otra expresión , e.g., en funciones, sentencias block, 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
```

```
(%t7)  $y = \frac{1}{5}$   

(%t8)  $x = -\frac{6}{5}$   

(%o8) [%t7, %t8]  

(%i8) %o6, %o8;  

(%o8)  $-4 = -4$   

(%i9) x + 1/x > gamma (1/2);  

(%o9)  $x + \frac{1}{x} > \sqrt{\pi}$   

(%i10) %, numer, x=1/2;  

(%o10) 2.5 > 1.772453850905516  

(%i11) %, pred;  

(%o11) true
```

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, dotsrules, %emode, %enumer, exponentialize, exptisolate, factorflag, float, halfangles, infeval, isolate_wrt_times, keepfloat, letrat, listarith, logabs, logarc, logexpand, lognegint, lognumer, m1pbranch, numer_pbranch, programmode, radexpand, ratalgdenom, ratfac, ratmx, ratsimpexpsons, simp, simpsum, sumexpand, y trigexpand.

Ejemplos:

```
(%i1) sin (1/2);  

(%o1)  $\sin\left(\frac{1}{2}\right)$   

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

```

evfun

Propiedad

Cuando la función F goza de la propiedad `evfun`, las expresiones `ev(expr, F)` y `expr, F` (en modo interactivo) equivalen a $F(\text{ev}(\text{expr}))$.

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:

`bfloor, factor, fullratsimp, logcontract, polarform, radcan, ratexpand, ratsimp, rectform, rootscontract, trigexpand, y trigreduce.`

Ejemplos:

```

        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 interrumpa la acción. 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.

<code>kill (a_1, ..., a_n)</code>	Función
<code>kill (labels)</code>	Función
<code>kill (inlabels, outlabels, linelabels)</code>	Función
<code>kill (n)</code>	Función
<code>kill ([m, n])</code>	Función
<code>kill (values, functions, arrays, ...)</code>	Función
<code>kill (all)</code>	Función
<code>kill (allbut (a_1, ..., a_n))</code>	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 `lan`.

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 evalúa 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 carácter 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 carácter 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**.

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 verdaderamente una variable opción (o interruptor).

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 evalúa 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 carácter 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)`.

<code>printprops (a, i)</code>	Función
<code>printprops ([a_1, ..., a_n], i)</code>	Función
<code>printprops (all, i)</code>	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`.

<code>prompt</code>	Variable opcional
Valor por defecto: <code>_</code>	
<code>prompt</code> es el símbolo del prompt de la función <code>demo</code> , del modo <code>playback (slow)</code> y del bucle de interrupción de Maxima (el que se invoca con <code>break</code>).	

<code>quit ()</code>	Función
Termina una sesión de Maxima. Nótese que la función debe ser invocada como <code>quit();</code> o <code>quit()\$</code> , no como <code>quit</code> .	
Para parar un cálculo muy demorado pulse <code>Control-C</code> . La acción por defecto es retornar a <code>prompt</code> de Maxima. Si <code>*debugger-hook*</code> tiene como valor <code>nil</code> , pulsar <code>Control-C</code> abrirá el depurador de Lisp. Vea también: <code>debugging</code> .	

<code>remfunction (f_1, ..., f_n)</code>	Función
<code>remfunction (all)</code>	Función
Desliga las definiciones de función de sus símbolos <i>f_1, ..., f_n</i> . Los argumentos pueden ser nombres de funciones ordinarias (creadas con <code>:=</code> o <code>define</code>) o de funciones macro (creadas con <code>::=</code>).	

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`.

<code>reset ()</code>	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**.

sstatus (feature, package)

Función

Configura el estado de *feature* en *package*. Después de que **sstatus (feature, package)** es ejecutado **status (feature, package)** retorna **true**. Esto puede ser muy útil para escritores de paquetes, con el objetivo de conservar las pistas de que características han cargado estos.

to_lisp ()

Function

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

values

Variable del sistema

Valor inicial: **[]**

values es una lista de todas las variables que el usuario ha creado (no constituye las opciones Maxima ni los interruptores). La lista comprende los símbolos **:**, **::**, o **:=**.

5 Operadores

5.1 n-arios

Un operador de tipo **nary**(n-ario, en Maxima **nary**) es usado para denotar una función cuyo número de argumentos es arbitrario; los argumentos se separan por el símbolo del operador, como en $A+B$ o $A+B+C$. La función **nary("x")** es una extensión de la función **Syntax** para declarar a **x** como un operador n-ario. Las funciones pueden ser declaradas para ser **nary**. Si se hace **declare(J,nary);**, se está diciendo al simplificador que haga la reducción de $j(j(a,b),j(c,d))$ a $j(a, b, c, d)$.

Véase también **Syntax**.

5.2 no-fijos

Los operadores de tipo **nofix** (no-fijos, en Maxima **nofix**) son usados para denotar funciones que no reciben argumentos. La mera presencia de tal operador en un comando causará que la correspondiente función sea evaluada. Por ejemplo, cuando se escribe "exit;" para salir de una interrupción de Maxima, "exit" se está comportando de forma similar a un operador no-fijo (**nofix**). La función **nofix("x")** es una extensión de la función **Syntax**, la cual declara a **x** como un operador **nofix**.

Véase también **Syntax**.

5.3 postfijos

Un operador de tipo **postfix** (postfijo), al contrario de los de tipo **prefix**, denota funciones de un solo argumento, pero en este caso el argumento precede inmediatamente la ocurrencia del operador en la cadena de entrada, como en $3!$. La función **postfix("x")** es una extensión de la función **syntax** para declarar a **x** como un operador **postfix**.

Véase también **Syntax**.

5.4 prefijos

Un operador de tipo **prefix** (prefijo, en Maxima **prefix**) indica una función de un (1) argumento, dicho argumento viene inmediatamente después de una ocurrencia del operador. La función **prefix("x")** es una extensión de la función **Syntax** para declarar a **x** como un operador de **prefix**.

Véase también **Syntax**.

5.5 Operadores aritméticos

<code>+</code>	Operador
<code>-</code>	Operador
<code>*</code>	Operador
<code>/</code>	Operador
<code>^</code>	Operador

Los símbolos `+` `*` `/` y `^` representan la suma, resta, multiplicación, división y exponentiación, 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 exponentiación 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 exponentiación, todas las operaciones aritméticas con números dan lugar a resultados en forma de números. La exponentiación 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 exponentiación 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 será 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]);
(%o6) 144 a^2 x^3
```

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

```
(%i1) [a / b, a ^ b];
(%o1) [-, a^b]
(%i2) [map (op, %), map (args, %)];
(%o2) [[/, ^], [[a, b], [a, b]]]
(%i3) [apply ("/", [a, b]), apply ("^", [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)];
(%o2) [* , a, -]
(%o2) 1
(%o2) 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);
(%o1) b + sqrt(11) + -
(%o1) 5
(%o1) 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);
(%o2)          29 11      3
           17 + ----- - 5
                           7
(%i3) simp : true;
(%o3)          true
(%i4) '(17 + 29*11/7 - 5^3);
(%o4)          437
           - ---
                           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]);
(%o1)          [ a - 1   x - 2 ]
           [                   ]
           [ h - 3   u - 4 ]
(%i2) 5 * matrix ([a, x], [h, u]);
(%o2)          [ 5 a   5 x ]
           [                   ]
           [ 5 h   5 u ]
(%i3) listarith : false;
(%o3)          false
(%i4) [a, c, m, t] / [1, 7, 2, 9];
(%o4)          [a, c, m, t]
           -----
           [1, 7, 2, 9]
(%i5) [a, c, m, t] ^ x;
(%o5)          [a, c, m, t]^x
(%i6) listarith : true;
(%o6)          true
(%i7) [a, c, m, t] / [1, 7, 2, 9];
(%o7)          [c   m   t]
           [a, -, -, -]
           7   2   9
(%i8) [a, c, m, t] ^ x;
(%o8)          [x   x   x   x]
           [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);                                true
(%o1)

(%i2) x**y + x^z;                                     z      y
                                                       x + x
(%o2)

(%i3) string (x**y + x^z);                           x^z+x^y
(%o3)

(%i4) fortran (x**y + x^z);                         x**z+x**y
(%o4) done

```

5.6 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 relationales 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.

```
(%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]
```

5.7 Operadores generales

^

Operator

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

La exponenciación no comutativa 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
```

!

Operator

El operador factorial. Para cualquier número complejo `x` (incluyendo enteros, racionales y números reales) excepto para enteros negativos, `x!` se define como `gamma(x+1)`.

Para un entero `x`, `x!` simplifica el producto de los enteros desde 1 hasta `x`. `0!` simplifica a 1. Para a un número de punto flotante `x`, `x!` calcula al valor de `gamma(x+1)`. Para `x` igual a `n/2` donde `n` es un entero impar, `x!` simplifica a un factor racional por `sqrt(%pi)` (donde `gamma(1/2)` es igual a `sqrt(%pi)`). Si `x` es cualquier otra cosa, `x!` no se simplifica.

Las variables `factlim`, `minfactorial` y `factcomb` controlan la simplificación de expresiones que contienen factoriales.

Las funciones `gamma`, `bffac` y `cbffac` son variaciones de la función `gamma`. `makegamma` substituye a `gamma` para factoriales y funciones relacionadas.

Funciones relacionadas: `binomial`

- El factorial de un entero, semi-entero o de punto flotante es simplificado a menos que el operando sea mayor que `factlim`.

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

- El factorial de un número complejo, constante conocida (por ejemplo %e) o una expresión general no es simplificado.

Sin embargo puede ser posible simplificar el factorial después de evaluar el operando.

```
(%i1) [(%i + 1)!, %pi!, %e!, (cos(1) + sin(1))!];
(%o1)      [(%i + 1)!, %pi!, %e!, (sin(1) + cos(1))!]
(%i2) ev (%o1, numer, %enumer);
(%o2)  [(%i + 1)!, 7.188082728976037, 4.260820476357,
          1.227580202486819]
```

- El factorial de un símbolo no se simplifica.

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

- Los factoriales son simplificados, no evaluados. Así x! puede ser reemplazado en una expresión precedida por el operador comilla.

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

!!

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 definiciones 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).

#

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

El operador punto, para multiplicación de matrices (no-commutativo). 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`.

: 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 $f(x_1, \dots, x_n) := \text{expr}$ define una función de nombre f con argumentos x_1, \dots, 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 argumen-

tos se asignan uno a uno a los argumentos formales $x_1, \dots, x_{(n - 1)}$, 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
```

=

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
```

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 commutativo: `a and b` puede no ser igual a `b and a` debido al tratamiento de operandos indeterminados.

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 commutativo: **a or b** puede no ser igual a **b or 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.

abs (expr)

Función

Devuelve el valor absoluto de **expr**. Si la expresión es compleja, retorna el módulo de **expr**.

additive

Clave

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)**.

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, ...`.

antisymmetric

Declaración

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.

cabs (expr)

Función

Devuelve el valor absoluto complejo (módulo complejo) de `expr`.

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
```

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]
```

commutative

Declaración

Si `declare(h,commutative)` es ejecutado, le dice al simplificador que *h* es una función conmutativa. Por ejemplo, *h(x,z,y)* se simplificará a *h(x,y,z)*. Esto es lo mismo que `symmetric`.

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* \neq *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) compareacos(x^2+1), acos(x^2+1) + 1);
(%o1) <
```

Aquí, el dominio real de `acos (x^2 + 1)` es el conjunto vacío.

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.

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);           x - y
(%o7)                               x - y
(%i8) is (equal (x, y));         unknown
(%o8)                               unknown
(%i9) is (x = y);                false
(%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
      [x  + 2 x + 1, x  - 2 x - 1]
(%o1) 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
```

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
```

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
```

eval

Operador

El operador **eval** realiza una evaluación extra de una expresión *expr*. Véase **ev**.

evenp (*expr*)

Función

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

fix (*x*)

Función

Es un sinónimo de **entier** (*x*).

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]]
```

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
```

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
```

isqrt (x)	Función
Devuelve la "raíz cuadrada entera" del valor absoluto de x, el cual debe ser un entero.	
lmax (L)	Función
Si L es una lista o conjunto, devuelve <code>apply ('max, args (L))</code> . Si L no es una lista o conjunto, envía un mensaje de error.	
lmin (L)	Función
Si L es una lista o conjunto, devuelve <code>apply ('min, args (L))</code> . 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 <code>get (trylevel, maxmin)</code> es 2 o más, <code>max</code> aplica la simplificación <code>max (e, -e) --> e </code> . Si <code>get (trylevel, maxmin)</code> es 3 o más, <code>max</code> intenta eliminar las expresiones que estén entre otros dos de los argumentos dados; por ejemplo, <code>max (x, 2*x, 3*x) --> max (x, 3*x)</code> . Para asignar el valor 2 a <code>trylevel</code> se puede hacer <code>put (trylevel, 2, maxmin)</code> .	
min (x_1, ..., x_n)	Función
Devuelve un valor simplificado de la menor de las expresiones desde x_1 hasta x_n. Si <code>get (trylevel, maxmin)</code> es 2 o más, <code>min</code> aplica la simplificación <code>min (e, -e) --> e </code> . Si <code>get (trylevel, maxmin)</code> es 3 o más, <code>min</code> intenta eliminar las expresiones que estén entre otros dos de los argumentos dados; por ejemplo, <code>min (x, 2*x, 3*x) --> min (x, 3*x)</code> . Para asignar el valor 2 a <code>trylevel</code> se puede hacer <code>put (trylevel, 2, maxmin)</code> .	
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 <code>modulus</code> .	
La llamada <code>polymod (p, m)</code> especifica un módulo m para ser utilizado en lugar de valor almacenado en <code>modulus</code> .	
Véase <code>modulus</code> .	
mod (x, y)	Función
Si x e y son números reales e y es distinto de cero, devuelve <code>x - y * floor(x / y)</code> . Para todos los reales x, se tiene <code>mod (x, 0) = x</code> . Para información sobre la definición de <code>mod (x, 0) = x</code> , véase la sección 3.4 de "Concrete Mathematics", by Graham, Knuth, and Patashnik. La función <code>mod (x, 1)</code> es de diente de sierra con periodo unidad y con <code>mod (1, 1) = 0</code> y <code>mod (0, 1) = 0</code> .	
Para encontrar el argumento principal (un número del intervalo <code>(-%pi, %pi]</code>) de un número complejo, hágase uso de la función <code>x -> %pi - mod (%pi - x, 2*%pi)</code> , donde x es un argumento.	
Si x e y son expresiones constantes (por ejemplo, <code>10 * %pi</code>), <code>mod</code> 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);                                x
(%o1)
(%i2) mod (a*x, a*y);                           a mod(x, y)
(%o2)
(%i3) mod (0, x);                                0
(%o3)
```

oddp (*expr*)

Función

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

pred

Operador

El operador **pred** realiza una evaluación extra de un predicado (expresión cuya evaluación debe dar **true** o **false**). Véase **ev**.

make_random_state (*n*)
make_random_state (*s*)
make_random_state (**true**)
make_random_state (**false**)

Función
Función
Función
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)          11804491615036831636390
(%i6) s2: make_random_state (false)$
(%i7) random (1.0);
(%o7)          .2310127244107132
(%i8) random (10.0);
(%o8)          4.394553645870825
(%i9) random (100.0);
(%o9)          32.28666704056853
(%i10) set_random_state (s2);
(%o10)                                done
(%i11) random (1.0);
(%o11)          .2310127244107132
(%i12) random (10.0);
(%o12)          4.394553645870825
(%i13) random (100.0);
(%o13)          32.28666704056853
```

rationalize (expr)

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);
(%o1)
      1
      -
      2
(%i2) rationalize (0.1);
(%o2)
      1
      --
      10
(%i3) fpprec : 5$ 
(%i4) rationalize (0.1b0);
(%o4)
      209715
      -----
      2097152
```

```
(%i5) fpprec : 20$  

(%i6) rationalize (0.1b0);  

                               236118324143482260685  

(%o6)  

-----  

                               2361183241434822606848  

(%i7) rationalize (sin (0.1*x + 5.6));  

                               x      28  

(%o7)           sin(-- + --)  

                  10      5
```

Ejemplo de uso:

```
(%i1) unitfrac(r) := block([uf : [], q],  

   if not(ratnump(r)) then  

     error("The input to 'unitfrac' must be a rational number"),  

   while r # 0 do (  

     uf : cons(q : 1/ceiling(1/r), uf),  

     r : r - q),  

   reverse(uf));  

(%o1) unitfrac(r) := block([uf : [], q],  

   if not ratnump(r) then  

     error("The input to 'unitfrac' must be a rational number"),  

     1  

   while r # 0 do (uf : cons(q : -----, uf), r : r - q),  

           1  

           ceiling(-)  

           r  

   reverse(uf))  

                               1   1   1  

(%o2) [-, -, --]  

          2   3   15  

(%i3) unitfrac (9/10);  

                               9  

(%o3)  

-----  

          10  

(%i4) apply ("+", %);  

                               1  

(%o4) [- 1, --]  

          10  

(%i5) unitfrac (-9/10);  

                               9  

(%o5)  

- --  

          10  

(%i6) apply ("+", %);  

                               1   1   1   1   1  

(%o6) [-, -, -, --, -----]  

          2   3   8   69   6808  

(%i7) unitfrac (36/37);  

                               36  

(%o7)  

-----  

          37
```

```
(%i8) apply ("+", %);
```

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**.

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).

signum (x)

Función

Para x numérico, devuelve 0 si x es 0, en caso contrario devuelve -1 o +1, según que x sea menor o mayor que 0, respectivamente.

Si x no es numérico, entonces se devuelve una forma simplificada equivalente. Por ejemplo, **signum(-x)** devuelve **-signum(x)**.

sort (L, P)

Función

sort (L)

Función

Ordena la lista L de acuerdo con el predicado P de dos argumentos, de tal manera que $P(L[k], L[k+1])$ es **true** (verdadero) para cualesquiera dos elementos sucesivos. El predicado se puede especificar como nombre de una función o de un operador infixo binario, o como una expresión **lambda**. Si se especifica con el nombre de un operador, este nombre debe encerrarse con "comillas dobles".

La lista ordenada se devuelve como un objeto nuevo, de manera que el argumento L no se ve alterado. A fin de construir el valor de retorno, **sort** hace una copia previa de los elementos de L .

Si el predicado P no ordena totalmente los elementos de L , entonces **sort** puede seguir ejecutándose hasta el final sin emitir errores, pero el resultado no es predecible. La función muestra un mensaje de error en caso de que el predicado devuelva algo diferente de **true** o **false**.

La llamada **sort (L)** equivale a **sort (L, orderlessp)**; esto es, el orden por defecto es el ascendente, tal como queda definido por **orderlessp**. Todos los átomos y expresiones de Maxima son comparables para **orderlessp**, aunque existen ejemplos aislados de expresiones para las cuales **orderlessp** deja de ser transitivo; se trata de un fallo de Maxima.

Ejemplos:

```
(%i1) sort ([11, -17, 29b0, 7.55, 3, -5/2, b + a, 9 * c, 19 - 3 * x]);■
      5
(%o1) [- 17, - -, 3, 7.55, 11, 2.9b1, b + a, 9 c, 19 - 3 x]
      2
(%i2) sort ([11, -17, 29b0, 7.55, 3, -5/2, b + a, 9 * c, 19 - 3 * x],■
           ordergreatp);
```

```

      5
(%o2) [19 - 3 x, 9 c, b + a, 2.9b1, 11, 7.55, 3, - -, - 17]
      2

(%i3) sort ([%pi, 3, 4, %e, %gamma]);
(%o3) [3, 4, %e, %gamma, %pi]
(%i4) sort ([%pi, 3, 4, %e, %gamma], "<");
(%o4) [%gamma, %e, 3, %pi, 4]
(%i5) my_list : [[aa, hh, uu], [ee, cc], [zz, xx, mm, cc], [%pi, %e]];■
(%o5) [[aa, hh, uu], [ee, cc], [zz, xx, mm, cc], [%pi, %e]]
(%i6) sort (my_list);
(%o6) [[%pi, %e], [aa, hh, uu], [ee, cc], [zz, xx, mm, cc]]
(%i7) sort (my_list, lambda ([a, b],
                               orderlessp (reverse (a), reverse (b))));■
(%o7) [[%pi, %e], [ee, cc], [zz, xx, mm, cc], [aa, hh, uu]]

```

sqrt (x)

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**.

sqrtdispflag

Variable opcional

Valor por defecto: **true**

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

sublis (list, expr)

Función

Realiza sustituciones múltiples en paralelo en una expresión.

La variable **sublis_apply_lambda** controla la simplificación después de **sublis**.

Ejemplo:

```

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

```

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]

```

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.

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 derecho será sustituida por la expresión del miembro izquierdo en *expr*.

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

Para más ejemplos, ejecútese **example (subst)**.

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

Función

Es similar a **substpart**, pero **substinpart** trabaja con la representación interna de *expr*.

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

Si el último argumento de la función **part** es una lista de índices, entonces se toman varias subexpresiones, cada una de las cuales en correspondencia con un índice de la lista. Así,

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

`piece` guarda el valor de la última expresión seleccionada cada vez que se utiliza la función `part`. Esta asignación se hace durante la ejecución de la función, con lo que puede ser referenciada en la propia función tal como se muestra más abajo. Si `partswitch` vale `true` entonces se devuelve `end` cuando la parte seleccionada de una expresión no existe, en caso contrario se muestra 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]);
(%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
      - - + - + -
            z   x   x
(%i6) substpart (xthru (piece), expr, [2, 3]);
            y + 1   1
      ----- - -
            x       z
(%o6)
```

Además, darle a la opción `inflag` el valor `true` y llamar a `part` o a `substpart` es lo mismo que llamar a `inpart` o a `substingroup`.

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

Función

Sustituye *x* por la subexpresión determinada por el resto de argumentos, según el esquema de `part`. Devuelve el nuevo valor de *expr*. El argumento *x* puede ser un operador a ser sustituido por un operador de *expr*. En algunos casos *x* necesita estar acotado por comillas dobles ", como en `substpart ("+", a*b, 0)` para que retorne *b + a*.

```
(%i1) 1/(x^2 + 2);
(%o1)          1
              -----
                  2
                  x  + 2
(%i2) substpart (3/2, %, 2, 1, 2);
(%o2)          1
              -----
                  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, darle a la opción `inflag` el valor `true` y llamar a `part` o a `substpart` es lo mismo que llamar a `inpart` o a `substinpart`.

subvarp (expr)

Función

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

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`. Véase también `ordergreat` y `orderless`.

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

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.

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 (%);
(%o1)      1         (x + 2)      - 2 y      x
----- + ----- - -----
(y + x)      19          20            20
(y + x)      (y + x)
```

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*.

6 Expresiones

6.1 Introducción a las expresiones

Existe un cierto número de palabras reservadas que no pueden utilizarse como nombres de variables. Su uso podría causar errores críticos de sintaxis.

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
```

6.2 Expresiones complejas

Una expresión compleja se especifica en Maxima añadiendo a la parte real de la expresión el producto de `%i` por la parte imaginaria. Así las raíces de la ecuación $x^2 - 4*x + 13 = 0$ son $2 + 3*i$ y $2 - 3*i$. Nótese que la simplificación de productos de expresiones complejas puede ser efectuado expandiendo el producto. La simplificación de cocientes, raíces y otras funciones que contengan expresiones complejas pueden normalmente conseguirse a través de las funciones `realpart`, `imagpart`, `rectform`, `polarform`, `abs` o `carg`.

6.3 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;
(%o1)
(%i2) foo (42);
(%o2)
(%i3) 'foo (42);
(%o3)
(%i4) 'foo (42), nouns;
(%o4)
(%i5) declare (bar, noun);
(%o5)
(%i6) bar (x) := x/17;
(%o6)
(%i7) bar (52);
(%o7)
(%i8) bar (52), nouns;
(%o8)
(%i9) integrate (1/x, x, 1, 42);
(%o9)
(%i10) 'integrate (1/x, x, 1, 42);
(%o10)
```

```

1
(%i11) ev (% , nouns);
(%o11)          log(42)

```

6.4 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`, `F00` y `Foo` son distintos. Véase [\(undefined\)](#) [Lisp y Maxima], página [\(undefined\)](#) 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 [\(undefined\)](#) [Lisp y Maxima], página [\(undefined\)](#) 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 = F00), is (F00 = Foo), is (Foo = foo)];
(%o11)          [false, false, false]
(%i12) :lisp (defvar *my-lisp-variable* '$foo)
*MY-LISP-VARIABLE*
(%i12) ?\*my\~-lisp\~-variable\*;
(%o12)          foo

```

6.5 Cadenas de caracteres

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."
```

6.6 Desigualdades

Maxima dispone de los operadores de desigualdad <, <=, >=, >, # y `notequal`. Véase `if` para una descripción de las expresiones condicionales.

6.7 Sintaxis

Es posible definir nuevos operadores con una precedencia especificada, o eliminar o redefinir la precedencia de operadores ya existentes. Un operador puede ser de tipo prefijo unario o postfijo unario, infijo binario, infijo n-ario, "bi-fijo" (matchfix) 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. A continuación ejemplos sobre los diferentes tipos de operadores.

prefijo unario

negación - a

postfijo unario

factorial a!

infijo binario

exponenciación a^b

n-ary infix suma a + b

"bi-fijo" construcción de una lista [a, b]

(Maxima no incluye operadores "no-fijos", pero se puede ver un ejemplo en `nofix`.)

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
```

Las funciones de Maxima que definen nuevos operadores se resumen en esta tabla, en la que se establecen las fuerzas de enlace a izquierda (lbp, de *left binding power*) y a derecha (rbp, de *right binding power*) por defecto. (La fuerza de enlace determina la precedencia del operador. Sin embargo, puesto que las fuerzas de enlace a izquierda y derecha pueden ser diferentes, la fuerza de enlace es algo más que la simple precedencia.) Algunas de las funciones para definir operadores toman argumentos adicionales; véanse las descripciones de estas funciones para más detalles.

`prefix rbp=180`

`postfix lbp=180`

```

infix      lbp=180, rbp=180
nary       lbp=180, rbp=180
matchfix   (la fuerza de enlace no se aplica aquí)
nofix      (la fuerza de enlace no se aplica aquí)

```

A efectos comparativos, aquí se presentan algunos operadores de Maxima junto con sus fuerzas de enlace a izquierda y derecha.

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

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)
```

6.8 Funciones y variables para expresiones

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

Función

Evaluá 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, consultense **subst** y **ev**.

Ejemplos:

```
(%o4) 8 f(x, y) (--) (f(x, y)) - 2 u(x, y) (--) (u(x, y))
          dx                               dx
(%i5) at (% , [x = 0, y = 1]);
!
(%o5)   16 a  - 2 u(0, 1) (--) (u(x, y))!
          dx
!
!x = 0, y = 1
```

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 excluida 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.

carg (z)

Función

Devuelve el argumento complejo de *z*. El argumento complejo es un ángulo **theta** en $(-\pi, \pi]$ tal que $r \exp(\theta) = z$ donde *r* es la magnitud de *z*.

La función **carg** es computacional, no simplificativa.

La función **carg** ignora la declaración **declare (x, complex)**, y trata a *x* como una variable real. Se trata de un fallo conocido en Maxima.

Véanse también **abs** (módulo complejo), **polarform**, **rectform**, **realpart** y **imagpart**.

Ejemplos:

```
(%i1) carg (1);
(%o1)                               0
(%i2) carg (1 + %i);
(%o2)           %pi
-----  
                    4
(%i3) carg (exp (%i));
(%o3)                               1
(%i4) carg (exp (%pi * %i));
(%o4)           %pi
(%i5) carg (exp (3/2 * %pi * %i));
(%o5)           %pi
-----  
                    2
(%i6) carg (17 * exp (2 * %i));
(%o6)                               2
```

constant

Operator especial

La llamada **declare (a, constant)** declara *a* como constante. Véase **declare**.

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`. No obstante, `featurep` no reconoce algunas propiedades, lo cual es un fallo conocido de Maxima.

Véase también `features`.

La función `declare` reconoce las siguientes propiedades:

`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 `evfun`.

`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`.

`bindtest` Hace que Maxima envíe un error si `a_i` es evaluado sin habersele asignado un valor.

`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.

`constant` Hace que Maxima considere a `a_i` como una constante simbólica.

`scalar` Hace que Maxima considere a `a_i` como una variable escalar.

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.

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**.

alphabetic

Indica a Maxima que reconozca todos los caracteres de la cadena alfanumérica *a_i* como caracteres alfábéticos.

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*.

rassociative, lassociative

Hace que Maxima reconozca a *a_i* como una función asociativa por la derecha o por la izquierda.

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)**.

symmetric, antisymmetric, commutative

Hace que Maxima reconozca a *a_i* como una función simétrica o anti-simétrica. La propiedad **commutative** equivale a **symmetric**.

evenfun, oddfun

Hace que Maxima reconozca a *a_i* como una función par o impar.

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.

multiplicative

Hace que Maxima simplifique las expresiones *a_i* haciendo uso de la sustitución $a_i(x * y * z * \dots) \rightarrow a_i(x) * a_i(y) * a_i(z) * \dots$. Tal sustitución se aplica únicamente al primer argumento.

additive Hace que Maxima simplifique las expresiones *a_i* haciendo uso de la sustitución $a_i(x + y + z + \dots) \rightarrow a_i(x) + a_i(y) + a_i(z) + \dots$. Tal sustitución se aplica únicamente al primer argumento.

linear Equivale a declarar *a_i* conjuntamente como **outative** y **additive**.

integer, noninteger

Hace que Maxima reconozca a *a_i* como una variable entera o no entera.

even, odd Hace que Maxima reconozca a *a_i* como una variable entera par o impar.

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.

increasing, decreasing Hace que Maxima reconozca a *a_i* como una función creciente o decreciente.

posfun Hace que Maxima reconozca a *a_i* como una función positiva.

integervalued

Hace que Maxima reconozca a *a_i* como una función de valor entero.

Ejemplos:

Declaraciones en base a **evfun** y **evflag**.

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

Declaración en base a **bindtest**.

```
(%i1) aa + bb;
(%o1)
bb + aa
(%i2) declare (aa, bindtest);
(%o2)
done
(%i3) aa + bb;
aa unbound variable
-- an error. Quitting. To debug this try debugmode(true);
(%i4) aa : 1234;
(%o4)
1234
(%i5) aa + bb;
(%o5)
bb + 1234
```

Declaración en base a **noun**.

```
(%i1) factor (12345678);
          2
(%o1)           2 3 47 14593
(%i2) declare (factor, noun);
(%o2)           done
(%i3) factor (12345678);
(%o3)           factor(12345678)
(%i4) ',%, nouns;
          2
(%o4)           2 3 47 14593
```

Declaraciones en base a `constant`, `scalar`, `nonscalar` y `mainvar`.

Declaración en base a `alphabetic`.

```
(%i1) xx\~yy`@\ : 1729;
(%o1)           1729
(%i2) declare ("~`@", alphabetic);
(%o2)           done
(%i3) xx~yy`@ + @yy`xx + `xx@yy~;
(%o3)           `xx@yy~ + @yy`xx + 1729
(%i4) listofvars (%);
(%o4)           [@yy`xx, `xx@yy~]
```

Declaración en base a `feature`.

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

Declaraciones en base a `rassociative` y `lassociative`.

Declaración en base a `nary`.

```
(%i1) H (H (a, b), H (c, H (d, e)));
(%o1)           H(H(a, b), H(c, H(d, e)))
(%i2) declare (H, nary);
(%o2)           done
(%i3) H (H (a, b), H (c, H (d, e)));
(%o3)           H(a, b, c, d, e)
```

Declaraciones en base a `symmetric` y `antisymmetric`.

```
(%i1) S (b, a);
(%o1)           S(b, a)
(%i2) declare (S, symmetric);
(%o2)           done
(%i3) S (b, a);
(%o3)           S(a, b)
(%i4) S (a, c, e, d, b);
(%o4)           S(a, b, c, d, e)
(%i5) T (b, a);
(%o5)           T(b, a)
```

```
(%i6) declare (T, antisymmetric);
(%o6)                                done
(%i7) T (b, a);
(%o7)                                - T(a, b)
(%i8) T (a, c, e, d, b);
(%o8)                                T(a, b, c, d, e)
```

Declaraciones en base a `oddfun` y `evenfun`.

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

Declaración en base a `outative`.

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

Declaración en base a `multiplicative`.

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

Declaración en base a `additive`.

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

Declaración en base a `linear`.

```
(%i1) 'sum (F(k) + G(k), k, 1, inf);
                           inf
```

```

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

```

disolate (*expr, x₁, ..., 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 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
```

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)));
(%o3)                                1
                               -----
                               (b + a) (d + c)
(%i4) expand (1/((a+b) * (c+d)), 1, 0);
(%o4)                                1
                               -----
                               b d + a d + b c + a c
```

dpart (expr, n₁, ..., 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);
(%o1)                                y
                                         -----
                                         2
                                         """
                                         "z"
                                         """
```

exp (x)	Función
Representa la función exponencial. La expresión <code>exp (x)</code> en la entrada se simplifica en e^x ; <code>exp</code> no aparece en expresiones simplificadas.	
Si la variable <code>demoivre</code> vale <code>true</code> hace que $\text{e}^{(a + b \pi i)}$ se simplifique a $\text{e}^a (\cos(b) + \pi i \sin(b))$ si b no contiene a πi . Véase <code>demoivre</code> .	
Si la variable <code>%emode</code> vale <code>true</code> , hace que $\text{e}^{(\pi i x)}$ se simplifique. Véase <code>%emode</code> .	
Si la variable <code>%enumer</code> vale <code>true</code> hace que e se reemplace por 2.718... siempre que <code>numer</code> valga <code>true</code> . Véase <code>%enumer</code> .	
%emode	Variable opcional
Valor por defecto: <code>true</code>	
Si <code>%emode</code> vale <code>true</code> , $\text{e}^{(\pi i x)}$ se simplifica como sigue.	
$\text{e}^{(\pi i x)}$ se simplifica a $\cos(\pi x) + \pi i \sin(\pi x)$ 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 , $\text{e}^{(\pi i x)}$ se simplifica a $\text{e}^{(\pi i y)}$ donde y es $x - 2k$ para algún entero k tal que $\text{abs}(y) < 1$.	
Si <code>%emode</code> vale <code>false</code> , no se realizan simplificaciones especiales a $\text{e}^{(\pi i x)}$.	
%enumer	Variable opcional
Valor por defecto: <code>false</code>	
Si la variable <code>%enumer</code> vale <code>true</code> hace que e se reemplace por 2.718... siempre que <code>numer</code> valga <code>true</code> .	
Si <code>%enumer</code> vale <code>false</code> , esta sustitución se realiza sólo si el exponente en e^x tiene un valor numérico.	
Véanse también <code>ev</code> y <code>numer</code> .	
exptsubst	Variable opcional
Valor por defecto: <code>false</code>	
Si <code>exptsubst</code> vale <code>true</code> permite la sustitución y por e^x en e^a .	
freeof (x_1, ..., x_n, expr)	Función
<code>freeof (x_1, expr)</code> Devuelve <code>true</code> si ninguna subexpresión de <code>expr</code> coincide con x_1 , o si x_1 aparece como variable muda en <code>expr</code> , o si x_1 no es ni una forma nominal ni verbal de cualesquiera operadores presentes en <code>expr</code> , devolviendo <code>false</code> en otro caso.	
La llamada <code>freeof (x_1, ..., x_n, expr)</code> equivale a <code>freeof (x_1, expr) and ... and freeof (x_n, expr)</code> .	
Los argumentos x_1, \dots, x_n pueden ser nombres de funciones y variables, nombres subindicados, operadores (encerrados entre comillas dobles) o expresiones generales. La función <code>freeof</code> 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 <code>freeof</code> son el índice de una suma o producto, la variable límite en <code>limit</code> , 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`. Las variables locales en `block` no son reconocidas por `freeof` como variables mudas; esto es un fallo de Maxima.

La versión indefinida de `integrate` no está libre de su variable de integración.

- Los argumentos son nombres de funciones, variables, nombres subindicados, operadores 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
                               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
      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
```

genfact (x, y, z) Función

Devuelve el factorial generalizado, definido como $x(x-z)(x-2z)\dots(x-(y-1)z)$. Así, para el entero x , $\text{genfact}(x, x, 1) = x!$ y $\text{genfact}(x, x/2, 2) = x!!$.

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`.

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
```

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 `pickupart`.

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 a él. 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  
  
(%t4)          2          2  
              b      + 2 a b + a  
  
(%o4)          2  
              c      + %t3 c + %t2 c + %t4  
(%i4) isolate_wrt_times: false$  
(%i5) isolate (expand ((a+b+c)^2), c);  
  
(%o5)          2  
              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**, **%iy** cualquier otra variable declarada constante de las que aparezcan en *expr*. Estas constantes se omiten por defecto.

```
(%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.

lopow (expr, x) Función

Devuelve el mínimo exponente de *x* que aparece explícitamente en *expr*. Así,

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

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.

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 *f₁ f₂ ... 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;
(%o1) - ----- + ----- - -----
          1           x           f(x)
          x - y         2           (x - y)      3
                               2
                               (x - y)      (x - y)
(%i2) multthru ((x-y)^3, %);
(%o2) - (x - y)  + x (x - y) - f(x)
(%i3) ratexpand (%);
(%o3) - y  + x y - f(x)
(%i4) ((a+b)^10*s^2 + 2*a*b*s + (a*b)^2)/(a*b*s^2);
```

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

nounify (f)

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));            sin
(%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));    "if"
(%o8)                                "if"
(%i9) op ('foo (a));               foo
(%o9)                                foo
(%i10) prefix (foo);              "foo"
(%o10)                               "foo"
(%i11) op (foo a);                "foo"
(%o11)                               "foo"
(%i12) op (F [x, y] (a, b, c));   F
(%o12)                               F
                                         x, y
(%i13) op (G [u, v, w]);          G
(%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*.

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*)
orderlessp (*expr_1*, *expr_2*)

Función
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)
(%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) orderbygreat (S, J);
(%o2)                                done
(%i3) orderbyless (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*.

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.

Ejemplo: **part** (*z+2*y*, 2, 1) devuelve 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 equivalentes 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));
(%o1)

$$-\log(\sqrt{x+1}+1) + \frac{\sin(x^2)}{3} + \frac{b+a}{2}$$

(%i2) pickapart (expr, 0);

(%t2)

$$-\log(\sqrt{x+1}+1) + \frac{\sin(x^2)}{3} + \frac{b+a}{2}$$

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

(%t3)

$$-\log(\sqrt{x+1}+1)$$

(%t4)

$$\frac{\sin(x^2)}{3}$$

```

```

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

              2

(%o5)          %t5 + %t4 + %t3  

(%i5) pickapart (expr, 2);

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

(%t7)          sin(x )  

              2

(%t8)          b + a  

              %t8   %t7  

----- + ----- - %t6  

              2       3
(%i8) pickapart (expr, 3);

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

(%t10)          x  

              2

(%o10)          b + a      sin(%t10)  

----- - log(%t9) + -----  

              2           3
(%i10) pickapart (expr, 4);

(%t11)          sqrt(x + 1)

(%o11)          sin(x )  

              2      b + a  

----- + ----- - log(%t11 + 1)  

              3           2
(%i11) pickapart (expr, 5);

(%t12)          x + 1

(%o12)          sin(x )  

              2      b + a  

----- + ----- - log(sqrt(%t12) + 1)  

              3           2
(%i12) pickapart (expr, 6);
              2

```

$$(\%o12) \quad \frac{\sin(x)}{3} + \frac{b + a}{2} - \log(\sqrt{x + 1}) + 1$$
piece

System variable

Guarda la última expresión seleccionada por las funciones `part`.**polarform (expr)**

Función

Devuelve una expresión de la forma $r \%e^{(\%i \theta)}$ equivalente a `expr`, con `r` y `theta` son reales.**powers (expr, x)**

Función

Devuelve las potencias de `x` dentro de `expr`.La instrucción `load (powers)` carga esta función.**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
          /===\ \
          ! !
          ! !   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
          ! !   -----
          ! !   k + 1
          k = 1
(%i9) product (if k <= 5 then a^k else b^k, k, 1, 10);
          15   40
          a   b
```

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 $a + b \sqrt{-1}$ equivalente a *expr*, con *a* y *b* reales.

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 cajas, 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
          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"
          "sin("%pi x)" + "-----"
          "      """""" "" "      "
          "      """""" "" " 2" "
          "      ""h "" "
          "      """" "
          """

(%i4) rembox (expr2, unlabelled);
          BAR"""""""
          FOO"""""""""" "a d - b c"
          (%o4)      "sin(%pi x)" + "-----"
          "      """""" "" " 2" "
          "      ""h "" "
          """

(%i5) rembox (expr2, FOO);
          BAR"""""""
          "      """""" "" "a d - b c"
          (%o5)      sin("%pi x") + "-----"
          "      """""" "" "      "
          "      """""" "" " 2" "
          "      ""h "" "
          "      """" "
          """

(%i6) rembox (expr2, BAR);
          FOO"""""""""""
          "      """""" "" a d - b c
          (%o6)      "sin("%pi x)" + -----
          "      """""" ""      "
          "      """""" "" " 2"
          "      ""h "" "
          "      """" "
          """

(%i7) rembox (expr2);
          a d - b c
          (%o7)      sin(%pi x) + -----
                      2
                      h

```

sum (expr, i, i_0, i_1)

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 **cauchy** 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
               ====
               \
(%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
```

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

```
3  
i in rootsof(x - 1)
```

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
```


7 Simplificación

7.1 Funciones y variables para simplificación

askexp

Variable del sistema

Cuando se invoca a `asksign`, la expresión que se va a analizar es precisamente `askexp`.

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)`.

Las llamadas `askinteger (expr, even)` y `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 completar 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`.

demoivre (expr)

Función

demoivre

Variable opcional

La función `demoivre (expr)` convierte una expresión sin modificar la variable global `demoivre`.

Cuando `demoivre` vale `true`, los exponentiales 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.

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)`.

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`.

La variable `expand` utilizada con `ev` provocará una expansión.

El fichero ‘`simplification/faceexp.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/faceexp.usg`’ se pueden encontrar breves descripciones de estas funciones. Se accederá a una demostración con la instrucción `demo("faceexp")`.

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 tendrá 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 (<i>expr</i> , <i>x_1</i> , ..., <i>x_n</i>)	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 <i>expr</i> que contienen a las variables <i>x_1</i> , ..., <i>x_n</i> .	
Esta función se carga automáticamente de ‘ <i>simplification/stopex.mac</i> ’.	
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 (<i>expr</i>)	Función
exponentialize	Variable opcional
La función exponentialize (<i>expr</i>) convierte las funciones trigonométricas e hiperbólicas de <i>expr</i> 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 <i>n</i> mayor que expop , entonces expand ($(x+1)^n$) se desarrollará sólo si maxposex no es menor que <i>n</i> .	
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.	
intosum (<i>expr</i>)	Función
Mueve los factores multiplicativos que están fuera de un sumatorio hacia dentro de éste. Si el índice 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 , <i>expr</i>) antes que intosum .	
lassociative	Declaración
La instrucción declare (<i>g</i> , lassociative) le indica al simplificador de Maxima que <i>g</i> es asociativo por la izquierda. Por ejemplo, <i>g</i> (<i>g</i> (<i>a</i> , <i>b</i>), <i>g</i> (<i>c</i> , <i>d</i>)) se reduce a <i>g</i> (<i>g</i> (<i>a</i> , <i>b</i>), <i>c</i>), <i>d</i>).	

linear Declaración

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`.

mainvar Declaración

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 .)

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 máxima altura a la que escalará `applyb1` antes de detenerse.

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 exponente que expandirá la función `expand`. Véase también `maxnegex`.

multiplicative Declaración

La instrucción `declare (f, multiplicative)` indica al simplificador de Maxima que f es multiplicativa.

- 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` ($x[i], i, m, n$).

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`.

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.

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.

noun

Declaración

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.

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`.

numer

Símbolo especial

El símbolo `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`.

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.

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**.

opsubst Variable opcional

Valor por defecto: **true**

Si **opsubst** vale **false**, **subst** no sustituye el operador de una expresión, de manera que (**opsubst: false, subst (x^2, r, r+r[0])**) trabajará correctamente.

outative Declaración

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**.

posfun Declaración

La instrucción **declare (f, posfun)** declara a *f* como función positiva, de forma que **is (f(x) > 0)** devolverá **true**.

radcan (*expr*) Función

Simplifica la expresión *expr*, que puede contener logaritmos, exponentiales 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 consumir 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.

Si `%e_to_numlog` vale `true`, `%e^(r*log(expr))` se reduce a `expr^r` si `r` es un número racional.

Si `radexpand` vale `false`, ciertas transformaciones se inhiben; `radcan(sqrt(1-x))` se mantiene como `sqrt(1-x)` y no se reduce a `%i sqrt(x-1)`, o `radcan(sqrt(x^2 - 2*x + 1))` se mantiene como `sqrt(x^2 - 2*x + 1)` sin reducirse a `x - 1`.

La instrucción `example(radcan)` muestra algunos ejemplos.

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`.

radsubstflag

Variable opcional

Valor por defecto: `false`

Si `radsubstflag` vale `true` se permite a `ratsubst` hacer la sustitución `u` por `sqrt(x)` in `x`.

rassociative

Declaración

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.

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.

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 para conjunto de tales sumatorios. 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**.

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  
          ===  ===  
          \      \  
          >      >    f(i1) g(i2)  
          /      /  
          ===  ===  
          i1 = 0 i2 = 0  
(%i3) sum (f (i), i, 0, m)^2;  
          m      m  
          ===  ===  
          \      \  
          >      >    f(i3) f(i4)  
          /      /  
          ===  ===  
          i3 = 0 i4 = 0
```

sumsplitfact

Variable opcional

Valor por defecto: **true**

Si **sumsplitfact** vale **false**, **minfactorial** se aplica después de **factcomb**.

symmetric

Declaración

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**.

unknown (*expr*)

Función

Devuelve **true** si y sólo si *expr* contiene un operador o función no reconocido por el simplificador de Maxima.

8 Gráficos

8.1 Funciones y variables para gráficos

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

Función

Dibuja las curvas de nivel *expr* en el rectángulo *x_range* por *y_range*. Cualesquiera otros argumentos adicionales se tratan como en **plot3d**.

contour_plot sólo trabaja cuando se utilizan **gnuplot** o **gnuplot_pipes**.

Véase también **implicit_plot**.

Ejemplos:

```
(%i1) contour_plot (x^2 + y^2, [x, -4, 4], [y, -4, 4]);
(%o1)
(%i2) contour_plot (sin(y) * cos(x)^2, [x, -4, 4], [y, -4, 4]);
(%o2)
(%i3) F(x, y) := x^3 + y^2;
            3      2
(%o3)           F(x, y) := x  + y
(%i4) contour_plot (F, [u, -4, 4], [v, -4, 4]);
(%o4)
(%i5) contour_plot (F, [u, -4, 4], [v, -4, 4],
                     [gnuplot_preamble, "set size ratio -1"]);
(%o5)
(%i6) set_plot_option ([gnuplot_preamble,
                        "set cntrparam levels 12"])$
(%i7) contour_plot (F, [u, -4, 4], [v, -4, 4]);
```

in_netmath

Variable opcional

Valor por defecto: **false**

Si **in_netmath** vale **true**, **plot3d** imprime salida de OpenMath en la consola si **plot_format** vale **openmath**, en caso contrario, **in_netmath** (incluso si vale **true**) deja de tener efecto alguno.

La variable **in_netmath** no afecta a **plot2d**.

plot2d (*expr, x_range, ..., options, ...*)

Función

plot2d (*[expr_1, ..., expr_n], ..., options, ...*)

Función

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

Función

Donde *expr*, *expr_1*, ..., *expr_n* pueden ser expresiones, funciones u operadores de Maxima o Lisp, o una lista de cualquiera de las siguientes formas: **[discrete, [x₁, ..., x_n], [y₁, ..., y_n]]**, **[discrete, [[x₁, y₁], ..., [x_n, y_n]]** 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 gráficamente la expresión *expr* o expresiones *[name_1, ..., name_n]*. Las expresiones que no sean de tipo paramétrico o discreto 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]. El gráfico mostrará el eje horizontal acotado por los valores *min* y *max*.

La expresión a ser representada puede ser dada en la forma discreta o paramétrica, esto es, como una lista que comienza con las palabras **discrete** o **parametric**. La clave *discrete* debe seguirse de dos listas de valores, ambas de igual longitud, conteniendo las coordenadas horizontales y verticales del conjunto de puntos; alternativamente, las coordenadas de cada punto pueden darse como listas de dos valores, todas ellas formando a su vez una lista. La clave *parametric* debe seguirse de dos expresiones, *x_expr* y *y_expr*, junto con un rango de la forma [var, min, max]; ambas expresiones deben depender únicamente de la variable cuyo nombre aparece en el rango. El gráfico mostrará los pares [x_expr, y_expr] según var varía de *min* a *max*.

El rango del eje vertical no es necesario especificarlo. Es una más de las opciones de la función, siendo su sintaxis [y, min, max], mostrando entonces la gráfica el rango completo, incluso si la función no alcanza estos valores. En caso de no especificarse el rango vertical en **set_plot_option**, se establecerá de forma automática como aquel rango en el que la función toma sus valores.

Cualesquiera otras opciones deben ser listas, comenzando con el nombre de la opción. La opción *xlabel* puede utilizarse para darle una etiqueta al eje horizontal; si no se usa esta opción, el eje horizontal será etiquetado con el nombre de la variable especificada en *x_range*.

Del mismo modo se puede asignar una etiqueta al eje vertical con la opción *ylabel*. Si sólo hay una expresión a ser representada y no se ha hecho uso de la opción *ylabel*, el eje vertical será etiquetado con la expresión a ser representada, a menos que sea muy larga, o con el texto “discrete data”, en caso de gráficos de puntos. Si la expresión es de tipo paramétrico, las dos expresiones que dan las coordenadas horizontal y vertical serán utilizadas para etiquetar ambos ejes.

Las opciones [**logx**] y [**logy**] no necesitan parámetros, permitiendo que los ejes horizontal y vertical se dibujen en la escala logarítmica.

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.

Por defecto, las funciones se dibujarán como un conjunto de segmentos lineales uniendo los puntos que bien se dan en formato *discrete*, o que se calculan automáticamente a partir de la expresión dada, de acuerdo con el tamaño muestral indicado por la opción *nticks*. Asimismo, la opción *style* puede utilizarse para mostrar los puntos aislados, o éstos junto con los segmentos que los unen.

Hay varias opciones globales almacenadas en la lista *plot_options*, las cuales se pueden modificar con la función **set_plot_option**; cualquiera de estas opciones puede ignorarse con las opciones que se utilicen desde el comando *plot2d*.

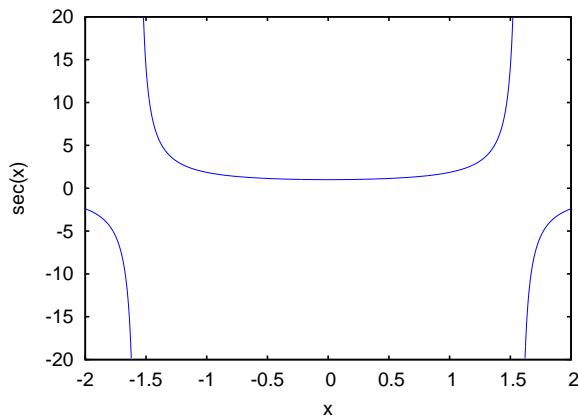
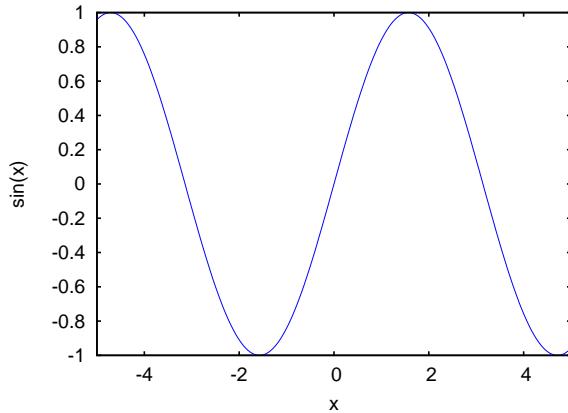
Las funciones a ser representadas pueden especificarse con el nombre de una función u operador de Maxima o de Lisp, con una expresión lambda de MMaxima, o con una expresión válida de maxima. En caso de especificarse con un nombre o expresión lambda, la función debe ser tal que dependa de un solo argumento.

Ejemplos:

Gráficos de funciones ordinarias.

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

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



Especificación de funciones por su nombre.

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

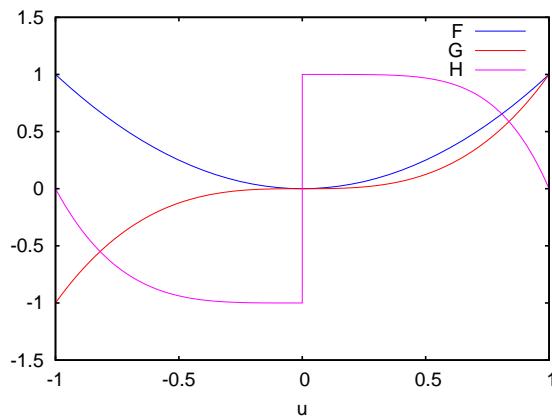
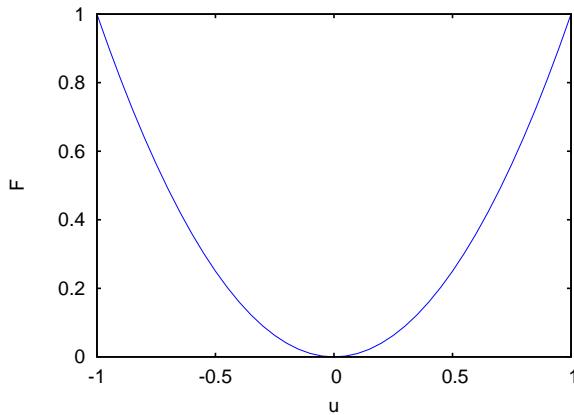
(%i4) :lisp (defun |$g| (x) (m* x x x))  

$g  

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

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

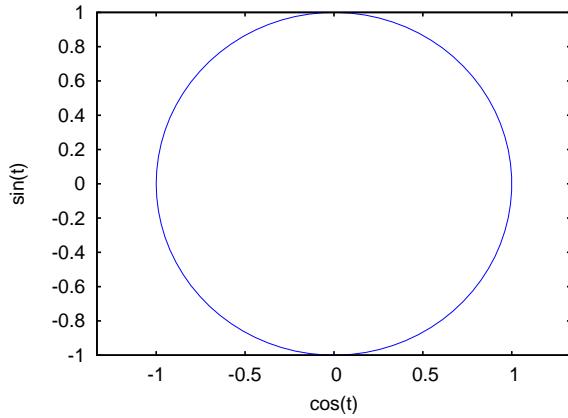
```
(%i17) plot2d ([F, G, H], [u, -1, 1], [y, -1.5, 1.5])$
```



Se puede representar una circunferencia como una función paramétrica de parámetro t . No es necesario especificar el rango del eje horizontal, pues el propio rango de t determina el dominio. No obstante, ya que las longitudes de los ejes horizontal y vertical están en una proporción de 4 a 3, se utilizará la opción `xrange` para conseguir la misma escala en ambos ejes:

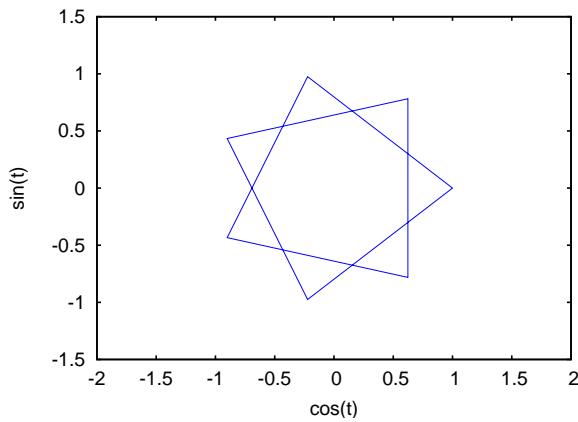
```
(%i18) plot2d ([parametric, cos(t), sin(t)], [t,-%pi,%pi],
```

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



Si se repite el mismo gráfico con solo 8 puntos y se extiende el rango del parámetro para que dé dos vueltas, se tiene el dibujo de una estrella:

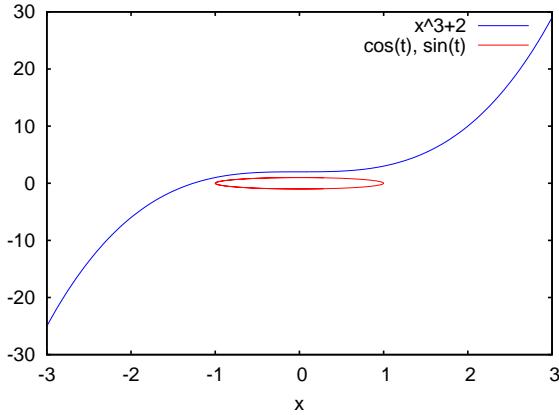
```
(%i9) plot2d ([parametric, cos(t), sin(t), [t, -%pi*2, %pi*2],
[nticks, 8]], [x, -2, 2], [y, -1.5, 1.5])$
```



Combinación del gráfico de un polinomio cúbico y de una circunferencia paramétrica:

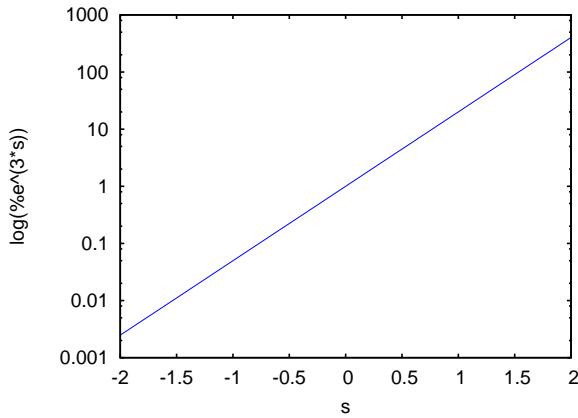
```
(%i10) plot2d ([x^3+2, [parametric, cos(t), sin(t), [t, -5, 5],
```

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



Ejemplo de gráfico logarítmico:

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



Ejemplos de gráficos de puntos, empezando por la definición de las coordenadas de cinco puntos en los dos formatos admisibles:

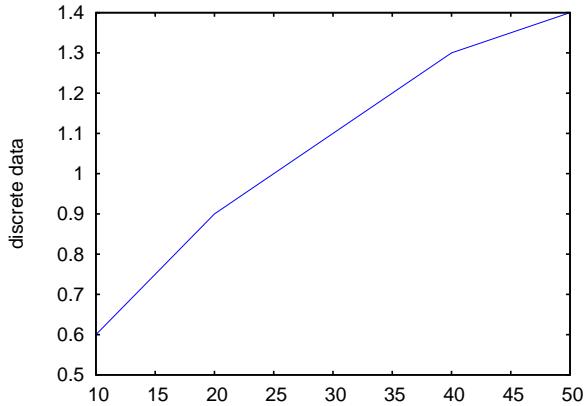
```
(%i12) xx:[10, 20, 30, 40, 50]$  

(%i13) yy:[.6, .9, 1.1, 1.3, 1.4]$  

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

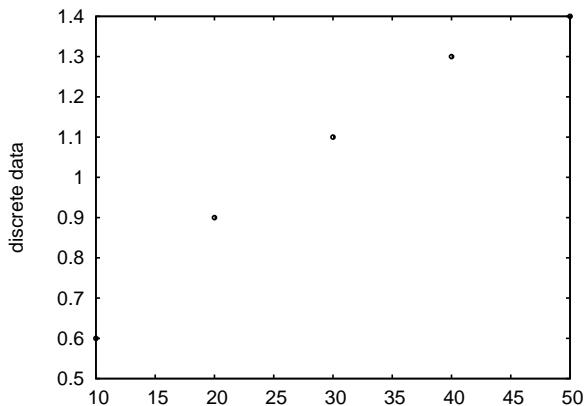
Representación de los puntos unidos por segmentos:

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



Representación de los puntos aislados, ilustrando también la segunda forma de especificar las coordenadas:

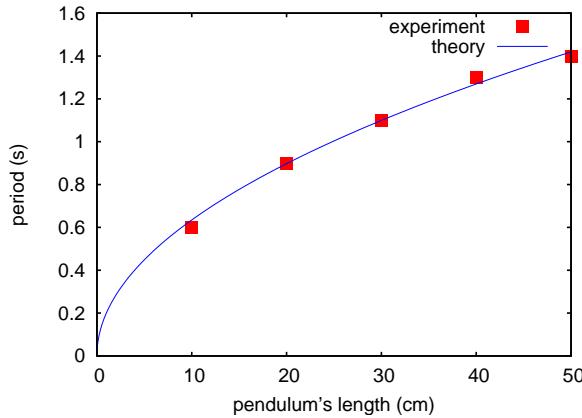
```
(%i16) plot2d([discrete, xy], [style, points])$
```



El gráfico de los puntos se puede mostrar conjuntamente con el de la función teórica que los predice:

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

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



El significado de los tres números después de la opción “points” es el siguiente: 5, radio de los puntos; 2, índice del color (rojo); 6, tipo de objeto utilizado (cuadrados sólidos). Los dos números después de la opción “lines” hacen referencia al ancho de la línea (1 punto) y al color (1 para el azul).

Véase también `plot_options`, que describe las opciones gráficas, junto con más ejemplos.

xgraph_curves (list)

Función

Dibuja el conjunto de puntos de la lista del argumento *list* con el programa xgraph. Si el programa xgraph no está instalado, este comando producirá un error.

El conjunto de puntos puede ser de la forma

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

o

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

Un conjunto de puntos también puede contener símbolos con etiquetas u otra información.

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

dibuja los tres conjuntos de puntos como tres curvas.

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

construye el conjunto de puntos, declara que no haya segmentos rectilíneos entre ellos y que se utilicen píxeles grandes. Véase el manual de xgraph para más opciones.

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

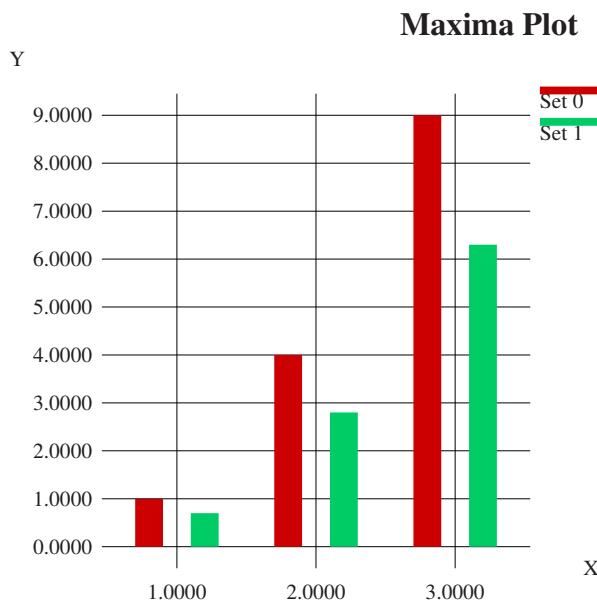
construye una etiqueta con el contenido “ x^2+y ” para este conjunto particular de puntos. Las comillas dobles “” al comienzo son las que le indican a xgraph que se trata de una etiqueta.

```
pt_set: append ([concat ("TitleText: Datos muestrales")], [x0, ...]);
```

establece el título principal del gráfico como “Datos muestrales” en lugar de “Maxima Plot”.

Para hacer un gráfico de barras con columnas de 0.2 unidades de ancho y para dibujar dos diagramas diferentes de este tipo:

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



Se utiliza un fichero temporal ‘xgraph-out’.

plot_options

Variable del sistema

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 de aquélla. En algunos casos el valor asignado es a su vez una lista, que puede contener varios elementos.

Las opciones gráficas que reconocen `plot2d` y `plot3d` son:

- Opción: `plot_format`

Determina qué programa gráfico se va a utilizar con `plot2d` y `plot3d`.

- Valor: `gnuplot` (es el valor por defecto en Windows) Gnuplot es el programa por defecto y el más avanzado. Requiere de una instalación externa de gnuplot.

- Value: **gnuplot_pipes** (es el valor por defecto en sistemas distintos de Windows) Es similar al formato **gnuplot**, excepto que la comunicación con Gnuplot se hace por medio de una tubería. Se debería utilizar para ver gráficos en pantalla; para guardar gráficos en archivos, mejor utilizar el formato **gnuplot**.
- Valor: **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.
- Valor: **openmath** Openmath es un programa gráfico escrito en Tcl/Tk. Este formato lo suministra el paquete Xmaxima, que se distribuye junto con Maxima; se deberá instalar Xmaxima si se quiere usar este formato. No sólo se puede utilizar Openmath desde Xmaxima, sino también desde cualquier otro interfaz gráfico para Maxima.
- Opción: **run_viewer**
Controla si el visor apropiado para la salida gráfica debe ejecutarse o no.
 - Valor por defecto: **true**
Ejecuta el visor.
 - Valor: **false**
No ejecuta el visor.
- Opción: **y**
Rango vertical del gráfico.
Ejemplo:

$$[y, - 3, 3]$$

Establece el rango vertical como $[-3, 3]$.
- Opción: **plot_realpart** Si **plot_realpart** vale **true**, se representará la parte real de un valor complejo x , lo cual equivale a representar **realpart(x)** en lugar de x . Si vale **false**, sólo se representarán aquellos valores con parte imaginaria nula, ignorando así cualesquiera valores complejos.
Ejemplo:

$$\begin{aligned} \text{plot2d}(\log(x), [x, -5, 5], [\text{plot_realpart}, \text{false}]); \\ \text{plot2d}(\log(x), [x, -5, 5], [\text{plot_realpart}, \text{true}]); \end{aligned}$$

El valor por defecto es **false**.
- Opción: **nticks**
En **plot2d**, es el número inicial de puntos utilizados por el procedimiento adaptativo para la representación de funciones. También es el número de puntos a ser calculados en los gráficos paramétricos.
Ejemplo:

$$[\text{nticks}, 20]$$

El valor por defecto para **nticks** es 29.

- Opción: **adapt_depth**

Número máximo de particiones utilizado por el algoritmo adaptativo de representación gráfica.

Ejemplo:

```
[adapt_depth, 5]
```

El valor por defecto para **adapt_depth** es 10.

- Opción: **xlabel**

Etiqueta del eje horizontal en gráficos 2d.

Ejemplo:

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

- Opción: **ylabel**

Etiqueta del eje vertical en gráficos 2d.

Ejemplo:

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

- Opción: **logx**

Hace que el eje horizontal en los gráficos 2d se dibuje en la escala logarítmica. No necesita de parámetros adicionales.

- Opción: **logy**

Hace que el eje vertical en los gráficos 2d se dibuje en la escala logarítmica. No necesita de parámetros adicionales.

- Opción: **legend**

Etiquetas para las expresiones de los gráficos 2d. Si hay más expresiones que etiquetas, éstas se repetirán. Por defecto se pasarán los nombres de las expresiones o funciones, o las palabras **discrete1**, **discrete2**, ..., para gráficos de puntos.

Ejemplo:

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

- Opción: **box**

Actualmente esta opción sólo puede ir seguida del símbolo *false*, pero será utilizada en el futuro para eliminar el marco alrededor del gráfico.

Ejemplo:

```
[box, false]
```

- Opción: **style**

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 llenos; 2, circunferencias; 3, +; 4, x; 5, *; 6, cuadrados llenos; 7, cuadrados huecos; 8, triángulos llenos; 9, triángulos huecos; 10, triángulos llenos invertidos; 11, triángulos huecos invertidos; 12, rombos llenos; 13, rombos huecos.

linesdots acepta hasta cuatro parámetros: ancho de línea, radio de los puntos, color y tipo de objetos para representar puntos.

Ejemplo:

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

En este ejemplo se representará la primera (tercera, quinta, etc.) expresión con segmentos rectilíneos magenta de ancho 2, la segunda (cuarta, sexta, etc.) expresión con símbolos de suma naranja de tamaño 1 (círculos naranja en el caso de Openmath).

El estilo por defecto es *lines* de ancho 1 y diferentes colores.

- Opción: **grid** Establece el número de puntos de la retícula a utilizar en las direcciones x e y en los gráficos de tres dimensiones.

Ejemplo:

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

establece la retícula en 50 por 50 puntos. El valor por defecto es 30 por 30.

- Opción: **transform_xy**

Permite que se realicen transformaciones en los gráficos de tres dimensiones.

Ejemplo:

```
[transform_xy, false]
```

El valor por defecto de **transform_xy** es **false**. Cuando vale **false**, da el resultado de

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

La transformación **polar_xy** está definida en Maxima. Devuelve la misma transformación que

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

Opciones de Gnuplot:

Hay varias opciones gráficas que son específicas de gnuplot. Algunas de ellas son comandos propios de gnuplot que se especifican como cadenas alfanuméricas. Consultese la documentación de Gnuplot para más detalles.

- Opción: **gnuplot_term**

Establece el terminal de salida para Gnuplot.

- Valor por defecto: `default`

Gnuplot muestra el gráfico en una ventana gráfica.
- Valor: `dumb`

Gnuplot muestra el gráfico en la consola de Maxima en estilo ASCII artístico.
- Valor: `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`.
- Valor: Cualquier otro terminal admitido por Gnuplot.

Gnuplot puede generar gráficos en otros muchos formatos, tales como png, jpeg, svg etc. Para crear gráficos en cualquiera 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.
- Opción: `gnuplot_out_file`

Guarda el gráfico generado por Gnuplot en un archivo.

 - Valor por defecto: `false`

No se especifica nombre de fichero.
 - Valor: `filename`

Con `[gnuplot_out_file, "myplot.ps"]` se envía código PostScript al archivo `myplot.ps` cuando se utiliza conjuntamente con el terminal PostScript de Gnuplot.
- Opción: `gnuplot_pm3d`

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`.

Ejemplo:

```
[gnuplot_pm3d, true]
```
- Opción: `gnuplot_preamble`

Introduce instrucciones de gnuplot antes de que se haga el dibujo. Puede utilizarse cualquier comando válido de gnuplot. Si interesa introducir varios comandos se separarán con punto y coma. El ejemplo que se muestra produce un gráfico en escala logarítmica. El valor por defecto de `gnuplot_preamble` es la cadena vacía "".

Ejemplo:

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

- Opción: `gnuplot_curve_titles`

Controla los títulos dados a la clave del gráfico. El valor por defecto es `[default]`, el cual establece automáticamente los títulos para cada curva representada. Si no es `[default]`, `gnuplot_curve_titles` debe contener una lista de cadenas, cada una de las cuales es `"title 'title_string'"`. (Para desactivar la clave del gráfico, añádase `"set nokey"` a `gnuplot_preamble`.)

Ejemplo:

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

- Opción: `gnuplot_curve_styles`

Es una lista de cadenas que controlan el aspecto de las curvas, como el color, el ancho, la discontinuidad, etc., y que deben enviarse al comando `plot` de gnuplot. El valor por defecto es `["with lines 3", "with lines 1", "with lines 2", "with lines 5", "with lines 4", "with lines 6", "with lines 7"]`, que realiza un ciclo sobre un conjunto de colores diferentes. Consultese la documentación de gnuplot sobre `plot` para más información.

Ejemplo:

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

- Opción: `gnuplot_default_term_command`

Comando de gnuplot para establecer el tipo de terminal por defecto. El valor por defecto es `set term windows "Verdana" 15` en sistemas Windows, y `set term x11 font "Helvetica,16"` en sistemas X11.

Ejemplo:

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

- Opción: `gnuplot_dumb_term_command`

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.

Ejemplo:

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

- Opción: `gnuplot_ps_term_command`

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. Consultese la documentación de gnuplot para más información sobre `set term postscript`.

Ejemplo:

Todas las figuras de los ejemplos de la función `plot2d` de este manual se obtuvieron a partir de archivos Postscript generados asignándole a `gnuplot_ps_term_command` el valor

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

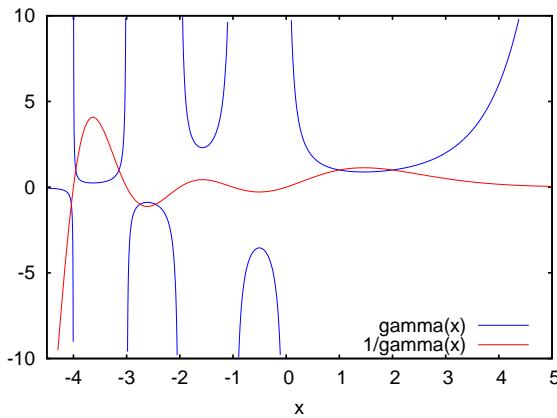
Ejemplos:

- Almacena un gráfico de $\sin(x)$ en el fichero `sin.eps`.

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

- Utiliza la opción `y` para saltarse las singularidades, así como la opción `gnuplot_preamble` para colocar la clave en la parte inferior del dibujo.

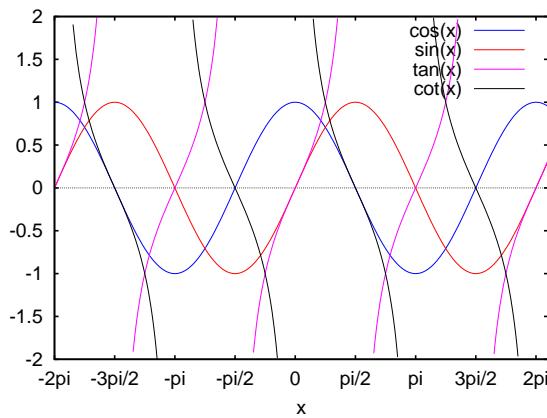
```
(%i2) plot2d ([gamma(x), 1/gamma(x)], [x, -4.5, 5], [y, -10, 10],
              [gnuplot_preamble, "set key bottom"])$
```



- Utiliza un `gnuplot_preamble` bastante complicado a fin de producir etiquetas en el eje x. (Nótese que la cadena de `gnuplot_preamble` debe introducirse sin saltos de línea.)

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

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



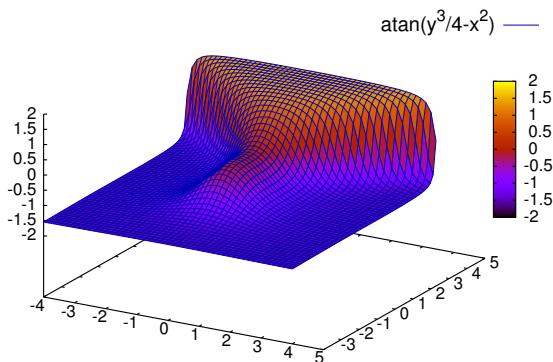
- Utiliza un `gnuplot_preamble` bastante complicado a fin de producir etiquetas en el eje x, produciendo una salida PostScript que aprovecha el formateo avanzado de texto disponible en gnuplot. (Nótese que la cadena de `gnuplot_preamble` debe introducirse sin saltos de línea.)

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

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

- Un gráfico tridimensional utilizando el terminal pm3d de gnuplot.

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

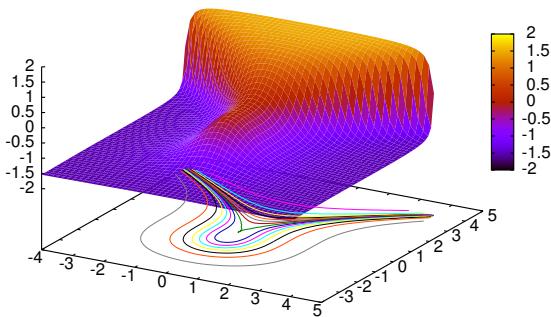


- Un gráfico tridimensional sin malla y con contornos proyectados sobre el plano inferior.

```
(%i8) my_preamble: "set pm3d at s;unset surface;set contour; \
set cntrparam levels 20;unset key"$  

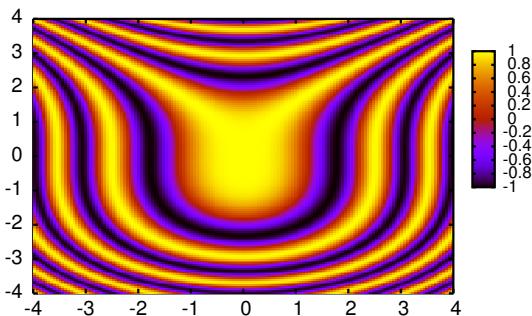
(%i9) plot3d(atan(-x^2 + y^3/4), [x, -4, 4], [y, -4, 4], \
[grid, 50, 50], [gnuplot_pm3d, true],
```

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



- Un gráfico en el que el eje z sólo se representa por el color. (Nótese que la cadena de gnuplot_preamble debe introducirse sin saltos de línea.)

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



plot3d ([expr_1, expr_2, expr_3], x_range, y_range, ..., options, ...)

Function

plot3d (expr, x_range, y_range, ..., options, ...)

Función

plot3d (name, x_range, y_range, ..., options, ...)

Función

plot3d ([expr_1, expr_2, expr_3], x_rge, y_rge)

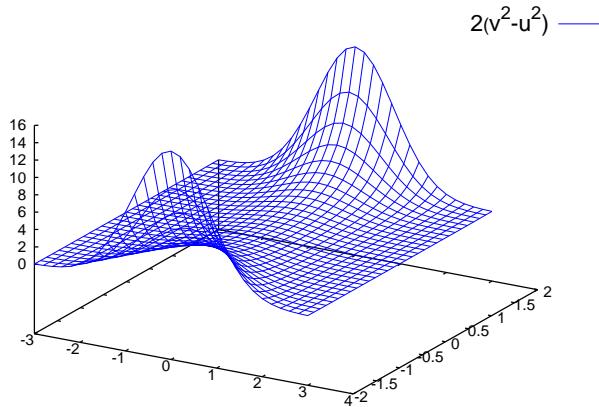
Función

plot3d ([name_1, name_2, name_3], x_range, y_range, ..., options, ...)

Función

Representa gráficamente una o tres expresiones como funciones de dos variables.

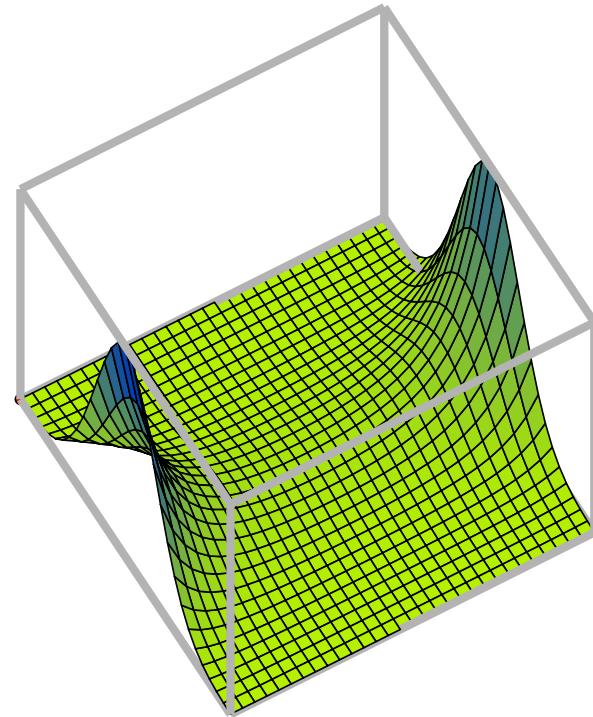
```
(%i1) plot3d (2^(-u^2 + v^2), [u, -3, 3], [v, -2, 2]);
```



dibuja $z = 2^{(-u^2+v^2)}$ con u y v variando en $[-3,3]$ y $[-2,2]$ respectivamente, y con u sobre el eje x, y con v sobre el eje y.

El mismo gráfico se puede dibujar usando openmath (si Xmaxima fué instalado):

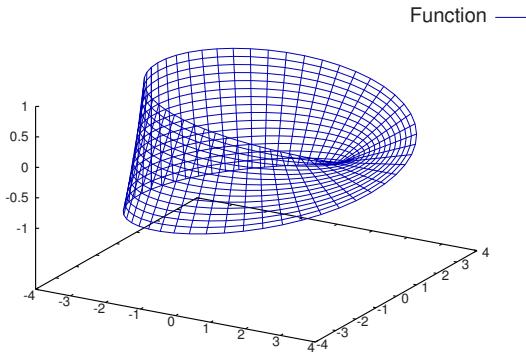
```
(%i2) plot3d (2^(-u^2 + v^2), [u, -3, 3], [v, -2, 2],
[plot_format, openmath]);
```



en este caso el ratón se puede usar para rotar el gráfico y ver la superficie desde diferentes lados.

Un ejemplo del tercer patrón de argumentos es

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

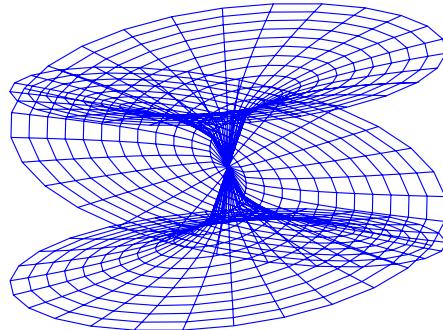


que dibuja una banda de Moebius, parametrizada por las tres expresiones dadas como primer argumento a `plot3d`. Un argumento opcional `['grid, 50, 15]` da el número de intervalos en las direcciones x e y, respectivamente.

Cuando la función a representar ha sido definida en Maxima mediante `:=` o `define`, o en Lisp por DEFUN o DEFMFUN, entonces se podrá especificar por su nombre. Las funciones definidas a nivel de LISP por DEFMSPEC, las funciones de simplificación, junto con muchas otras funciones, no pueden especificarse directamente por su nombre.

Este ejemplo muestra un gráfico de la parte real de $z^{1/3}$.

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

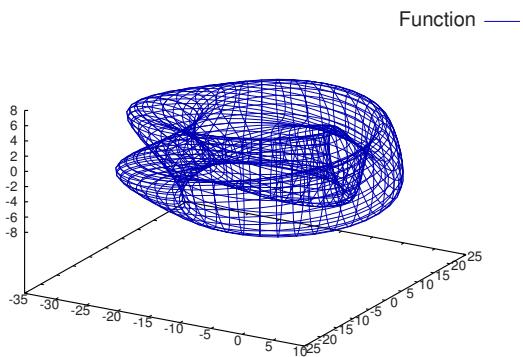


Otros ejemplos son la botella de Klein:

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

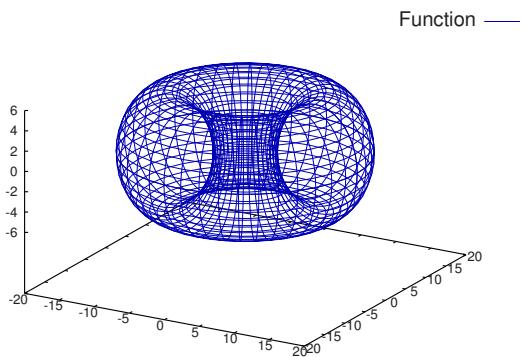
```
(%i6) expr_2: -5*sin(x)*(cos(x/2)*cos(y) + sin(x/2)*sin(2*y)
+ 3.0)$
(%i7) expr_3: 5*(-sin(x/2)*cos(y) + cos(x/2)*sin(2*y))$

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



y un toro:

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

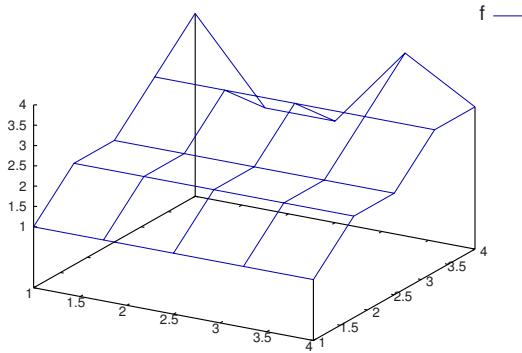


En ocasiones puede ser necesario definir una función para representarla. Todos los argumentos de `plot3d` se evalúan, de manera que puede ser difícil escribir una expresión que haga lo que el usuario realmente quiere; en tales casos facilita las cosas definir previamente la función.

```
(%i13) M: matrix([1, 2, 3, 4], [1, 2, 3, 2], [1, 2, 3, 4],
[1, 2, 3, 3])$
```

```
(%i14) f(x, y) := float (M [?round(x), ?round(y)])$  

(%i15) plot3d (f, [x, 1, 4], [y, 1, 4], ['grid, 4, 4])$
```



Véase `plot_options` para más ejemplos.

`make_transform (vars, fx, fy, fz)`

Función

Devuelve una función apropiada para la función de transformación de `plot3d`. Debe usarse con la opción gráfica `transform_xy`.

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

es una transformación para pasar a coordenadas polares.

`set_plot_option (option)`

Función

Asigna un valor a una de las variables globales que controlan los gráficos. El argumento `option` se especifica como una lista de dos o más elementos, en la que el primero es el nombre de una de las opciones de la lista `plot_options`.

La función `set_plot_option` evalúa sus argumentos y devuelve `plot_options` tal como queda después de la actualización.

Véanse también `plot_options`, `plot2d` y `plot3d`.

Ejemplos:

Se modifican los valores de `grid` y `x`. Si a un nombre de opción de `plot_options` tiene ya un valor asignado, hacerlo preceder de un apóstrofo para evitar su evaluación.

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

(%o1) [[x, - 1.755559702014E+305, 1.755559702014E+305],  

[y, - 1.755559702014E+305, 1.755559702014E+305], [t, - 3, 3],  

[grid, 30, 40], [transform_xy, false], [run_viewer, true],  

[plot_format, gnuplot], [gnuplot_term, default],  

[gnuplot_out_file, false], [nticks, 10], [adapt_depth, 10],  

[gnuplot_pm3d, false], [gnuplot_preamble, ],  

[gnuplot_curve_titles, [default]],  

[gnuplot_curve_styles, [with lines 3, with lines 1,  

with lines 2, with lines 5, with lines 4, with lines 6,  

with lines 7]], [gnuplot_default_term_command, ],
```

```
[gnuplot_dumb_term_command, set term dumb 79 22],
[gnuplot_ps_term_command, set size 1.5, 1.5;set term postscript #
eps enhanced color solid 24]]
(%i2) x: 42;
(%o2) 42
(%i3) set_plot_option ([x, -100, 100]);
(%o3) [[x, - 100.0, 100.0], [y, - 1.755559702014E+305,
1.755559702014E+305], [t, - 3, 3], [grid, 30, 40],
[transform_xy, false], [run_viewer, true],
[plot_format, gnuplot], [gnuplot_term, default],
[gnuplot_out_file, false], [nticks, 10], [adapt_depth, 10],
[gnuplot_pm3d, false], [gnuplot_preamble, ],
[gnuplot_curve_titles, [default]],
[gnuplot_curve_styles, [with lines 3, with lines 1,
with lines 2, with lines 5, with lines 4, with lines 6,
with lines 7]], [gnuplot_default_term_command, ],
[gnuplot_dumb_term_command, set term dumb 79 22],
[gnuplot_ps_term_command, set size 1.5, 1.5;set term postscript #
eps enhanced color solid 24]]
```

Funciones para trabajar con 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 <i>s</i> , entonces <i>s</i> 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 .	

9 Lectura y escritura

9.1 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
```

9.2 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: `save`, `load`, `loadfile`, `stringout`, `batch`, `demo`, `writefile`, `closefile` y `appendfile`.

9.3 Funciones y variables para lectura y escritura

	Variable del sistema
<code>--</code>	
<code>--</code>	es la expresión de entrada que está siendo actualmente evaluada. Esto es, mientras se está evaluando una expresión de entrada, <code>--</code> es igual a <code>expr</code> .
A <code>--</code>	se le asigna la expresión de entrada antes de que ésta sea simplificada o evaluada. Sin embargo, el valor de <code>--</code> es simplificado, pero no evaluado, cuando su valor es mostrado en el terminal.
La variable <code>--</code>	es reconocida por <code>batch</code> y por <code>load</code> . Cuando un fichero es procesado por <code>batch</code> , la variable <code>--</code> tiene el mismo significado que en el modo interactivo. Cuando un fichero es procesado por <code>load</code> , a la variable <code>--</code> 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 <code>--</code> una expresión de entrada del fichero que está siendo procesado. En particular, si <code>load (filename)</code> 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. Por ejemplo,

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

block ([prev], prev: integrate (x^5, x),  

      ev (prev, x=2) - ev (prev, x=1));
```

devuelven el mismo resultado **21/2**.

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

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

devuelve **7*a^n**.

Dentro de una sentencia compuesta, el valor de **%%** puede inspeccionarse en un punto de interrupción que se abra ejecutando la función **break**. Por ejemplo, en el punto de interrupción abierto por

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

introduciendo **%%**; se obtiene **42**.

En la primera sentencia de una sentencia compuesta, o fuera de una sentencia compuesta, **%%** no está definido.

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.

Véase también **%**.

%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 .	
%th (i)	Función
Es el valor de la expresión de la <i>i</i> -ésima salida anterior. Esto es, si la siguiente expresión a calcular es la salida <i>n</i> -ésima, %th (m) es la salida $(n - m)$ -ésima.	
La función %th es útil en archivos tipo batch o para hacer referencia a un grupo de expresiones de salida. Por ejemplo,	
block (s: 0, for i:1 thru 10 do s: s + %th (i))\$	
asigna a s la suma de las diez últimas expresiones de salida.	
La variable %th es reconocida por batch y por load . Cuando un fichero es procesado por batch , la variable %th tiene el mismo significado que en el modo interactivo. Cuando un fichero es procesado por load , a la variable %th 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 %th una expresión de entrada del fichero que está siendo procesado.	
Véase también % .	
?	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.	
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.	
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.

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

Lee expresiones de Maxima desde `filename` y las evalúa. La función `batch` busca `filename` en la lista `file_search_maxima`. Véase `file_search`.

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 líneas de instrucciones, 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.

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**.

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 apástrofo ' evita la evaluación.

```
(%i1) y: 7$  
(%i2) z: 88$  
(%i3) concat (y, z/2);  
(%i3) 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;  
(%o8) b  
(%i9) %, numer;  
(%o9) 123
```

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
```

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.

dispcon (*tensor_1, tensor_2, ...*)
dispcon (*all*)

Función
 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.

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 subindexadas o llamadas a funciones. Véase también **disp**.

```
(%i1) display(B[1,2]);
          2
          B      = X - X
  1, 2
(%o1)                                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.

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 ocultan 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/b;	a -	- 1 a b
	b	
sqrt(x);	sqrt(x)	x ^{1/2}
X*4/3;	4 X --- 3	4 - X 3

disptterms (*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, **disptterms** (P₁ + P₂ + ...) muestra P₁, debajo P₂, etc. Cuando una expresión exponencial es demasiado ancha para ser representada como A^B, si no se utiliza **disptterms**, entonces aparecerá como **expt** (A, B) (o como **ncexpt** (A, B), en lugar de A^B).

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
  -----
          cos(X - 1) + 1
(%i4)

(%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 asignan 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`.

expt (<i>a, b</i>)	Funcióñ
Si una expresión exponencial es demasiado ancha para ser mostrada como a^b aparecerá como <code>expt (a, b)</code> (o como <code>ncexpt (a, b)</code> en lugar de $a^{^b}$).	
Las funciones <code>expt</code> y <code>ncexpt</code> no se reconocen en una entrada.	

exptdispflag	Variable opcional
Valor por defecto: <code>true</code>	
Si <code>exptdispflag</code> vale <code>true</code> , Maxima muestra las expresiones con exponentes negativos como cocientes; por ejemplo, $X^{(-1)}$ se muestra como $1/X$.	

filename_merge (<i>path, filename</i>)	Funcióñ
Construye una ruta modificada a partir de <i>path</i> y <i>filename</i> . Si la componente final de <i>path</i> es de la forma <code>###.something</code> , la componente se reemplaza con <i>filename.something</i> . En otro caso, la componente final se reemplaza simplemente por <i>filename</i> .	
El resultado es un objeto Lisp de tipo <i>pathname</i> .	

file_search (<i>filename</i>)	Funcióñ
file_search (<i>filename, pathlist</i>)	Funcióñ
La función <code>file_search</code> busca el archivo <i>filename</i> y devuelve su ruta como una cadena; si no lo encuentra, <code>file_search</code> devuelve <code>false</code> . La llamada <code>file_search (filename)</code> busca en los directorios de búsqueda por defecto, que son los especificados por las variables <code>file_search_maxima</code> , <code>file_search_lisp</code> y <code>file_search_demo</code> .	
La función <code>file_search</code> analiza primero si el nombre del argumento existe antes de hacerlo coincidir con los comodines de los patrones de búsqueda de archivos. Véase <code>file_search_maxima</code> para más información sobre patrones de búsqueda de archivos.	
El argumento <i>filename</i> 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`.

<code>file_search_maxima</code>	Variable opcional
<code>file_search_lisp</code>	Variable opcional
<code>file_search_demo</code>	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`.

<code>file_type (filename)</code>	Función
-----------------------------------	---------

Devuelve una descripción del contenido de `filename` basada en la extensión, sin intentar abrir el archivo para inspeccionar su contenido.

El valor que la función retorna puede ser cualquiera de los siguientes: `object`, `lisp` o `maxima`. Si la extensión comienza por `m` o `d`, `file_type` devuelve `maxima`. Si la extensión comienza por `l`, `file_type` devuelve `lisp`. En cualquier otro caso, `file_type` devuelve `object`.

<code>grind (expr)</code>	Función
<code>grind</code>	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 lo 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 las expresiones de entrada.

La función `grind` evalúa sus argumentos.

Ejemplos:

```
(%i1) aa + 1729;
(%o1)
(%i2) grind (%);
aa+1729$                                done
(%i3) [aa, 1729, aa + 1729];
(%o3)                               [aa, 1729, aa + 1729]
(%i4) grind (%);
[aa,1729,aa+1729]$                        done
(%i5) matrix ([aa, 17], [29, bb]);
      [ aa  17 ]
(%o5)                               [             ]
      [ 29   bb ]
(%i6) grind (%);
matrix([aa,17],[29,bb])$                  done
(%o6)
(%i7) set (aa, 17, 29, bb);
(%o7)                               {17, 29, aa, bb}
(%i8) grind (%);
{17,29,aa,bb}$                           done
(%o8)
(%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:

íbase 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; 10  
(%o3) abcd; 4096  
(%i4) abcd;  
(%o4) abcd  
(%i5) symbolp (abcd); true  
(%o5) true  
(%i6) 0abcd;  
(%o6) 43981  
(%i7) symbolp (0abcd); false  
(%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
```

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 carácter sencillo.

```
(%o1)      3      2      2      3
          b  + 3 a b  + 3 a  b + a
(%input2)
```

Véase también `labels`.

lisp (*expr_1, ..., expr_n*)

Función

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

Véase también `disp`.

```
(%i1) e: (a+b)^3;
(%o1)                               3
                               (b  + a)
(%i2) f: expand (e);
(%o2)      3      2      2      3
          b  + 3 a b  + 3 a  b + a
(%i3) lisp (e, f);
(%t3)                               3
                               (b  + a)

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

(%o4) [%t3, %t4]
(%i4) %t3;
(%o4)                               3
                               (b  + a)
(%i5) %t4;
(%o5)      3      2      2      3
          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. Cada 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 `lisp` asigna una etiqueta a cada ecuación y devuelve la lista de etiquetas.

Véase también `display`.

```
(%i1) e: (a+b)^3;
(%o1)                               3
                               (b  + a)
(%i2) f: expand (e);
(%o2)      3      2      2      3
          b  + 3 a b  + 3 a  b + a
(%i3) ldisplay (e, f);
(%t3)                               3
                               e = (b  + a)
```

```

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

(%o4)                               [%t3, %t4]
(%i4) %t3;

          3
(%o4)                               e = (b + a)
(%i5) %t4;

          3      2      2      3
(%o5) f = b + 3 a b + 3 a b + a

```

linechar

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 cadena o símbolo, no necesariamente un carácter simple.

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

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;
          foo + bar
(%o2)

(%i3) lispdisp: true$ 
(%i4) ?foo + ?bar;
          ?foo + ?bar
(%o4)

```

load (*filename*)

Función

Evaluá 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.

loadfile (filename)

Función

Evaluá 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 cuando un archivo se carga automáticamente. Véase `setup_autoload`.
- Si `loadprint` vale `false`, nunca mostrará mensajes.

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)                            0FFFFFFF
(%i7) obase : 36;
(%o7)                               10
(%i8) 36^8 - 1;
(%o8)                            0ZZZZZZZZ
```

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 carácter sencillo.

```
(%i1) outchar: "output";
(%o1)                               output
(%i2) expand ((a+b)^3);
(%o2)          3           2           2           3
         b + 3 a b + 3 a b + a
(%i3)
```

Véase también **labels**.

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 (por ejemplo, **values**, **functions**) excepto allí donde sea necesario cuando el archivo sea cargado en memoria.

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)
```

print (*expr_1*, ..., *expr_n*)

Función

Evaluá 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)))$
```

$$(a+b)^3 \text{ is } b^3 + 3ab^2 + 3a^2b + a^3 \text{ log}(a^{10}/b) \text{ is }$$

$$10 \log(a) - \log(b)$$

```
(%i2) r;
```

$$10 \log(a) - \log(b)$$

```
(%i3) disp ("(a+b)^3 is", expand ((a+b)^3), "log (a^10/b) is",
            radcan (log (a^10/b)))$
```

$$(a+b)^3 \text{ is }$$

$$b^3 + 3ab^2 + 3a^2b + a^3$$

$$\log(a^{10}/b) \text{ is }$$

$$10 \log(a) - \log(b)$$
printfile (*path*)

Función

Envía el fichero al que hace referencia 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 encontrado.

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**.

reveal (*expr, nivel*)

Función

Reemplaza partes de *expr* al *nivel* especificado y las sustituye 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**.

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)
-----+
          2 %e      2 b      2 a
          + %e      + %e
(%i2) reveal (e, 1);
(%o2)                               Quotient
(%i3) reveal (e, 2);
(%o3)                               Sum(3)
-----+
          Sum(3)
(%i4) reveal (e, 3);
          Expt + Negterm + Expt
(%o4)
-----+
          Product(2) + Expt + Expt
(%i5) reveal (e, 4);
          2           2
          b - Product(3) + a
(%o5)
-----+
          Product(2)      Product(2)
          2 Expt + %e      + %e
(%i6) reveal (e, 5);
          2           2
          b - 2 a b + a
(%o6)
-----+
          Sum(2)      2 b      2 a
          2 %e      + %e      + %e
(%i7) reveal (e, 6);
          2           2
          b - 2 a b + a
(%o7)
-----+
          b + a      2 b      2 a
          2 %e      + %e      + %e
```

rmxchar

Variable opcional

Valor por defecto:]

La variable **rmxchar** es el carácter que se dibuja al lado derecho de una matriz.

Véase también **lmxchar**.

save (<i>filename, name_1, name_2, name_3, ...</i>)	Función
save (<i>filename, values, functions, labels, ...</i>)	Función
save (<i>filename, [m, n]</i>)	Función
save (<i>filename, name_1=expr_1, ...</i>)	Función
save (<i>filename, all</i>)	Función
save (<i>filename, name_1=expr_1, name_2=expr_2, ...</i>)	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 mensaje 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.

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`.

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.

showratvars (expr) Función
 Devuelve una lista de las variables de expresiones canónicas racionales (CRE) en la expresión `expr`.
 Véase también `ratvars`.

stardisp Variable opcional
 Valor por defecto: `false`
 Si `stardisp` vale `true`, la multiplicación se muestra con un asterisco `*` entre los operandos.

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 muestran 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."
```

stringout (<i>filename, expr_1, expr_2, expr_3, ...</i>)	Función
stringout (<i>filename, [m, n]</i>)	Función
stringout (<i>filename, input</i>)	Función
stringout (<i>filename, functions</i>)	Función
stringout (<i>filename, values</i>)	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.

tex (<i>expr</i>)	Función
tex (<i>expr, destination</i>)	Función
tex (<i>expr, false</i>)	Función
tex (<i>label</i>)	Función
tex (<i>label, destination</i>)	Función
tex (<i>label, false</i>)	Función

Devuelve la expresión en un formato apropiado 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);
(%o1)

$$\frac{-\log(x^2 - x + 1)^2}{6} + \frac{\operatorname{atan}\left(\frac{2x - 1}{\sqrt{3}}\right)}{\sqrt{3}} + \frac{\log(x + 1)}{3}$$

(%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 postfixo..

```
(%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 ", " \\backslash ", " \\backslash ,"], matchfix);
(%o5)                                [ \langle , \rangle , \backslash , \backslash , ]
```

```
(%i6) tex (<<a>>);
$$ \langle a \rangle
(%o6)                                     false
(%i7) tex (<<a, b>>);
$$ \langle a \rangle, \langle 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
```

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)&")
```

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**.

with_stdout (*f, expr_1, expr_2, expr_3, ...*)

Función

with_stdout (*s, expr_1, expr_2, expr_3, ...*)

Función

Evaluá *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`.

10 Aritmética de punto flotante

10.1 Funciones y variables para la aritmética de punto flotante

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**.

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**.

bfpsi (*n, z, fprec*)

Función

bfpsi0 (*z, fprec*)

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, fprec*) equivale a **bfpsi** (0, *z, fprec*).

Estas funciones devuelven valores "bigfloat". La variable *fprec* es la precisión "bigfloat" del valor de retorno.

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 .	
cbffac (<i>z, fpprec</i>)	Función
Calcula el factorial de números complejos de punto flotante grandes.	
La instrucción load ("bffac") carga esta función.	
float (<i>expr</i>)	Función
Convierte los enteros, números racionales y los decimales de punto flotante grandes ("bigfloats") que están presentes en <i>expr</i> a números de punto flotante. También actúa como símbolo evflag .	
float2bf	Variable optativa
Valor por defecto: false	
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 (<i>expr</i>)	Función
Devuelve true si <i>expr</i> es un número de punto flotante, en caso contrario 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 (<i>bigfloats</i>). 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 los casos igual a 16.	
En el caso de los decimales de precisión ilimitada (<i>bigfloats</i>), 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.	

11 Contextos

11.1 Funciones y variables para Contextos

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.

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 $< \leq \text{equal} \text{ notequal} > = y$. 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.

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 **scalarmp (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);  
          pos  
(%i4) sign (a[1]);  
          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 (<i>contexto_1, ..., contexto_n</i>)	Función
Desactiva los contextos especificados <i>contexto_1, ..., contexto_n</i> .	
facts (<i>item</i>)	Función
facts ()	Función
Si <i>item</i> es el nombre de un contexto, facts (<i>item</i>) devuelve una lista con los hechos asociados al contexto especificado.	
Si <i>item</i> no es el nombre de un contexto, facts (<i>item</i>) devuelve una lista con los hechos conocidos acerca de <i>item</i> 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.	
features	Declaración
Maxima reconoce ciertas propiedades matemáticas sobre funciones y variables.	
La llamada declare (<i>x, foo</i>) asocia la propiedad <i>foo</i> a la función o variable <i>x</i> .	
La llamada declare (<i>foo, feature</i>) declara una nueva propiedad <i>foo</i> . Por ejemplo, declare ([rojo, verde, azul], <i>feature</i>) declara tres nuevas propiedades, <i>rojo</i> , <i>verde</i> y <i>azul</i> .	
El predicado featurep (<i>x, foo</i>) devuelve true si <i>x</i> goza de la propiedad <i>foo</i> , 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 , and antisymmetric , junto con las definidas por el usuario.	
La lista features sólo contiene propiedades matemáticas. Hay otra lista con propiedades no matemáticas; Véase status .	
forget (<i>pred_1, ..., pred_n</i>)	Función
forget (<i>L</i>)	Función
Borra los predicados establecidos por assume . Los predicados pueden ser expresiones equivalentes, pero no necesariamente idénticas, a las establecidas por assume . The predicates may be expressions equivalent to (but not necessarily identical to) those previously assumed.	
La llamada forget (<i>L</i>), siendo <i>L</i> una lista de predicados, borra todos los predicados contenidos en ella.	
killcontext (<i>contexto_1, ..., contexto_n</i>)	Función
Elimina los contextos <i>contexto_1, ..., contexto_n</i> .	
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**.

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*.

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.

12 Polinomios

12.1 Introducción a los polinomios

Los polinomios se almacenan en Maxima, bien en un formato general, bien en una forma conocida como canónica (Canonical 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 $2*X^2Y+X-3$ 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 $X+\text{SIN}(X+1)+2*\text{SQRT}(X)+1$ son X, $\text{SQRT}(X)$ y $\text{SIN}(X+1)$. Si el usuario no especifica una ordenación de las variables mediante la función **ratvars** Maxima escogerá una alfabéticamente. En general, las 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 símbolo /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.

12.2 Funciones y variables para polinomios

algebraic

Valor por defecto: **false**

Variable opcional

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.

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

Devuelve el coeficiente de x^n en *expr*. El argumento *n* puede omitirse si es igual a la unidad. El argumento *x* puede ser un átomo o una subexpresión completa de *expr*, por ejemplo **sin(x)**, **a[i+1]**, **x + y**, etc. (En este último caso, la expresión **(x + y)** debe aparecer en *expr*). En ocasiones puede ser necesario expandir o factorizar *expr* para hacer x^n explícito, lo cual no hace **coeff** automáticamente.

Ejemplos:

```
(%i1) coeff (2*a*tan(x) + tan(x) + b = 5*tan(x) + 3, tan(x));  

(%o1)                                2 a + 1 = 5  

(%i2) coeff (y + x*%e^x + 1, x, 0);  

(%o2)                                y + 1
```

combine (*expr*)

Función

Simplifica la suma *expr* combinando términos de con igual denominador reduciéndolos a un único término.

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;
(%o1)           2
                  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]);
(%o4) [7425 x^8 - 1170 x^7 + 1299 x^6 + 12076 x^5 + 22887 x^4
      - 5154 x^3 - 1291 x^2 + 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**.

faceexpand Variable opcional

Valor por defecto: **true**

La variable **faceexpand** controla si los factores irreducibles devueltos por **factor** están en formato expandido (por defecto) o recursivo (CRE normal).

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**.

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 enteros 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
              (%e + 1) (%e - %e + 1)

(%i6) factor (1 + x^4, a^2 - 2);
              2      2
              (x - a x + 1) (x + a x + 1)

(%i7) factor (-y^2*z^2 - x*z^2 + x^2*y^2 + x^3);
              2
              - (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
              (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
              - (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
              (c + (- b - 3) c + 3 b) (x + c)

              b - 2
-----  

              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

```

```

(%o11) - -----
          c - 4 c - b + 6           c - 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
          x + x + x + x + 1
(%i13) subst (a, x, %);
          4           3           2
          a + a + a + a + 1
(%i14) factor (%th(2), %);
          2           3           3           2
          (x - a) (x - a) (x - a) (x + a + a + a + 1)
(%i15) factor (1 + x^12);
          4           8           4
          (x + 1) (x - x + 1)
(%i16) factor (1 + x^99);
          2           6           3
          (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 - x + 1)

          20         19         17         16         14         13         11         10         9         7         6
          (x + x - 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*, *x₁*, *x₂*, ...)

Función

Reorganiza la suma *expr* como una suma de términos de la forma *f* (*x₁*, *x₂*, ...) * *g* donde *g* es un producto de expresiones que no contienen ningún *x_i* y *f* se factoriza.**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 re-

cuperar 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
      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 (%);
(%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);
(%o3)           a
                           x  - 1
(%i4) rat (expr);
      a/2 4      a/2 2
      (x      ) - 2 (x      ) + 1
(%o4)/R/           -----
```

$$\frac{a}{x} - 1$$

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 sola 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** determina 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 $q = \text{quotient}(r[i]/r[i+1])$ 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 <code>factor (expr, a^2+1)</code> donde a vale <code>%i</code> .	

Ejemplo:

```
(%i1) gfactor (x^4 - 1);
(%o1)           (x - 1) (x + 1) (x - \%i) (x + \%i)
```

gfactorsum (expr)	Función
Esta función es similar a <code>factorsum</code> pero aplica <code>gfactor</code> en lugar de <code>factor</code> .	

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).

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 asociadas) se realizan módulo p , utilizando el llamado sistema de módulo balanceado, en el que n módulo p se define como un entero k de $[-(p-1)/2, \dots, 0, \dots, (p-1)/2]$ si p es impar, o de $[-(p/2 - 1), \dots, 0, \dots, p/2]$ 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 permitirá que a **modulus** se le asigne cero o un entero negativo, aunque no esté clara su utilidad.

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 $[p_1, \dots, p_n]$ tal que

$$\text{lambda}([x], p_1)(\text{lambda}([x], p_2)(\dots(\text{lambda}([x], p_n)(x))\dots))$$

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);
(%o1)                                [x , x , x , x ]
(%i2) p : expand (subst (x^3 - x - 1, x, x^2 - a));
(%o2)          x - 2 x - 2 x + x + 2 x - a + 1
(%i3) polydecomp (p, x);
(%o3)          [x - a, x - x - 1]
```

La siguiente función compone $L = [e_1, \dots, e_n]$ 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
```

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 exponentiación 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);
```

```
(%o1) 
$$\frac{(y + a)(2y + x)(x^4 - 2xy^3)^2 + 1}{(x^4 - 4y^2)^2}$$

(%i2) rat (% , y , a , x);
(%o2)/R/

$$\frac{2a + 2y}{x^2 + 2y}$$

```

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.

Examples:

```
(%i1) expr: (x^2 + x + 1)/(y^2 + 7);
          2
          x  + x + 1
(%o1)
          -----
          2
          y  + 7
(%i2) ratdenomdivide: true$ 
(%i3) ratexpand (expr);
          2
          x      x      1
(%o3)      ----- + ----- + -----
          2      2      2
          y  + 7   y  + 7   y  + 7
(%i4) ratdenomdivide: false$ 
(%i5) ratexpand (expr);
          2
          x  + x + 1
(%o5)
          -----
          2
          y  + 7
(%i6) expr2: a^2/(b^2 + 3) + b/(b^2 + 3);
          2
          b      a
(%o6)      ----- + -----
          2      2
          b  + 3   b  + 3
(%i7) ratexpand (expr2);
          2
          b + a
(%o7)
          -----
          2
          b  + 3
```

ratdiff (expr, x)

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
      f (x) - f (x) + 7
(%i4) ratdiff (expr, f(x));
      2
      3 f (x) - 2 f(x)
(%i5) expr: (a + b)^3 + (a + b)^2;
      3      2
      (b + a)  + (b + a)
(%i6) ratdiff (expr, a + b);
      2      2
      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`.

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.

ratexpand (expr)
ratexpand

 Función
 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
            - 27 y + 54 x y - 36 x y + 8 x
(%o1)
(%i2) expr: (x - 1)/(x + 1)^2 + 1/(x - 1);
           x - 1           1
           ----- + -----
           2           x - 1
           (x + 1)
(%o2)
(%i3) expand (expr);
           x           1           1
           ----- - ----- + -----
           2           2           x - 1
           x + 2 x + 1   x + 2 x + 1
(%o3)
(%i4) ratexpand (expr);
           2
           2 x           2
           ----- + -----
           3   2           3   2
           x + x - x - 1   x + x - x - 1
(%o4)
```

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 pequeño.

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 returnedo 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**.

ratnump (*expr*) Función

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

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*, *x₁*, ..., *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*, *x₁*, ..., *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
              sin(-----) = %e x
              x + 1
(%o2)
              ((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
(%o5)      -----
              a
              x
```

ratsimpexpons

Variable opcional

Valor por defecto: **false**

Si **ratsimpexpons** vale **true**, **ratsimp** se aplica a los exponentes de las expresiones durante la simplificación.

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.

Ejemplos:

```
(%i1) ratsubst (a, x*y^2, x^4*y^3 + x^4*y^8);
            3           4
            a x   y + a
(%o1)
(%i2) cos(x)^4 + cos(x)^3 + cos(x)^2 + cos(x) + 1;
            4           3           2
cos (x) + cos (x) + cos (x) + cos(x) + 1
(%o2)
(%i3) ratsubst (1 - sin(x)^2, cos(x)^2, %);
            4           2
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
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 (*x₁, ..., x_n*)

Función

ratvars ()

Función

ratvars

Variable del sistema

Declara como variables principales *x₁, ..., 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 *x₁*, 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 *x₁*.

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** reinitializa la lista. La llamada **ratvars** () vacía la lista.

ratweight (*x₁, w₁, ..., 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

resultant

Variable

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 **resultant** controla qué algoritmo será utilizado para calcular la resultante.

La función **bezout** toma los mismos argumentos que **resultant** y devuelve una matriz. El determinante del valor returned es la resultante buscada.

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.

tellrat (p_{-1}, \dots, p_{-n})
tellrat ()

Función
Función

Añade al anillo de enteros algebraicos conocidos por Maxima los elementos que son soluciones de los polinomios p_{-1}, \dots, 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
(%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, \dots, x_n)

Función

Elimina de x_1, \dots, x_n las propiedades relacionadas con **tellrat**.

13 Constantes

13.1 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.	
ind	Constante
ind representa un resultado acotado indeterminado. Véase también limit . Ejemplo:	
(%i1) $\lim_{x \rightarrow 0} (\sin(1/x), x, 0);$ (%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 <i>razón áurea</i> , $(1 + \sqrt{5})/2$. 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));
(%o1)

$$\frac{\phi^n - (1 - \phi)^n}{2\phi - 1}$$

(%i2) fib (n-1) + fib (n) - fib (n+1);
(%o2)

$$- \text{fib}(n + 1) + \text{fib}(n) + \text{fib}(n - 1)$$

(%i3) fibtophi (%);
(%o3)

$$-\frac{\phi^{n+1} - (1 - \phi)^{n+1}}{2\phi - 1} + \frac{\phi^n - (1 - \phi)^n}{2\phi - 1}$$


$$+ \frac{\phi^{n-1} - (1 - \phi)^{n-1}}{2\phi - 1}$$

(%i4) ratsimp (%);
(%o4) 0
```

Por defecto, Maxima desconoce las propiedades algebraicas de ϕ . Despu s de evaluar `tellrat ($\phi^2 - \phi - 1$)` y `algebraic: true`, `ratsimp` puede simplificar algunas expresiones que contengan ϕ .

```
(%i1) e : expand ((%phi^2 - %phi - 1) * (A + 1));
(%o1)

$$\frac{\phi^2 A - \phi A - A + \phi^2 - \phi - 1}{2}$$

(%i2) ratsimp (e);
(%o2)

$$(\phi^2 - \phi - 1) A + \phi^2 - \phi - 1$$

(%i3) tellrat (%phi^2 - %phi - 1);
(%o3)

$$[\phi^2 - \phi - 1]$$

(%i4) algebraic : true;
(%o4) true
(%i5) ratsimp (e);
(%o5) 0
```

%pi Constante

El s mbolo `%pi` representa la raz n entre la longitud de una circunferencia y su radio.
El valor num rico de `%pi` es el n mero decimal en coma flotante `3.141592653589793d0`.

true Constante

El s mbolo `true` representa al valor l gico "verdadero". Maxima implementa `true` con el valor `T` de Lisp.

und Constante

`und` representa un resultado indefinido.

V ase tambi n `limit`.

Ejemplo:

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


14 Logaritmos

14.1 Funciones y variables para logaritmos

%e_to_numlog

Valor por defecto: `false`

Variable opcional

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`.

li [s] (z)

Función

Representa la función polilogarítmica de orden `s` y argumento `z`, definida por la serie infinita

$$\text{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}$$

`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)
(%i2) integrate ((log (1 - t)) / t, t, 0, x);
(%o2)
(%i3) li [2] (7);
(%o3)
(%i4) li [2] (7), numer;
(%o4)      1.24827317833392 - 6.113257021832577 %i
(%i5) li [3] (7);
(%o5)
(%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 $\text{log10}(x) := \text{log}(x) / \text{log}(10)$.

La simplificación y evaluación de logaritmos se controla mediante variables globales:

logexpand - hace que $\text{log}(a^b)$ se convierta en $b*\text{log}(a)$. Si toma el valor **all**, $\text{log}(a*b)$ también se reducirá a $\text{log}(a)+\text{log}(b)$. Si toma el valor **super**, entonces $\text{log}(a/b)$ también se reducirá a $\text{log}(a)-\text{log}(b)$, siendo a/b racional y $a \neq 1$, (la expresión $\text{log}(1/b)$, 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.

lognumer - si vale **true**, entonces los argumentos de **log** que sean números decimales negativos en coma flotante se convertirán siempre a su valor absoluto antes de aplicar **log**. Si **numer** vale también **true**, entonces los argumentos enteros negativos de **log** también se convertirán en su valor absoluto.

lognegint - si vale **true** se aplica la regla $\text{log}(-n) \rightarrow \text{log}(n)+\pi*i$ siendo n un entero positivo.

%e_to_numlog - si vale **true**, r es un número racional y x una expresión, $%e^{(r*\text{log}(x))}$ 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**.

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(%);  
(%o2) a log(x2 y4)
```

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 y **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.

lognumer	Variable opcional
Valor por defecto: false	
Si lognumer vale true , entonces los argumentos de log que sean números decimales negativos en coma flotante se convertirán siempre a su valor absoluto antes de aplicar log . Si numer vale también true , entonces los argumentos enteros negativos de log también se convertirán en su valor absoluto.	
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 $-\pi < \text{carg}(x) \leq \pi$.	

15 Trigonometría

15.1 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.

15.2 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)];
(%o2)                                [0, 1, 0, (- 1)^m/2]
```

%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 <code>atan(y/x)</code> en el intervalo de <code>-%pi</code> a <code>%pi</code> .	
atanh (x)	Función
Arco tangente hiperbólica.	
atrig1	Paquete
El paquete <code>atrig1</code> 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: <code>0</code> , <code>%pi/6</code> , <code>%pi/4</code> , <code>%pi/3</code> y <code>%pi/2</code> . Los ángulos correspondientes en los otros tres cuadrantes también están disponibles. Para hacer uso de estas reglas, ejecútese <code>load(atrig1);</code> .	
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`.

Ejemplos:

```
(%i1) halfangles : false$  
(%i2) sin (x / 2);  
          x  
(%o2)      sin(-)  
                  2  
(%i3) halfangles : true$  
(%i4) sin (x / 2);  
          sqrt(1 - cos(x))  
(%o4)  -----  
          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
              - 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 **trigsing** vale **true**, se permite la simplificación de argumentos negativos en funciones trigonométricas, como en **sin(-x)**, que se transformará en **-sin(x)** sólo si **trigsing** vale **true**.

trigsimp (expr)

Función

Utiliza las identidades $\sin^2(x) + \cos^2(x) = 1$ y $\cosh^2(x) - \sinh^2(x) = 1$ para simplificar expresiones que contienen **tan**, **sec**, etc., en expresiones con **sin**, **cos**, **sinh**, **cosh**.

Las funciones **trigreduce**, **ratsimp** y **radcan** pueden 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 quasi-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**.

```
(%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);
(%o2)      sin(a) sin(3 (- b - a + %pi/3))
----- 
                           sin(b + a)
```

```
(%i3) ba : bc, c=a, a=c;
(%o3)

$$\frac{\sin(3a) \sin(b + a - \frac{\pi}{3})}{\sin(a - \frac{\pi}{3})}$$

(%i4) ac2 : ba^2 + bc^2 - 2*bc*ba*cos(b);
(%o4)

$$\frac{\sin^2(a) \sin^2(b + a - \frac{\pi}{3}) - (2 \sin(a) \sin(3a) \sin(3(-b - a + \frac{\pi}{3})) \cos(b) \sin(b + a - \frac{\pi}{3}) / (\sin(a - \frac{\pi}{3}) \sin(b + a)) + \sin^2(a) \sin^2(3(-b - a + \frac{\pi}{3}))}{\sin^2(b + a)}$$

(%i5) trigrat (ac2);
(%o5)

$$-\sqrt{3} \sin(4b + 4a) - \cos(4b + 4a) - 2\sqrt{3} \sin(4b + 2a) + 2\cos(4b + 2a) - 2\sqrt{3} \sin(2b + 4a) + 2\cos(2b + 4a) + 4\sqrt{3} \sin(2b + 2a) - 8\cos(2b + 2a) - 4\cos(2b - 2a) + \sqrt{3} \sin(4b) - \cos(4b) - 2\sqrt{3} \sin(2b) + 10\cos(2b) + \sqrt{3} \sin(4a) - \cos(4a) - 2\sqrt{3} \sin(2a) + 10\cos(2a) - 9/4$$

```


16 Funciones Especiales

16.1 Introducción a las funciones especiales

A continuación se especifican las notaciones correspondientes a las funciones especiales:

<code>bessel_j (index, expr)</code>	Función de Bessel de primera especie
<code>bessel_y (index, expr)</code>	Función de Bessel de segunda especie
<code>bessel_i (index, expr)</code>	Función de Bessel modificada de primera especie
<code>bessel_k (index, expr)</code>	Función de Bessel modificada de segunda especie
<code>%he[n] (z)</code>	Polinomio de Hermite (Ojo: he, no h. Ver A&S 22.5.18)
<code>%p[u,v] (z)</code>	Función de Legendre de primera especie
<code>%q[u,v] (z)</code>	Función de Legendre de segunda especie
<code>hstruve[n] (z)</code>	Función H de Struve
<code>lstruve[n] (z)</code>	Función L de Struve
<code>%f[p,q] ([], [], expr)</code>	Función hipergeométrica generalizada
<code>gamma()</code>	Función Gamma
<code>gammagreek(a,z)</code>	Función Gamma incompleta
<code>gammaincomplete(a,z)</code>	Extremo de la función Gamma incompleta
<code>slommel</code>	
<code>%m[u,k] (z)</code>	Función de Whittaker de primera especie
<code>%w[u,k] (z)</code>	Función de Whittaker de segunda especie
<code>erfc (z)</code>	Complemento de la función de error, erf
<code>ei (z)</code>	Integral exponencial (?)
<code>kelliptic (z)</code>	Integral elíptica completa de primera especie (K)
<code>%d [n] (z)</code>	Función de cilindro parabólico

16.2 Funciones de Bessel

bessel_j (v, z) Función de Bessel de primera especie de orden v y argumento z .
La función `bessel_j` se define como

$$\sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{z}{2}\right)^{v+2k}}{k! \Gamma(v+k+1)}$$

aunque la serie infinita no se utiliza en los cálculos.

bessel_y (v, z) 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

$$\sum_{k=0}^{\infty} \frac{1}{k! \Gamma(v+k+1)} \left(\frac{z}{2}\right)^{v+2k}$$

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)**hankel_2** (*v, z*)

Función

Función de Hankel de segunda especie de orden *v* y argumento *z*. (A&S 9.1.4)**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  

          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 $\text{scaled}_bessel_i(v, z) = \exp(-\text{abs}(z)) * bessel_i(v, z)$. 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 $\exp(-\text{abs}(z)) * bessel_i(v, z)$.

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.

16.3 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 $\text{diff}(y(x), x, 2) - x y(x) = 0$.

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 $\text{diff}(\text{airy_ai}(x), x)$ es `airy_dai(x)`.

Véanse `airy_bi`, `airy_dai` y `airy_db`.

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 $\text{diff}(y(x), x, 2) - x y(x) = 0$.

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 $\text{diff}(\text{airy}_bi(x), x)$ es `airy_dbi(x)`.

Véanse `airy_ai` y `airy_db`.

airy_db (x) Función

Es la derivada de la función Bi de Airy, `airy_bi(x)`.

Véanse `airy_ai` y `airy_bi`.

16.4 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*.

gamma (x)

Función

La definición básica de la función gamma (A&S 6.1.1) es

$$\text{gamma}(z) = \int_0^{\infty} t^{z-1} e^{-t} dt$$

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]);
(%o2) [sqrt(%pi), sqrt(%pi)/2, sqrt(%pi)/4, sqrt(%pi)/8]

(%i3) map('gamma,[2/3,5/3,7/3]);
(%o3) [gamma(-2/3), 2*gamma(-2/3)/3, 4*gamma(-1/3)/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:

```
(%i8) declare(z,complex)$
(%i9) conjugate(gamma(z));
(%o9) gamma(conjugate(z))
```

Maxima expande $\text{gamma}(z+n)$ y $\text{gamma}(z-n)$ si gamma_expand vale true:

```
(%i10) gamma_expand:true$
```

$$(\%i11) [\text{gamma}(z+1), \text{gamma}(z-1), \text{gamma}(z+2)/\text{gamma}(z+1)];$$

$$(\%o11) \frac{\text{gamma}(z)}{z \text{gamma}(z), \frac{\text{gamma}(z+1)}{z-1}}$$

Derivada de gamma :

```
(%i12) diff(gamma(z),z);
(%o12) \frac{\psi(z) \text{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 (a,z) Función
Función gamma incompleta superior, A&S 6.5.2: $\text{gamma_incomplete}(a,x) = \int_{-\infty}^x \exp(-t) * t^{(a-1)} dt$

gamma_incomplete_regularized (a,z) Función
Función gamma incompleta superior regularizada, A&S 6.5.1.
 $\text{gamma_incomplete_regularized}(a,z) = \text{gamma_incomplete}(a,z) / \text{gamma}(a)$

gamma_incomplete_generalized (a,z1,z2) Función
Función gamma incompleta generalizada.
 $\text{gamma_incomplete_generalized}(a,z) = \int_{z1}^z t^{(a-1)} \exp(-t) dt$

gammalim Variable optativa
Valor por defecto: 1000000
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 $b \leq n$, 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`:

$$-\log_{\text{gamma}}(b + a) + \log_{\text{gamma}}(b) + \log_{\text{gamma}}(a)$$

%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:

$$\begin{aligned} (\%i1) & [\text{beta}(2, 3), \text{beta}(2, 1/3), \text{beta}(2, a)]; \\ (\%o1) & \left[\frac{1}{12}, \frac{9}{4}, \frac{1}{a(a+1)} \right] \end{aligned}$$

Simplificación para argumentos racionales que suman un entero:

$$\begin{aligned} (\%i2) & [\text{beta}(1/2, 5/2), \text{beta}(1/3, 2/3), \text{beta}(1/4, 3/4)]; \\ (\%o2) & \left[\frac{3\pi}{8}, \frac{2\pi}{\sqrt{3}}, \frac{\sqrt{2}\pi}{\sqrt{3}} \right] \end{aligned}$$

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:

$$\begin{aligned} (\%i3) & \text{beta_args_sum_to_integer:true\$} \\ (\%i4) & \text{beta}(a+1, -a+2); \\ (\%o4) & \frac{\pi(a-1)a}{2\sin(\pi(2-a))} \end{aligned}$$

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

$$\frac{z}{\int_0^1 (1-t)^{b-1} t^{a-1} dt}$$

$$\int_0^z$$

Esta definición es posible para $\operatorname{realpart}(a) > 0$, $\operatorname{realpart}(b) > 0$ y $\operatorname{abs}(z) < 1$. 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 functions.wolfram.com para una completa definición de la función beta incompleta.)

Para enteros negativos $a = -n$ y enteros positivos $b = m$ con $m \leq n$ la función beta incompleta se define como

$$\int_0^z \frac{(1 - m)_k}{k!} z^k \frac{(-n)_k}{(n - k)_k} dz$$

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 $\operatorname{realpart}(a) > 0$, `beta_incomplete` se anula. Para $z=1$ y $\operatorname{realpart}(b) > 0$, `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);
(%o1) 
$$\frac{1 - (1 - z)^b (b z + 1)}{b (b + 1)}$$

```

Simplificación para b entero positivo:

```
(%i2) beta_incomplete(a,2,z);
(%o2) 
$$\frac{(a (1 - z) + 1) z^a}{a}$$

```

$$a(a + 1)$$

Simplificación para a y b enteros positivos:

$$\begin{aligned} (\%i3) \quad & \text{beta_incomplete}(3, 2, z); \\ (\%o3) \quad & \frac{(3(1 - z) + 1)z^3}{12} \end{aligned}$$

Para a entero negativo y $b \leq (-a)$:

$$\begin{aligned} (\%i4) \quad & \text{beta_incomplete}(-3, 1, z); \\ (\%o4) \quad & -\frac{1}{3z^3} \end{aligned}$$

Simplificación para los valores $z = 0$ y $z = 1$:

$$\begin{aligned} (\%i5) \quad & \text{assume}(a > 0, b > 0); \\ (\%i6) \quad & \text{beta_incomplete}(a, b, 0); \\ (\%o6) \quad & 0 \\ (\%i7) \quad & \text{beta_incomplete}(a, b, 1); \\ (\%o7) \quad & \text{beta}(a, b) \end{aligned}$$

Evaluación numérica, tanto con float (precisión doble) como big float (precisión arbitraria):

$$\begin{aligned} (\%i8) \quad & \text{beta_incomplete}(0.25, 0.50, 0.9); \\ (\%o8) \quad & 4.594959440269333 \\ (\%i9) \quad & \text{fpprec}:25; \\ (\%i10) \quad & \text{beta_incomplete}(0.25, 0.50, 0.9b0); \\ (\%o10) \quad & 4.594959440269324086971203b0 \end{aligned}$$

Para $\text{abs}(z) > 1$, beta_incomplete devuelve un resultado complejo:

$$\begin{aligned} (\%i11) \quad & \text{beta_incomplete}(0.25, 0.50, 1.7); \\ (\%o11) \quad & 5.244115108584249 - 1.45518047787844 \text{ } \%i \end{aligned}$$

Resultados para argumentos complejos más generales:

$$\begin{aligned} (\%i14) \quad & \text{beta_incomplete}(0.25+\%i, 1.0+\%i, 1.7+\%i); \\ (\%o14) \quad & 2.726960675662536 - .3831175704269199 \text{ } \%i \\ (\%i15) \quad & \text{beta_incomplete}(1/2, 5/4*\%i, 2.8+\%i); \\ (\%o15) \quad & 13.04649635168716 \text{ } \%i - 5.802067956270001 \\ (\%i16) \quad & \end{aligned}$$

Expansión cuando beta_expand vale true:

$$\begin{aligned} (\%i23) \quad & \text{beta_incomplete}(a+1, b, z), \text{beta_expand:true}; \\ (\%o23) \quad & \frac{a \text{ beta_incomplete}(a, b, z) (1 - z)^b z^a}{b + a} \\ (\%i24) \quad & \text{beta_incomplete}(a-1, b, z), \text{beta_expand:true}; \\ (\%o24) \quad & \frac{\text{beta_incomplete}(a, b, z) (-b - a + 1) (1 - z)^{b-a-1} z^a}{b + a} \end{aligned}$$

$$1 - a \quad 1 - a$$

Derivada e integral de `beta_incomplete`:

```
(%i34) diff(beta_incomplete(a,b,z),z);
          b - 1   a - 1
          (1 - z)      z
(%o34)
(%i35) integrate(beta_incomplete(a,b,z),z);
(%o35)   beta_incomplete(a, b, z) z - beta_incomplete(a + 1, b, z)
(%i36) 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(a,b,z)/beta(a,b)`.

Al igual que `beta_incomplete`, esta definición no es completa. Véase [functions.wolfram.com](http://functions.wolfram.com/Bessel-and-Struve-Functions/BetaIncompleteRegularized/0001/) para una definición completa de `beta_incomplete_regularized`. `beta_incomplete_regularized` se simplifica para a o b entero positivo.

Para $z = 0$ y $\operatorname{realpart}(a) > 0$, `beta_incomplete_regularized` se anula. Para $z=1$ y $\operatorname{realpart}(b) > 0$, `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
          1 - (1 - z)  (b z + 1)
(%o1)

(%i2) beta_incomplete_regularized(a,2,z);
          a
          (a (1 - z) + 1) z
(%o2)

(%i3) beta_incomplete_regularized(3,2,z);
          3
          (3 (1 - z) + 1) z
(%o3)
```

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):

```
(%i17) beta_incomplete_regularized(0.12,0.43,0.9);
(%o7)                               .9114011367359802
(%i18) fpprec:32$ 
(%i19) 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);
(%o13)      beta_incomplete_regularized(a, b, z) - -----
                                b   a
                         (1 - z)   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);
(%o15)      -----
                                b - 1   a - 1
                         (1 - z)   z
                                -----
                                beta(a, b)
(%i16) integrate(beta_incomplete_regularized(a,b,z),z);
(%o16) beta_incomplete_regularized(a, b, z) z
                                a beta_incomplete_regularized(a + 1, b, z)
                                -----
                                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

$$\frac{z2}{\int [\frac{b - 1}{(1 - t)} t^{a - 1} dt] / z1}$$

Maxima simplifica `beta_incomplete_regularized` para a y b enteros positivos.

Para $\text{realpart}(a) > 0$ y $z1 = 0$ o $z2 = 0$, Maxima reduce **beta_incomplete_generalized** a **beta_incomplete**. Para $\text{realpart}(b) > 0$ 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 z_1 e integrales respecto de z_1 y z_2 :

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 bffpsi del paquete bffac puede calcular valores numéricicos.	
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 (<i>expr</i>)	Función
Transforma las funciones binomial , gamma y beta que aparecen en <i>expr</i> 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);
(%o1) 
$$\frac{15 \sqrt{\pi}}{8}$$

```

```
(%i2) numfactor (%);
(%o2) 
$$\frac{15}{8}$$

```

16.5 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 (n,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 .	

16.6 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 $\text{erf}(z)$ (A&S 7.1.1)	
Véase también erfflag .	

erfc (z)	Función
Complemento de la función de error $\text{erfc}(z)$ (A&S 7.1.2)	
$\text{erfc}(z) = 1 - \text{erf}(z)$	

erfi (z)	Función
Función de error imaginaria.	
$\text{erfi}(z) = -\frac{i}{\sqrt{\pi}} \text{erf}(\frac{iz}{\sqrt{\pi}})$	

erf_generalized (z1,z2)	Función
Función de error generalizada $\text{Erf}(z1, z2)$	

fresnel_c (z)	Función
Integral de Fresnel $C(z) = \int_0^z \cos(\frac{\pi}{2}t^2) dt$. (A&S 7.3.1)	
La simplificación $\text{fresnel}_c(-x) = -\text{fresnel}_c(x)$ se aplica cuando la variable global trigsign vale true .	
La simplificación $\text{fresnel}_c(\frac{iz}{\sqrt{\pi}}) = \frac{i}{\sqrt{\pi}} \text{fresnel}_c(x)$ se aplica cuando la variable global %iargs vale true .	
Véanse también erf_representation y hypergeometric_representation .	

fresnel_s (z)	Función
Integral de Fresnel $S(z) = \int_0^z \sin(\frac{\pi}{2}t^2) dt$. (A&S 7.3.2)	
La simplificación $\text{fresnel}_s(-x) = -\text{fresnel}_s(x)$ se aplica cuando la variable global trigsign vale true .	
La simplificación $\text{fresnel}_s(\frac{iz}{\sqrt{\pi}}) = \frac{i}{\sqrt{\pi}} \text{fresnel}_s(x)$ 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.	

16.7 Funciones de Struve

Las funciones de Struve se definen en el capítulo 12 de Abramowitz y Stegun, *Handbook of Mathematical Functions*.

Maxima tiene un soporte limitado sobre estas funciones, que pueden aparecer en resultados devueltos por `hgfred`.

hstruve [v] (z)	Función
Función de Struve H de orden v y argumento z. (A&S 12.1.1)	

lstruve [v] (z)	Función
Función de Struve modificada L de orden v y argumento z. (A&S 12.2.1)	

16.8 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 $pFq(a_1, a_2, \dots, a_p; b_1, b_2, \dots, b_q; z)$, donde a es una lista de longitud p y b otra lista de longitud q.	

16.9 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`.

%d [v] (z)	Función
Es la función de cilindro parabólico $\%d[v](z)$. (A&S 19.3.1)	

16.10 Funciones y variables para las funciones especiales

specint ($\exp(-s*t) * \text{expr}$, 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$
(%i8) radcan(specint((cos(t)*expintegral_si(t)
          -sin(t)*expintegral_ci(t))*%e^(-s*t),t));
          log(s)
```

```
(%o8) 
$$\frac{2}{s^2 + 1}$$

(%i9) ratsimp(specint((2*t*log(a)+2/a*sin(a*t)
-2*t*expintegral_ci(a*t))*%e^(-s*t),t));
(%o9) 
$$\frac{\log(s^2 + a^2)}{s^2}$$

(%i10) assume(s>0)$
(%i11) specint(1/sqrt(%pi*t)*unit_step(t-k)*%e^(-s*t),t);
(%o11) 
$$\frac{\gamma_{incomplete}(-, k s)}{\sqrt{\pi} \sqrt{s}}$$

```

Resultados cuando se utiliza la expansión de `gamma_incomplete` y se cambia la representación de `expintegral_e1`:

```
(%i12) gamma_expand:true$
(%i13) specint(1/sqrt(%pi*t)*unit_step(t-k)*%e^(-s*t),t);
(%o13) 
$$\frac{\operatorname{erfc}(\sqrt{k} \sqrt{s})}{\sqrt{s}}$$

(%i14) expintrep:expintegral_e1$
```

$$(%i15) ratsimp(specint(1/(t+a)^2 * %e^(-s*t), t));
(%o15) - \frac{a s \%e^{\gamma_{incomplete}(a s) - 1}}{a}$$
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 $\text{pf}[p,q]([a], [b], x)$, 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);

(%o2) 
$$\frac{4^{\frac{v}{2}} \operatorname{bessel\_j}(v, z) \gamma(v + 1) \%e^{\frac{\%i z}{2}}}{\gamma(v + 1)}$$

```

```

          v
          z
(%i3) hgfred([1,1],[2],z);

(%o3)      - log(1 - z)
                  -----
                  z

(%i4) hgfred([a,a+1/2],[3/2],z^2);

(%o4)      (1 - 2 a)           1 - 2 a
      (z + 1)           - (1 - z)
-----           -----
                  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
-----           -----
          a (a + 1)

(%i7) ev(%);

          2
          z           2 z
----- - --- + 1
          a (a + 1)     a

```

lambert_w (z)

Función

Rama principal de la función W de Lambert, la solución de la ecuación $z = W(z) * \exp(W(z))$.

nzeta (z)

Función

Función de dispersión del plasma. $\text{nzeta}(z) = \%i * \sqrt{\%pi} * \exp(-z^2) * (1 - \text{erf}(-\%i * z))$

nzetar (z)

Función

Devuelve `realpart(nzeta(z))`.

nzetai (z)

Función

Devuelve `imagpart(nzeta(z))`.

17 Funciones elípticas

17.1 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 parámetro m en lugar del módulo k o del ángulo α . 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 relacionadas 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);
(%o3)      phi   %pi
           log(tan(--- + ---))
           2       4
(%i4) elliptic_e (phi, 1);
(%o4)                                sin(phi)
(%i5) elliptic_e (phi, 0);
(%o5)                                phi
(%i6) elliptic_kc (1/2);
(%o6)      1
           elliptic_kc(-)
           2
(%i7) makegamma (%);
(%o7)      2 1
           gamma (-)
           4
-----_
           4 sqrt(%pi)
(%i8) diff (elliptic_f (phi, m), phi);
(%o8)      1
-----_
           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).

17.2 Funciones y variables para funciones elípticas

jacobi_sn (<i>u, m</i>)	Función
Función elíptica jacobiana $sn(u, m)$.	
jacobi_cn (<i>u, m</i>)	Función
Función elíptica jacobiana $cn(u, m)$.	
jacobi_dn (<i>u, m</i>)	Función
Función elíptica jacobiana $dn(u, m)$.	

jacobi_ns (u, m)	Función elíptica jacobiana $ns(u, m) = 1/sn(u, m)$.	Función
jacobi_sc (u, m)	Función elíptica jacobiana $sc(u, m) = sn(u, m)/cn(u, m)$.	Función
jacobi_sd (u, m)	Función elíptica jacobiana $sd(u, m) = sn(u, m)/dn(u, m)$.	Función
jacobi_nc (u, m)	Función elíptica jacobiana $nc(u, m) = 1/cn(u, m)$.	Función
jacobi_cs (u, m)	Función elíptica jacobiana $cs(u, m) = cn(u, m)/sn(u, m)$.	Función
jacobi_cd (u, m)	Función elíptica jacobiana $cd(u, m) = cn(u, m)/dn(u, m)$.	Función
jacobi_nd (u, m)	Función elíptica jacobiana $nc(u, m) = 1/cn(u, m)$.	Función
jacobi.ds (u, m)	Función elíptica jacobiana $ds(u, m) = dn(u, m)/sn(u, m)$.	Función
jacobi_dc (u, m)	Función elíptica jacobiana $dc(u, m) = dn(u, m)/cn(u, m)$.	Función
inverse_jacobi_sn (u, m)	Inversa de la función elíptica jacobiana $sn(u, m)$.	Función
inverse_jacobi_cn (u, m)	Inversa de la función elíptica jacobiana $cn(u, m)$.	Función
inverse_jacobi_dn (u, m)	Inversa de la función elíptica jacobiana $dn(u, m)$.	Función
inverse_jacobi_ns (u, m)	Inversa de la función elíptica jacobiana $ns(u, m)$.	Función
inverse_jacobi_sc (u, m)	Inversa de la función elíptica jacobiana $sc(u, m)$.	Función
inverse_jacobi_sd (u, m)	Inversa de la función elíptica jacobiana $sd(u, m)$.	Función

inverse_jacobi_nc (<i>u, m</i>)	Función
Inversa de la función elíptica jacobiana $nc(u, m)$.	
inverse_jacobi_cs (<i>u, m</i>)	Función
Inversa de la función elíptica jacobiana $cs(u, m)$.	
inverse_jacobi_cd (<i>u, m</i>)	Función
Inversa de la función elíptica jacobiana $cd(u, m)$.	
inverse_jacobi_nd (<i>u, m</i>)	Función
Inversa de la función elíptica jacobiana $nc(u, m)$.	
inverse_jacobi_ds (<i>u, m</i>)	Función
Inversa de la función elíptica jacobiana $ds(u, m)$.	
inverse_jacobi_dc (<i>u, m</i>)	Función
Inversa de la función elíptica jacobiana $dc(u, m)$.	

17.3 Funciones y variables para integrales elípticas

elliptic_f (<i>phi, m</i>)	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 (<i>phi, m</i>)	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 (<i>u, m</i>)	Función
Integral elíptica incompleta de segunda especie, definida como	

$$\int_0^u dn(v, m)^2 dv = \int_0^\tau \sqrt{\frac{1 - mt^2}{1 - t^2}} dt$$

donde $\tau = sn(u, m)$.

Esto se relaciona con **elliptic_e** mediante

$$E(u, m) = E(\phi, m)$$

donde $\phi = \sin^{-1} sn(u, m)$.

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.

18 Límites

18.1 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).

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.

19 Diferenciación

19.1 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  
          y(x) %e      (-- (z(x)))  
                  dx  
(%o2)      (%i3) a1: antid (expr, x, z(x));  
          z(x)      z(x) d  
          [y(x) %e      , - %e      (-- (y(x)))]  
                  dx  
(%o3)      (%i4) a2: antidiff (expr, x, z(x));  
          z(x)      /  
          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  
          [0, y(x) %e      (-- (z(x)))]  
                  dx  
(%o6)      (%i7) antidiff (expr, x, y(x));  
          z(x)      /  
          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 realizar 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  
          y(x) %e      (-- (z(x)))  
                  dx  
(%o2)  
(%i3) a1: antid (expr, x, z(x));  
          z(x) z(x) d  
          [y(x) %e , - %e      (-- (y(x)))]  
                  dx  
(%o3)  
(%i4) a2: antidiff (expr, x, z(x));  
          /  
          z(x) [ z(x) d  
          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  
          [0, y(x) %e      (-- (z(x)))]  
                  dx  
(%o6)  
(%i7) antidiff (expr, x, y(x));  
          /  
          [ z(x) d  
          I y(x) %e      (-- (z(x))) dx  
          ] dx  
          /
```

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 $f(x_1, \dots, x_m)$, o una derivada, `diff(f(x_1, \dots, x_m), x_1, n_1, \dots, 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 $[x_1 = a_1, \dots, x_m = a_m]$. Si hay una única variable x_1 , la ecuación puede escribirse sin formar parte de una lista.

La llamada `printprops ([f1, f2, ...], atvalue)` muestra los valores asignados por `atvalue` a las funciones f_1, f_2, \dots . 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:

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 \wedge (producto exterior) y \lrcorner (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 returnedo es el llamado "diferencial total".

Ejemplos:

```
(%i1) diff (log (x));
(%o1)                               del(x)
                                         -----
                                         x
(%i2) diff (exp (x*y));
(%o2)      x y           x y           x y
             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;
(%o1)           - a s
                     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_i , x_i , 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 elementos del arreglo 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);
(%o5) 
$$\frac{dr}{du} \cdot s + r \cdot \frac{ds}{du}$$

(%i6) diff (r.s, t);
(%o6) 
$$\frac{dr}{dt} \frac{du}{dt} \cdot s + r \cdot \frac{ds}{dt} \frac{du}{dt}$$

(%i7) remove (r, dependency);
(%o7) done
(%i8) diff (r.s, t);
(%o8) 
$$r \cdot \frac{ds}{dt} \frac{du}{dt}$$

```

derivabbrev

Valor por defecto: `false`

Variable optativa

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
      --- + --- + 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 variable 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);  

(%o2)                                grad (z^2 + y^2 + x^2)  

(%i3) express (%);  

(%o3) [--(z^2 + y^2 + x^2), --(z^2 + y^2 + x^2), --(z^2 + y^2 + x^2)]  

      dx                      dy                      dz  

(%i4) ev (% , diff);  

(%o4) [2 x, 2 y, 2 z]  

(%i5) div ([x^2, y^2, z^2]);  

(%o5)                                div [x^2, y^2, z^2]  

(%i6) express (%);  

(%o6) --(z^2) + --(y^2) + --(x^2)  

      dz          dy          dx  

(%i7) ev (% , diff);  

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

(%i8) curl ([x^2, y^2, z^2]);  

(%o8)                                curl [x^2, y^2, z^2]  

(%i9) express (%);  

(%o9) [--(z^2) - --(y^2), --(x^2) - --(z^2), --(y^2) - --(x^2)]  

      dy          dz          dx          dz          dx          dy  

(%i10) ev (% , diff);  

(%o10) [0, 0, 0]  

(%i11) laplacian (x^2 * y^2 * z^2);  

(%o11)                                laplacian (x^2 y^2 z^2)  

(%i12) express (%);  

(%o12) --- (x^2 y^2 z^2) + --- (x^2 y^2 z^2) + --- (x^2 y^2 z^2)  

      dz          dy          dx  

(%i13) ev (% , diff);  

(%o13) 2 y^2 z^2 + 2 x^2 z^2 + 2 x^2 y^2  

(%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 ($f(x_1, \dots, x_n), g_1, \dots, g_m$)
gradef (a, x, expr)

Función
Función

Define las derivadas parciales, o componentes del gradiente, de la función f o variable a .

La llamada **gradef** ($f(x_1, \dots, x_n), g_1, \dots, g_m$) 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, \dots, 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 $f(x_1, \dots, x_n)$ o a no se evalúa, pero sí lo hacen el resto de argumentos g_1, \dots, 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** ($[f_1, \dots, f_n], \text{gradef}$) muestra las derivadas parciales de las funciones f_1, \dots, f_n , tal como las definió **gradef**.

La llamada **printprops** ($[a_n, \dots, a_n], \text{atomgrad}$) muestra las derivadas parciales de las variables a_n, \dots, 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

Variante 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
```

$$\begin{aligned} & \frac{-(\pi s)^2}{(s + i)(1 - e^{\pi s})^2(s - i)(1 - e^{-\pi s})} \\ (%o9) & \text{factor}(\%); \\ & \frac{\pi^2 s^2}{(s - i)(s + i)(e^{\pi s} - 1)} \end{aligned}$$

20 Integración

20.1 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)`.

20.2 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 (%i2, 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 $i = j + \dots$, 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
          ====
          \
          >      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
          /
          [
          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
```

$$\begin{aligned}
 (%o4) & - \frac{3}{a^3 b^2 - 2 a^2 b + a^2} + \frac{2}{a b - 1} \\
 & + \frac{2}{a^3 b^2 - 2 a^2 b + a^2}
 \end{aligned}$$

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 indefinida 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 presente, `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)      -----
                  3
                  - cos(x)

(%i2) integrate (x/ sqrt (b^2 - x^2), x);
                  2      2
                  - sqrt(b  - x )
(%o2)
(%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;
(%o2)                                3
                                         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);
(%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));

```

```
(%o3)      a   1
           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
-- = x + %c3
            3

(%i4) reset (integration_constant_counter);
(%o4)           [integration_constant_counter]
(%i5) integrate (x^2 = 1, x);
            3
            x
-- = 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  

      ] 3   2                                2  

      / x  - x  + 1      log(x  + x + 1)      5 atan(-----)  

      ----- - ----- + -----  

      (%o2) 7                                     14                               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(x - x + 1)  

(%o4) -----  

7  


$$\frac{\log(x^3 + x^2 + 1)}{7} \sqrt{3} \operatorname{atan}\left(\frac{2x + 1}{\sqrt{3}}\right)$$

```

$$\frac{14}{7 \sqrt{3}}$$

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
          %pi x   erf(x) + (sqrt(%pi) x   + sqrt(%pi)) %e
(%o1)      -----
                  - x
```

```
(%i2) diff(%, x), ratsimp;
          3 %pi
          2
(%o2)           x   erf(x)
```

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

Función

Equivale a **ldefint** cuando **tlimswitch** vale **true**.

20.3 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. Piessens¹, E. de Doncker², C. Ueberhuber³, and D. Kahaner⁴.

La librería QUADPACK incluida en Maxima es una traducción automática (mediante el programa **f2c1**) del código fuente Fortran de QUADPACK tal como se encuentra en la SLATEC Common Mathematical Library, Versión 4.1⁵. La librería SLATEC está fechada en julio de 1993, pero las funciones QUADPACK fueron escritas algunos años antes. Hay otra versión de QUADPACK en Netlib⁶, 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 libro⁷.

20.3.1 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).

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. Überhuber, and D.K. Kahaner. *QUADPACK: A Subroutine Package for Automatic Integration*. Berlin: Springer-Verlag, 1983, ISBN 0387125531.

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 $\cos(\omega x)f(x)$ o $\sin(\omega x)f(x)$ en un intervalo finito, siendo ω 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**.

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 $(x-a)^{\alpha}(b-x)^{\beta}etav(x)$, 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 $x = c$.

20.4 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
Función

quad_qags (*f*, *x*, *a*, *b*, [*epsrel*, *epsabs*, *limit*])

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
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 argumentosopcionales 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
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 $w(x) = \cos(\omega gax)$

sin $w(x) = \sin(\omega gax)$

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 argumentosopcionales 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 , ω , trig , [epsrel , epsabs , limit , maxp1 ,
 limlst])

Función

quad_qawo (*f*, *x*, *a*, *b*, *omega*, *trig*, [*epsrel*, *epsabs*, *limit*, *maxp1*, *limlst*])

Función

Integración de $\cos(\omega x)f(x)$ o $\sin(\omega x)f(x)$ en un intervalo finito, siendo ω 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 :

$$\cos w(x) = \cos(\omega q x)$$

$$\sin w(x) = \sin(\omega q ax)$$

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 argumentosopcionalespuedenespecificarseenquierorden.Todosellos toman la forma `key=val`. Tales argumentos son:

<code>epsrel</code>	El error absoluto deseado para la aproximación. El valor por defecto es 1d-8.
<code>epsabs</code>	Error absoluto deseado de la aproximación. El valor por defecto es 0.
<code>limit</code>	Tamaño del arreglo interno de trabajo. <code>limit/2</code> es el número máximo de subintervalos para la partición. El valor por defecto es 200.
<code>maxp1</code>	Número máximo de momentos de Chebyshev. Debe ser mayor que 0. El valor por defecto es 100.
<code>limlst</code>	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 , α , β , w , wf , [epsrel , epsabs , limit])

Función

quad_qaws (f , x , a , b , α , β , w , wf , [epsrel , epsabs , limit])

Función

Integración de $w(x)f(x)$ en un intervalo finito $[a, b]$, siendo w una función de la forma $(x - a)^{\alpha}(b - x)^{\beta}v(x)$, 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 utilizando 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;
(%o2)      alpha      alpha
           2 %pi 2      sqrt(2 2      + 1)
-----alpha
           4 2      + 2
(%i3) ev (% , alpha=4, numer);
(%o3)      8.750097361672829
```

21 Ecuaciones

21.1 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 función `algsys`. Las variables `%r` se añaden a `%rnum_list` en su orden de creación. Esto es útil para hacer sustituciones en la solución.

```
(%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 t2 - 3 t1 + 4, y = t2, z = t1]]
```

algexact

Variable opcional

Default value: `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 biquadrática.

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 sola incógnita y es lineal, o cuadrática o biquadrática, 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 biquadrática, y si la variable **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);
(%o3)                                a1 (- y - x^2 + 1)
(%i4) e4: a2*(y - (x - 1)^2);
(%o4)                                a2 (y - (x - 1)^2)
(%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)
[ 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)


$$(x + 1.015755543828121) (x^2 + .8139194463848151 x$$


$$+ 1.098699797110288)$$


```

bfallroots (*expr*)
bfallroots (*eqn*)

Función
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);

(%t3)
      sqrt(23)   25 1/3
      (----- + --)
      6 sqrt(3)   54

Solution:

(%t4)
      sqrt(3) %i   1
      -----
      sqrt(3) %i   1   2   2   1
      (- ----- - -) %t3 + ----- - -
      2           2           9 %t3   3

(%t5)
      sqrt(3) %i   1
      -----
      sqrt(3) %i   1   2   2   1
      (----- - -) %t3 + ----- - -
      2           2           9 %t3   3

(%t6)
      1   1
      x = %t3 + ----- - -
      9 %t3   3
(%o6)
      [%t4, %t5, %t6]
(%i6) breakup: false$
(%i7) solve (x^3 + x^2 - 1);
Solution:

(%t7)
      sqrt(3) %i   1
      -----
      2   2   sqrt(23)   25 1/3
      ----- + (----- + --)
      sqrt(23)   25 1/3   6 sqrt(3)   54
      9 (----- + --)
      6 sqrt(3)   54

      sqrt(3) %i   1   1
      (- ----- - -) - -
      2           2           3

(%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
                                         1          1
(%t9) x = (----- + --) + ----- sqrt(23)   25 1/3   3
                                         6 sqrt(3)   54
                                         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 $[g(t) = \dots]$ 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  
  

(%t2)                               17  

      x : --  

           7  
  

(%t3)                               1  

      y : - -  

           7  

(%o3) [[%t2, %t3]]  

(%i3) x;  

(%o3)                               17  

      --  

       7  

(%i4) y;  

(%o4)                               1  

      - -  

       7  

(%i5) globalsolve: false$  

(%i6) kill (x, y)$  

(%i7) solve ([x + 3*y = 2, 2*x - y = 5], [x, y]);  

      Solution  
  

(%t7)                               17  

      x = --  

       7  
  

(%t8)                               1  

      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 returnedo 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)
(%i2) lhs (e);
(%o2)
(%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 triangularización 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;
(%o2) 2 a x - y = 2 a2
(%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`.

programmode Variable opcional

Valor por defecto: `true`

Si `programmode` vale `true`, `solve`, `realroots`, `allroots` y `linsolve` devuelven sus soluciones como elementos de una lista.

Si `programmode` vale `false`, `solve` y las demás crean expresiones intermedias etiquetadas `%t1`, `t2`, etc., y les asignan 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:
```

```
(%t3)      sqrt(3) %i + 1  
x = - -----  
          2  
  
(%t4)      sqrt(3) %i - 1  
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 (<i>expr, bound</i>)	Función
realroots (<i>eqn, bound</i>)	Función
realroots (<i>expr</i>)	Función
realroots (<i>eqn</i>)	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 (<i>expr</i>)	Función
Devuelve el miembro derecho (es decir, el segundo argumento) de la expresión <i>expr</i> , cuando el operador de <i>expr</i> es uno de los operadores de relación < <= # equal notequal >= > , o un operadores de asignación := ::= : :: , o un operador infix 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 <code>rootsconmode</code>	Resultadod de <code>rootscontract</code>
$x^{(1/2)}*y^{(3/2)}$	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  

          sqrt(x y )  

(%i3) rootscontract (x^(1/2)*y^(1/4));  

          1/4  

          sqrt(x) y  

(%i4) rootsconmode: true$  

(%i5) rootscontract (x^(1/2)*y^(1/4));  

          sqrt(x sqrt(y))  

(%i6) rootscontract (x^(1/2)*y^(1/3));  

          1/3  

          sqrt(x) y  

(%i7) rootsconmode: all$  

(%i8) rootscontract (x^(1/2)*y^(1/4));  

          2   1/4  

          (x y)  

(%i9) rootscontract (x^(1/2)*y^(1/3));  

          3   2 1/6  

          (x y )  

(%i10) rootsconmode: false$  

(%i11) rootscontract (sqrt(sqrt(x) + sqrt(1 + x))  

                      *sqrt(sqrt(1 + x) - sqrt(x)));  

          1  

(%i12) rootsconmode: true$  

(%i13) rootscontract (sqrt(5 + sqrt(5)) - 5^(1/4)*sqrt(1 + sqrt(5)));■  

          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 la 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
          sqrt(3) %i   1   sqrt(4 a + 27)   1 1/3
      (- ----- - -) (----- - -)
          2           2       6 sqrt(3)           2
          sqrt(3) %i   1
          (- ----- - -) a
          2           2
      - -----, x =
          3
          sqrt(4 a + 27)   1 1/3
      3 (----- - -)
          6 sqrt(3)           2
```

```

            3
      sqrt(4 a  + 27)   1 1/3           a
      (----- - -) - -----]
      6 sqrt(3)          2                   3
                                         sqrt(4 a  + 27)   1 1/3
                                         3 (----- - -)
                                         6 sqrt(3)          2

(%i6) solve (x^3 - 1);
      sqrt(3) %i - 1           sqrt(3) %i + 1
(%o6)  [x = -----, x = -----, x = 1]
           2                      2

(%i7) solve (x^6 - 1);
      sqrt(3) %i + 1           sqrt(3) %i - 1
(%o7)  [x = -----, x = -----, x = - 1,
           2                      2
                                         sqrt(3) %i + 1           sqrt(3) %i - 1
                                         x = - -----, x = - -----, x = 1]
                                         2                      2

(%i8) ev (x^6 - 1, %[1]);
      6
      (sqrt(3) %i + 1)
(%o8)  ----- - 1
           64

(%i9) expand (%);
(%o9) 0
(%i10) x^2 - 1;
      2
(%o10) x  - 1
(%i11) solve (%), x);
(%o11) [x = - 1, x = 1]
(%i12) ev (%th(2), %[1]);
(%o12) 0

```

solvedecomposes

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.

solvenglwarn	Variable opcional
Valor por defecto: true	
Si solvenglwarn 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 exponentiales 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.	

22 Ecuaciones Diferenciales

22.1 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**.

22.2 Funciones y variables para ecuaciones diferenciales.

bc2 (*solut, xval1, yval1, xval2, yval2*)

Función

Resuelve el problema del valor en la frontera para ecuaciones diferenciales de segundo orden. Aquí, *solut* 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 $x = x_0$, 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* ($i=1,\dots,n$) 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
          -- (f(x)) = -- (g(x)) + sin(x)
          dx          dx
(%o1)
(%i2) 'diff(g(x),x,2)='diff(f(x),x)-cos(x);
          2
          d           d
          --- (g(x)) = --- (f(x)) - cos(x)
          2           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)]);
(%o5) [f(x) = a %ex - a + 1, g(x) =
                                              x
                                         cos(x) + a %ex - a + g(0) - 1]
(%i6) [%o1,%o2],%o5,diff;
(%o6) [a %ex = a %ex, a %ex - cos(x) = a %ex - 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 $x = x_0$ para la variable independiente y *yval* es una ecuación de la forma $y = y_0$ 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 $x = x_0$ para la variable independiente y *yval* es una ecuación de la forma $y = y_0$ 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;
(%o1)          2 dy           sin(x)
              x -- + 3 x y = -----
                           dx           x
(%i2) ode2(% ,y,x);
(%o2)          %c - cos(x)
              y = -----
                           3
                           x
(%i3) ic1(%o2,x=%pi,y=0);
(%o3)          cos(x) + 1
              y = - -----
                           3
                           x
(%i4) 'diff(y,x,2) + y*'diff(y,x)^3 = 0;
(%o4)          2
              d y      dy 3
              --- + y (--) = 0
              2           dx
(%i5) ode2(% ,y,x);
(%o5)          3
              y + 6 %k1 y
              ----- = x + %k2
              6
(%i6) ratsimp(ic2(%o5,x=0,y=0,'diff(y,x)=2));
(%o6)          3
              2 y - 3 y
              - ----- = x
              6
(%i7) bc2(%o5,x=0,y=1,x=1,y=3);
(%o7)          3
              y - 10 y      3
              ----- = x - -
              6                  2
```


23 Métodos numéricos

23.1 Introducción a los métodos numéricos

23.2 Series de Fourier

El paquete **fft** contiene funciones para el cálculo numérico (no simbólico) de la transformada rápida de Fourier. La instrucción **load ("fft")** carga el paquete. Véase **fft**.

El paquete **fourie** contiene funciones para el cálculo simbólico de series de Fourier. La instrucción **load ("fourie")** carga el paquete. Hay funciones en el paquete **fourie** para calcular los coeficientes de Fourier y para la transformación de expresiones. Véase **Funciones y variables para las series de Fourier**.

23.3 Funciones y variables para los métodos numéricos

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 tamaños 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

$$\begin{aligned} a &= r \cos(t) \\ b &= r \sin(t) \end{aligned}$$

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 b la imaginaria. Ambos valores a y b son arrays unidimensionales cuyos tamaños 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

$$\begin{aligned} r &= \sqrt{a^2 + b^2} \\ t &= \text{atan2}(b, a) \end{aligned}$$

El ángulo calculado pertenece 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

$$x[j] = \text{sum}(y[k] \exp(+2\pi i k j / n), k, 0, n - 1)$$

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

$$y[k] = (1/n) \sum (x[j] \exp(-2\pi i j k / n)), \quad j, 0, n - 1$$

Si los datos x son reales, los coeficientes reales a y b se pueden calcular de manera que

$$x[j] = \sum (a[k] * \cos(2\pi j k / n) + b[k] * \sin(2\pi j k / n)), \quad k, 0, n/2 - 1$$

con

$$\begin{aligned} a[0] &= \text{realpart}(y[0]) \\ b[0] &= 0 \end{aligned}$$

y, para k entre 1 y $n/2 - 1$,

$$\begin{aligned} a[k] &= \text{realpart}(y[k] + y[n - k]) \\ b[k] &= \text{imagpart}(y[n - k] - y[k]) \end{aligned}$$

y

$$\begin{aligned} a[n/2] &= \text{realpart}(y[n/2]) \\ b[n/2] &= 0 \end{aligned}$$

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)
(%i18) makelist (float (f (j)), j, 0, n - 1);
(%o18) [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]
```

fortindent

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 exponentiación \wedge 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 reconocida 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.

`horner (expr, x)`
`horner (expr)`

Función
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 estabilidades 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)	Función
find_root (f, a, b)	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 $\text{lhs}(\text{expr}) - \text{rhs}(\text{expr})$.

Dado que Maxima puede evaluar *expr* o *f* en $[a, b]$, entonces, si *expr* o *f* es continua, **find_root** encontrará la raíz buscada; o raíces en caso de existir varias.

La función **find_root** aplica al principio la búsqueda binaria. Si la expresión es lo suficientemente suave, entonces **find_root** aplicará el método de interpolación lineal.

La precisión del resultado se controla con **find_root_abs** y **find_root_rel**. La búsqueda se detiene cuando la función alcanza valores menores o iguales a **find_root_abs**, o cuando dos soluciones sucesivas x_0 y x_1 se diferencian en una cantidad menor o igual a **find_root_rel * 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 esto no se verifica, 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;
(%o1)           f(x) := sin(x) - -
          2
(%i2) find_root (sin(x) - x/2, x, 0.1, %pi);
```

```
(%o2) 1.895494267033981
(%i3) find_root (sin(x) = x/2, x, 0.1, %pi);
(%o3) 1.895494267033981
(%i4) find_root (f(x), x, 0.1, %pi);
(%o4) 1.895494267033981
(%i5) find_root (f, 0.1, %pi);
(%o5) 1.895494267033981
(%i6) find_root (exp(x) = y, x, 0, 100);
          x
(%o6) find_root(%e = y, x, 0.0, 100.0)
(%i7) find_root (exp(x) = y, x, 0, 100), y = 10;
(%o7) 2.302585092994046
(%i8) log (10.0);
(%o8) 2.302585092994046
```

newton (*expr*, *x*, *x_0*, *eps*)

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
```

23.4 Funciones y variables para las 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 (<i>f, expr</i>)	Función
remfun (<i>f, expr, x</i>)	Función
La llamada remfun (<i>f, expr</i>) reemplaza todas las subexpresiones <i>f</i> (<i>arg</i>) por <i>arg</i> en <i>expr</i> .	
La llamada remfun (<i>f, expr, x</i>) reemplaza todas las subexpresiones <i>f</i> (<i>arg</i>) por <i>arg</i> en <i>expr</i> sólo si <i>arg</i> contiene a la variable <i>x</i> .	
funp (<i>f, expr</i>)	Función
funp (<i>f, expr, x</i>)	Función
La llamada funp (<i>f, expr</i>) devuelve true si <i>expr</i> contiene la función <i>f</i> .	
La llamada funp (<i>f, expr, x</i>) devuelve true si <i>expr</i> contiene la función <i>f</i> y la variable <i>x</i> está presente en el argumento de alguna de las presencias de <i>f</i> .	
absint (<i>f, x, halfplane</i>)	Función
absint (<i>f, x</i>)	Función
absint (<i>f, x, a, b</i>)	Función
La llamada absint (<i>f, x, halfplane</i>) devuelve la integral indefinida de <i>f</i> con respecto a <i>x</i> en el semiplano dado (pos , neg o both). La función <i>f</i> puede contener expresiones de la forma abs (<i>x</i>), abs (sin (<i>x</i>)), abs (<i>a</i>) * exp (- abs (<i>b</i>) * abs (<i>x</i>)).	
La llamada absint (<i>f, x</i>) equivale a absint (<i>f, x, pos</i>).	
La llamada absint (<i>f, x, a, b</i>) devuelve la integral definida de <i>f</i> con respecto a <i>x</i> de <i>a</i> a <i>b</i> .	
fourier (<i>f, x, p</i>)	Función
Devuelve una lista con los coeficientes de Fourier de <i>f</i> (<i>x</i>) definida en el intervalo [- <i>p</i> , <i>p</i>].	
foursimp (<i>l</i>)	Función
Simplifica sin (<i>n %pi</i>) a 0 si sinnpiflag vale true y cos (<i>n %pi</i>) 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 (<i>l, x, p, limit</i>)	Función
Calcula y devuelve la serie de Fourier a partir de la lista de los coeficientes de Fourier <i>l</i> hasta el término <i>limit</i> (<i>limit</i> puede ser inf). Los argumentos <i>x</i> y <i>p</i> tienen el mismo significado que en fourier .	
fourcos (<i>f, x, p</i>)	Función
Devuelve los coeficientes de los cosenos de Fourier de <i>f</i> (<i>x</i>) definida en [0, <i>p</i>].	

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 <code>fourexpand (foursimp (fourier (f, x, p)), x, p, 'inf)</code> .	
fourint (f, x)	Función
Calcula y y devuelve la lista de los coeficientes integrales de Fourier de $f(x)$ definida en $[\text{minf}, \text{inf}]$.	
fourintcos (f, x)	Función
Devuelve los coeficientes integrales de los cosenos $f(x)$ en $[0, \text{inf}]$.	
fourintsin (f, x)	Función
Devuelve los coeficientes integrales de los senos $f(x)$ en $[0, \text{inf}]$.	

24 Arrays

24.1 Funciones y variables para Arrays

array (<i>nombre, dim_1, ..., dim_n</i>)	Función
array (<i>nombre, type, dim_1, ..., dim_n</i>)	Función
array ([<i>nombre_1, ..., nombre_m</i>], <i>dim_1, ..., dim_n</i>)	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 uno declarado.

arrayapply (<i>A, [i_1, ..., i_n]</i>)	Función
Evalúa <i>A</i> [<i>i_1, ..., i_n</i>], donde <i>A</i> es un array y <i>i_1, ..., i_n</i> son enteros.	

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

arrayinfo (<i>A</i>)	Función
Devuelve información sobre el array <i>A</i> . El argumento <i>A</i> 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)
(%i2) aa [2, 3] : %pi;
(%o2)
(%i3) aa [1, 2] : %e;
(%o3)
(%i4) arrayinfo (aa);
(%o4)
(%i5) listarray (aa);
(%o5) [#####, #####, #####, #####, #####, #####, %e, #####,
       #####, #####, #####, %pi]
```

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

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

arrayinfo y **listarray** aplicadas a una función array.

```
(%i1) cc [x, y] := y / x;
(%o1)
(%i2) cc [u, v];
(%o2)
(%i3) cc [4, z];
(%o3)
(%i4) arrayinfo (cc);
(%o4)
(%i5) listarray (cc);
(%o5)
```

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

```
(%i1) dd [x] (y) := y ^ x;
(%o1)                               dd (y) := y
                                         x
(%i2) dd [a + b];
(%o2)                               dd (y) := lambda([y], y
                                         b + a)
(%i3) dd [v - u];
(%o3)                               dd (y) := lambda([y], y
                                         v - u)
(%i4) arrayinfo (dd);
(%o4)                               [hashed, 1, [b + a], [v - u]]
(%i5) listarray (dd);
(%o5)      [lambda([y], y
                                         b + a), lambda([y], y
                                         v - u)]
```

arraymake (*name*, [*i₁*, ..., *i_n*])

Función

El resultado es una referencia a array no evaluada.

Devuelve la expresión *name* [*i₁*, ..., *i_n*].

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

Ejemplos:

```
(%i1) arraymake (A, [1]);
(%o1)                               A
                                         1
(%i2) arraymake (A, [k]);
(%o2)                               A
                                         k
(%i3) arraymake (A, [i, j, 3]);
(%o3)                               A
                                         i, j, 3
(%i4) array (A, fixnum, 10);
(%o4)                               A
(%i5) fillarray (A, makelist (i^2, i, 1, 11));
(%o5)                               A
(%i6) arraymake (A, [5]);
(%o6)                               A
                                         5
(%i7) '';
(%o7)                               36
(%i8) L : [a, b, c, d, e];
(%o8)                               [a, b, c, d, e]
(%i9) arraymake ('L, [n]);
(%o9)                               L
                                         n
(%i10) '';
(%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
(%o14)          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;
(%o2)          (b + a)^2
(%i3) cc [x] := x/100;
(%o3)          cc := --- x
(%i4) dd : make_array ('any, 7);
(%o4)          {Array: #(NIL NIL NIL NIL NIL NIL NIL)}
(%i5) arrays;
(%o5)          [aa, bb, cc]
```

bashindices (expr)

Función

Transforma la expresión *expr* dándole a cada suma y producto un único índice. Esto le da a **changevar** mayor precisión cuando opera con sumas y productos. La forma del único índice es *jnumber*. La cantidad *number* se determina en función de **gensumnum**, valor que puede cambiar el usuario. Por ejemplo, haciendo **gensumnum:0\$**.

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 llenan 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)
(%i2) aa [2, 3] : %pi;
(%o2)
(%i3) aa [1, 2] : %e;
(%o3)
(%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;
(%o1)
(%i2) bb [BAR] : (c - d)^3;
(%o2)
(%i3) listarray (bb);
(%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;
(%o1)
(%i2) cc [u, v];
(%o2)
(%i3) cc [4, z];
(%o3)
(%i4) listarray (cc);
(%o4) [-, -]
(%i5) arrayinfo (cc);
(%o5) [hashed, 2, [4, z], [u, v]]
```

listarray y arrayinfo aplicadas a una función subindicada.

```
(%i1) dd [x] (y) := y ^ x;
(%o1)
(%i2) dd [a + b];
```

```

(%o2)                                b + a
                                         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 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)}
(%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)}
(%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 llenado con los elementos del viejo según el orden de las filas. Si el array antiguo era demasiado pequeño, los elementos restantes se llenan 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.

subvar (*x, i*)

Función

Evaluá 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
```

use_fast_arrays

Variable opcional

Valor por defecto: `false`

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

25 Matrices y Álgebra Lineal

25.1 Introducción a las matrices y el álgebra lineal

25.1.1 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`.

25.1.2 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`.

Aviso: el paquete `vect` declara el operador `.` como conmutativo.

25.1.3 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`.

25.2 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 ($[eqn_1, \dots, eqn_m]$, $[x_1, \dots, x_n]$) Función
Devuelve la matriz aumentada de coeficientes del sistema de ecuaciones lineales eqn_1, \dots, eqn_m de variables x_1, \dots, 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, \dots, x_n .

```
(%i1) m: [2*x - (a - 1)*y = 5*b, c + b*y + a*x = 0]$
(%i2) augcoefmatrix (m, [x, y]);
          [ 2   1   - a   - 5   b ]
          [                           ]
(%o2)          [ a     b       c   ]
```

charpoly (M , x) 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
          lambda - 7 lambda + 10
(%i3) (programmode: true, solve (%));
(%o3)          [lambda = 5, lambda = 2]
(%i4) matrix ([x1], [x2]);
          [ x1 ]
          [     ]
          [ x2 ]
(%o4)          [ x2 - 2 x1 ]
```

```
(%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;
(%o7) x2^2 + x1^2 = 1
(%i8) solve (%th(2), %), [x1, x2]);
(%o8) [[x1 = - ----, x2 = - -----], ]
              1                           2
              sqrt(5)                     sqrt(5)

[x1 = -----, x2 = -----]
              1                           2
              sqrt(5)                     sqrt(5)
```

coefmatrix ([eqn_1, ..., eqn_m], [x_1, ..., x_n])

Función

Devuelve la matriz de coeficientes para las variables x_1, \dots, x_n del sistema de ecuaciones lineales eqn_1, \dots, eqn_m .

```
(%i1) coefmatrix([2*x-(a-1)*y+5*b = 0, b*y+a*x = 3], [x,y]);
(%o1) [ 2   1 - a ]
          [      ]
          [ 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 ]
```

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)
```

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 ]
              [      ]
              [ c  d ]
(%o1)

(%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 $base^{\text{exponente}}$ 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) ]
```

$$(%o5) \begin{bmatrix} & & \\ & b + a & \%pi \\ [(1 - c) & (1 - c) \end{bmatrix}$$
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`.

dotscreules Variable opcional

Valor por defecto: `false`

Cuando `dotscreules` 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 vectores 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);
[ [[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 leídos 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 (<i>a</i> , <i>i</i> _2, <i>j</i> _2, <i>i</i> _1, <i>j</i> _1)	Función
genmatrix (<i>a</i> , <i>i</i> _2, <i>j</i> _2, <i>i</i> _1)	Función
genmatrix (<i>a</i> , <i>i</i> _2, <i>j</i> _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);
(%o1)          h      := -----
                           1
                           i, j   i + j - 1
(%i2) genmatrix (h, 3, 3);
(%o2)
[ 1   1   1 ]
[ 1   -   - ]
[ 2   3   ]
[       ]
[ 1   1   1 ]
[ -   -   - ]
[ 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);
(%o6)
[ %e   0   ]
[           ]
[ 0   %pi   ]
(%i7) genmatrix (lambda ([i, j], j - i), 3, 3);
(%o7)
[ 0   1   2   ]
[           ]
[ - 1   0   1   ]
[           ]
[ - 2   - 1   0   ]
(%i8) genmatrix (B, 2, 2);
(%o8)
[ B     B     ]
[ 1, 1   1, 2 ]
```

```
(%o8)      [          ]
              [ B       B   ]
              [ 2, 1   2, 2 ]
```

gramschmidt (*x*)
gramschmidt (*x*, *F*)

Función
 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 (<i>n</i>)	Función
Devuelve la matriz identidad de orden <i>n</i> .	
innerproduct (<i>x, y</i>)	Función
inprod (<i>x, y</i>)	Función
Devuelve el producto interior o escalar de <i>x</i> por <i>y</i> , que deben ser listas de igual longitud, o ambas matrices columna o fila de igual longitud. El valor devuelto es conjugate (<i>x</i>) . <i>y</i> , donde . es el operador de multiplicación no commutativa.	
Es necesario cargar la función haciendo load ("eigen") .	
El nombre inprod es sinónimo de innerproduct .	
invert (<i>M</i>)	Función
Devuelve la inversa de la matriz <i>M</i> , calculada por el método del adjunto.	
La implementación actual no es eficiente para matrices de orden grande.	
Cuando detout vale true , el determinante queda fuera de la inversa a modo de factor escalar.	
Los elementos de la matriz inversa no se expanden. Si <i>M</i> tiene elementos polinómicos, se puede mejorar el aspecto del resultado haciendo expand (invert (m)) , detout .	
Véase la descripción de ^^ (exponente no commutativo) para información sobre otro método para invertir matrices.	
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 (<i>fila_1, ..., fila_n</i>)	Función
Devuelve una matriz rectangular con las filas <i>fila_1, ..., fila_n</i> . 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 escalar y una matriz o una matriz y un escalar, pero no si los operandos son dos matrices.	
El producto matricial se representa con el operador de multiplicación no commutativa .. El correspondiente operador de exponenciación no commutativa es ^^ . Dada la matriz <i>A</i> , <i>A.A = A^^2</i> y <i>A^^-1</i> es la inversa de <i>A</i> , si existe.	

Algunas variables controlan la simplificación de expresiones que incluyan estas operaciones: `doallmxops`, `domxexpt`, `dommxops`, `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 ]
      [           ]
      [ - 8   11 ]
(%o1)
(%i2) y: matrix ([%pi, %e], [a, b]);
      [ %pi   %e ]
      [           ]
      [ a     b ]
(%o2)
```

- Suma elemento a elemento.

```
(%i3) x + y;
      [ %pi + 17   %e + 3 ]
      [                   ]
      [ a - 8       b + 11 ]
(%o3)
```

- Resta elemento a elemento.

```
(%i4) x - y;
      [ 17 - %pi   3 - %e ]
      [                   ]
      [ - a - 8     11 - b ]
(%o4)
```

- Multiplicación elemento a elemento.

```
(%i5) x * y;
      [ 17 %pi   3 %e ]
      [                   ]
      [ - 8 a     11 b ]
(%o5)
```

- División elemento a elemento.

```
(%i6) x / y;
      [ 17          - 1 ]
      [ --- 3 %e      ]
      [ %pi          ]
      [               ]
      [   8    11      ]
      [ --      --     ]
      [   a    b      ]
(%o6)
```

- Matriz elevada a un exponente escalar, operación elemento a elemento.

```
(%i7) x ^ 3;
      [ 4913     27   ]
      [               ]
      [ - 512   1331 ]
(%o7)
```

- Base escalar y exponente matricial, operación elemento a elemento.

(%i8) `exp(y);`

```
(%o8)
[ %pi    %e ]
[ %e    %e ]
[      ]
[   a    b ]
[ %e    %e ]
```

- Base y exponente matriciales. Esta operación no se realiza elemento a elemento.

(%i9) `x ^ y;`

```
(%o9)
[ %pi  %e ]
[      ]
[   a   b ]
[ 17   3 ]
[      ]
[ - 8  11 ]
```

- Multiplicación matricial no commutativa.

(%i10) `x . y;`

```
(%o10)
[      ]
[      ]
[ 3 a + 17 %pi  3 b + 17 %e ]
[ 11 a - 8 %pi  11 b - 8 %e ]
```

(%i11) `y . x;`

```
(%o11)
[      ]
[      ]
[ 17 %pi - 8 %e  3 %pi + 11 %e ]
[ 17 a - 8 b     11 b + 3 a ]
```

- Exponenciación matricial no commutativa. 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;`

```
(%o12)
[      ]
[      ]
[ 3833  1719 ]
[ - 4584  395 ]
```

(%i13) `%e ^^ y;`

```
(%o13)
[      ]
[      ]
[ %pi    %e ]
[ %e    %e ]
[      ]
[   a    b ]
[ %e    %e ]
```

- Una matriz elevada al exponente -1 con el operador de exponenciación no commutativa equivale a la matriz inversa, si existe.

(%i14) `x ^^ -1;`

```
(%o14)
[ 11      3 ]
[ ---  - --- ]
[ 211      211 ]
[      ]
[ 8      17 ]
[ ---  --- ]
[ 211      211 ]
```

```
(%i15) x . (x ^~ -1);
(%o15)
[ 1  0 ]
[      ]
[ 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**.**matrixxp** (*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]);
(%o3)
[ a   b   c ]
[           ]
[ d   e   f ]
(%i4) bb: matrix ([u, v, w], [x, y, z]);
(%o4)
[ u   v   w ]
[           ]
[ x   y   z ]
(%i5) aa . transpose(bb);
(%o5)
[ u   v   w   x   y   z ]
[ a   b   c   a   b   c ]
[           ]
[ 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           2 ]  

          [ sqrt((c - w) + (b - v) + (a - u)) ]  

(%o6) Col 1 = [ ]  

          [           2           2           2           2 ]  

          [ sqrt((f - w) + (e - v) + (d - u)) ]  

          [           2           2           2           2 ]  

          [ sqrt((c - z) + (b - y) + (a - x)) ]  

Col 2 = [ ]  

          [           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*.

ncexpt (*a, b*)

Función

Si una expresión exponencial no comutativa es demasiado grande para mostrarse en la forma *a*^{*b*} entonces aparece como **ncexpt** (*a, b*).

El nombre **ncexpt** no corresponde al de una función u operador, sino que tan solo aparece en la salida y no se reconoce como una entrada válida.

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 *x * ident [n] - a*. La función **ncharpoly** es más eficiente en el caso de matrices grandes y densas.

Para disponer de esta función es necesario cargar el paquete haciendo `load ("nchrpl")`.

newdet (M, n)	Función
Calcula el determinante de la matriz o arreglo M por el algoritmo del árbol menor de Johnson-Gentleman. El argumento n es el orden; es opcional si M es una matriz.	
nonscalar	Declaración
Hace que los átomos se comporten como hace una lista o matriz con respecto del operador . del la multiplicación no commutativa.	
nonscalarp ($expr$)	Función
Devuelve true si $expr$ no es escalar, es decir, si contiene átomos declarados como no escalares, listas o matrices.	
permanent (M, n)	Función
Calcula la permanente de la matriz M . La permanente es como un determinante pero sin cambios de signo.	
rank (M)	Función
Calcula el rango de la matriz M . Esto es, el orden del mayor subdeterminante no singular de M .	
La función <i>rango</i> 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.	
scalarmatrixp	Variable opcional
Valor por defecto: true	
Si scalarmatrixp vale true , entonces siempre que una matriz 1×1 se produce como resultado del cálculo del producto no commutativo de matrices se cambia al formato escalar.	
Si scalarmatrixp vale all , entonces todas las matrices 1×1 se simplifican a escalares.	
Si scalarmatrixp vale false , las matrices 1×1 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 **uniteigenvalues**. 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 ($i_1, \dots, i_m, M, j_1, \dots, j_n$) Función

submatrix (i_1, \dots, i_m, M) Función

submatrix (M, j_1, \dots, j_n) Función

Devuelve una nueva matriz formada a partir de la matriz M pero cuyas filas i_1, \dots, i_m y columnas j_1, \dots, 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 `known eigvects` 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 `list eigvects`, en tal caso el contenido de `list eigvects` debe ser una lista de estructura similar a la que devuelve la función `eigenvectors`.

Si `known eigvects` vale `true` y la lista de vectores propios está en la variable `list eigvects`, 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 función 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 función es necesario cargarla haciendo `load ("eigen")`.

El nombre `uvect` es sinónimo de `unitvector`.

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 $p \cdot (q \cdot r)$ en $(p \cdot r) * q - (p \cdot q) * r$.

`expandcurlcurl`

Simplifica $\text{curl}(\text{curl}(p))$ en $\text{grad}(\text{div}(p)) + \text{div}(\text{grad}(p))$.

`expandlaplaciantodivgrad`

Simplifica $\text{laplacian}(p)$ en $\text{div}(\text{grad}(p))$.

`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);  
(%o8)          1  
             k : = -----  
                  2  
                  k + 1  
(%i9) g[10];  
(%o9)          1  
             ---  
            101
```


26 Afines

26.1 Funciones y variables para Afines

fast_linsolve ([expr_1, ..., expr_m], [x_1, ..., x_n])

Función

Resuelve las ecuaciones lineales simultáneas $expr_1, \dots, expr_m$ para las variables x_1, \dots, 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 $[x_1 = a_1, \dots, x_n = a_n]$ donde todas las a_1, \dots, a_n están exentas de x_1, \dots, 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 $expr_1, \dots, expr_m$. 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 $expr_1, \dots, expr_m$, es decir, si y sólo si f es una combinación polinómica de los elementos de $expr_1, \dots, expr_m$.

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, \dots, w_n a x_1, \dots, 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, \dots, 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 álgebra 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 comutativos p_1, \dots, p_n de los monomios no comutativos m_1, \dots, 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 comutativos que aparecen en el polinomio p o una lista de polinomios en p_1, \dots, 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.

27 itensor

27.1 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.

27.1.1 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
                                                g   g   a
                                                i j
(%t3)                                         k l
                                                a
(%i4) ishow(contract(%))$
```

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
                                                g   a   g
                                                i j
(%t5)                                         l k
                                                a
(%i6) ishow(contract(%))$
```

El único código que hace uso de esta notación es la función **lc2kdt**.

Devido a que este código es nuevo, puede contener errores.

27.1.2 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))$      v   - v
(%t9)                               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
(%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])))$      s   - s
(%t17)           s   - s
           i j   j i
(%i18) decsym(s,2,0,[sym(all)],[]);      done
(%o18)
(%i19) ishow(canform(s([i,j],[])-s([j,i])))$      0
(%t19)
(%i20) ishow(canform(a([i,j],[])+a([j,i])))$      a   + a
(%t20)

```

```
j i      i j
(%i21) decsym(a,2,0,[anti(all)],[]);          done
(%o21)
(%i22) ishow(canform(a([i,j],[])+a([j,i])))$ 0
(%t22)
```

27.2 Funciones y variables para itensor

27.2.1 Trabajando con objetos indexados

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 *nuevo* 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 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
                                d e c      + a      b
                                x y      i j    u,v
(%t6)
(%i7) ishow(listoftens(%))$
                                k
                                [a      , b      , c      , d]
                                i j      u,v    x y
(%t7)
```

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  

          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*)
rename (*expr, count*)

Función
Función

Devuelve una expresión equivalente a *expr* pero con los índices mudos de cada término elegidos del conjunto `[%, %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([%,%4],[%,%3])*
          ichr2([%,%3],[u])*ichr2([%,%6],[%,%1])*ichr2([%,%7,r],[%,%2])-  

          g([],[%4,%5])*g([],[%6,%7])*ichr2([%,%1],[u])*  

          ichr2([%,%5],[%,%1])*ichr2([%,%6],[%,%3])*ichr2([%,%7,r],[%,%2]),noeval$  

(%i4) expr:ishow(%)$
```

$$\begin{aligned} & \frac{\partial}{\partial x^4} \frac{\partial}{\partial x^5} \frac{\partial}{\partial x^6} \frac{\partial}{\partial x^7} \frac{\partial}{\partial u} \frac{\partial}{\partial x^1} \frac{\partial}{\partial x^2} \\ & \left(g(x^4, x^5) g(x^6, x^7) \right) \left(\frac{\partial}{\partial x^1} \frac{\partial}{\partial x^4} \frac{\partial}{\partial x^2} \frac{\partial}{\partial x^3} \right) \left(\frac{\partial}{\partial x^5} \frac{\partial}{\partial x^6} \frac{\partial}{\partial x^7} \right) \left(\frac{\partial}{\partial u} \frac{\partial}{\partial x^1} \frac{\partial}{\partial x^2} \right) \end{aligned}$$

$$\begin{aligned} & - g(x^4, x^5) g(x^6, x^7) \left(\frac{\partial}{\partial x^1} \frac{\partial}{\partial x^2} \frac{\partial}{\partial x^3} \frac{\partial}{\partial x^5} \right) \left(\frac{\partial}{\partial x^4} \frac{\partial}{\partial x^6} \frac{\partial}{\partial x^7} \right) \left(\frac{\partial}{\partial u} \frac{\partial}{\partial x^1} \frac{\partial}{\partial x^2} \right) \end{aligned}$$

```

(%i5) flipflag:true;
(%o5)                               true
(%i6) ishow(rename(expr))$          %2 %5 %6 %7      %4      u      %1      %3
(%t6) g      g      ichr2      ichr2      ichr2      ichr2
      %1 %2      %3 %4      %5 %6      %7 r

      %4 %5 %6 %7      u      %1      %3      %2
- g      g      ichr2      ichr2      ichr2      ichr2
      %1 %2      %3 %4      %5 %6      %7 r

(%i7) flipflag:false;
(%o7)                               false
(%i8) rename(%th(2));
(%o8)                               0
(%i9) ishow(rename(expr))$          %1 %2 %3 %4      %5      %6      %7      u
(%t9) g      g      ichr2      ichr2      ichr2      ichr2
      %1 %6      %2 %3      %4 r      %5 %7

      %1 %2 %3 %4      %6      %5      %7      u
- g      g      ichr2      ichr2      ichr2      ichr2
      %1 %3      %2 %6      %4 r      %5 %7

```

flipflag

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 segúan 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*)
defcon (*tensor_1, tensor_2, tensor_3*)

Función

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]))$  

(%t3) i j  

e p
```

2) Como una matriz:

```
(%i5) lg:-ident(4)$lg[1,1]:1$lg;  

      [ 1   0   0   0 ]  

      [                 ]  

      [ 0   - 1   0   0 ]  

      [                 ]  

      [ 0   0   - 1   0 ]  

(%o5)
```

```

[      ]
[ 0    0    0   - 1 ]
(%i6) components(g([i,j],[]),lg);
(%o6)                                done
(%i7) ishow(g([i,j],[]))$                                g
(%t7)                               i j
(%i8) g([1,1],[]);                                 1
(%o8)
(%i9) g([4,4],[]);                                - 1
(%o9)

```

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]))$  

(%t5) 
$$k\delta_{ij}$$

(%i6) ishow(h([i,j],[k],1))$  

(%t6) 
$$g^{ijk}$$

```

4) Utilizando los patrones de Maxima, en particular las funciones `defrule` y `applyb1`:

%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 ]
               [ ]
               [ 0 0 0 sqrt(-----) ]
               [ 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 ]
               [ ]
               [ 0 0 0 sqrt(-----) ]
               [ 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 conflicto 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, lo que no implica que `kdelta([i,j],[])` sea un objeto tensorial de pleno derecho.

kdels (L_1, L_2) 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]))$
```

$$(\%t4) \quad \begin{matrix} & c & & d & & d & & c \\ & \text{kdelta} & \text{kdelta} & - & \text{kdelta} & \text{kdelta} & \\ & a & & b & & a & & b \end{matrix}$$

```
(%i4) ishow(kdels([a,b],[c,d]))$
```

$$(\%t4) \quad \begin{matrix} & c & & d & & d & & c \\ & \text{kdelta} & \text{kdelta} & + & \text{kdelta} & \text{kdelta} & \\ & a & & b & & a & & b \end{matrix}$$

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
          1   1
```

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
      a      - g      a
                  j
(%t11)

```

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
      a   a  levi_civita
      i   j   k
(%i3) el2:ishow('levi_civita([], [i,j,k])*a([i])*a([j]))$
      i   j   k
      levi_civita   a   a
      i   j
(%i4) canform(contract(expand(applyb1(el1,lc_1,lc_u))));
(%t4)          0
(%i5) canform(contract(expand(applyb1(el2,lc_1,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_1`, puede utilizarse para simplificar muchas expresiones de forma más eficiente que la evaluación de `levi_civita`. Véase `lc_1`.

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 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**.

concaten (*expr*) Función

Similar a **canten** pero también realiza la contracción de los índices.

27.2.2 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],[]),
                           *kdeps([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) 
$$\frac{3 \text{ a}}{i \text{ k} \text{ j}} + \frac{3 \text{ a}}{i \text{ j} \text{ k}}$$

(%i12) dispssym(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.

27.2.3 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([],[],1)))$  

          k   %2   %2   %2  

(%t2) b (v a + v a + v a )  

          ,1   i j,%2   ,j   i %2   ,i   %2 j  

          %1   k   %1   k   %1   k  

          + (v b - b v + v b ) a  

          ,%1 1   ,1   ,%1   ,1   ,%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))$  

(%t3)          icurvature  

                  i j k,m  

(%i4) ishow(evundiff(%))$  

(%t4) - ichr2      l      l      %1      l      %1  

      i k,j m      - ichr2      ichr2      - ichr2      ichr2  

                  %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))$  

(%t6) -----  

      g      (g      - g      + g      )  

      j %3,i l    i j,%3 l    i %3,j l  

-----  

      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  

          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:

$$\begin{aligned} \text{ab} & \quad \text{ab} & \quad \text{ab} & \quad \text{a} \\ g_{,d} g_{bc} + g_{bc,d} & = (g_{bc} g_{,d}) & = (\text{kdelta}_{bc,d}) & = 0 \\ & & & c_{,d} \end{aligned}$$

de donde

$$g_{,d} g_{bc} = - g_{bc} g_{,d}$$

y

$$g_{,j} g_{ab,i} = g_{ab,j} g_{,i}$$

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
          g     g     g
          a b,d  b c,e

(%i6) flipflag:not flipflag;
(%o6)                               true
(%i7) ishow(simpmetderiv(%th(2)))$
          a b   b c
          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
          - g     g     g
          ,e   a b,d  b c

(%i10) ishow(contract(%))$
          b c
          - 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.

27.2.4 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

$$(g_{ik,j} + g_{jk,i} - g_{ij,k})/2 .$$

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

$$\text{ichr2}([i,j],[k]) = g_{is,j} \frac{(g_{js,i} + g_{ij,s} - g_{ij,s})}{2}$$

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:

$$\begin{aligned} \text{icurvature}_{i,j,k} &= -\text{ichr2}_{i,k} - \text{ichr2}_{j,k} + \text{ichr2}_{i,j} + \text{ichr2}_{j,i} \\ &\quad + \text{ichr2}_{i,j} + \text{ichr2}_{j,i} \end{aligned}$$

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: [];;
(%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
```

$$\begin{aligned}
 & (\%t5) - \frac{g^2}{k} \left(g_{\%1,j} s_{\%3,i} - g_{i,s,\%3} + g_{i,\%3,s} \right) \\
 & + \frac{g^2}{k} \left(g_{i,j} s_{\%2,\%1} - g_{\%1,s,\%2} + g_{\%1,\%2,s} \right) k \\
 & + \frac{2}{i,j,s}
 \end{aligned}$$

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
          ichr2      - ichr2
          r s,t       r t,s
(%t3)
(%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

```

```

          - ichr2      + ichr2
          r t,s      r s,t
(%i5) canform(%);
(%o5)           0

```

27.2.5 Sistemas de referencia móviles

Maxima puede hacer cálculos utilizando sistemas de referencia móviles, los cuales pueden ser ortonormales o cualesquier 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 `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
          v      + icc2      v
          ,j      %1 j
(%t3)
(%i4) ishow(ev(%,icc2))$ 
          %1      i      i
          v      ifc2      + v
          %1 j      ,j
(%t4)
(%i5) ishow(ev(%,ifc2))$ 
          %1      i  %2      i
          v      ifg      ifc1      + v
          %1 j  %2      ,j
(%t5)

```

```
(%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]))$  

(%t7)      (%3      %4  

      (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:

$$\text{ifc1} = \frac{-\text{ifb}_{c a b} + \text{ifb}_{b c a} + \text{ifb}_{a b c}}{2}$$

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 **itortion_flag** vale **true**:

$$\text{ifb} = \frac{\text{ifr}_d \text{ifr}_e (\text{ifri}_{a d, e} - \text{ifri}_{a e, d} - \text{ifri}_{a f} \text{itr}_f)}{\text{abc}^2 \text{b}^2 \text{c}^2}$$

En otro caso:

$$\text{ifb} = \frac{(\text{ifr}_e \text{ifr}_d - \text{ifr}_{c, e} \text{ifr}_{b, e} - \text{ifr}_c \text{ifr}_{a d}) \text{ifri}_f}{\text{abc}^2 \text{b}^2 \text{c}^2 \text{d}^2}$$

icc1

Variable

Coeficientes de conexión de primera especie. Se definen en **itensor** como

$$\begin{array}{cccc} \text{icc1} & = & \text{ichr1} & - \text{ikt1} & - \text{inmc1} \\ & & \text{abc} & \text{abc} & \text{abc} & \text{abc} \end{array}$$

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

$$\begin{array}{cccc} \text{c} & & \text{c} & & \text{c} & & \text{c} \\ \text{icc2} & = & \text{ichr2} & - & \text{ikt2} & - & \text{inmc2} \\ & & \text{ab} & & \text{ab} & & \text{ab} & \text{ab} \end{array}$$

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 referencia. 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 representa la contribución de la métrica del sistema de referencia al coeficiente de conexión de primera especie, definido como

$$\begin{array}{c} -\text{ifb} & +\text{ifb} & +\text{ifb} \\ \text{c a b} & \text{b c a} & \text{a b c} \\ \text{ifc1} & = & \hline & & \\ \text{abc} & & 2 \end{array}$$

ifc2

Variable

Coeficiente del sistema de referencia de primera especie. Este tensor representa la contribución de la métrica del sistema de referencia al coeficiente de conexión de primera especie, definido como

$$\begin{array}{ccc} \text{c} & & \text{cd} \\ \text{ifc2} & = & \text{ifg} \quad \text{ifc1} \\ \text{ab} & & \text{abd} \end{array}$$

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`.

27.2.6 Torsión y no metricidad

Maxima trabaja con conceptos como la torsión y la no metrictad. 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 metrictad cuando `inonmet_flag` vale `true`.

inm Variable
 Vector de no metrictad. La no metrictad 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

$$g_{ij;k} = g_{inm} \quad i, j, k$$

inmc1 Variable
 Permutación covariante de las componentes del vector de no metrictad. Se define como

$$\text{inmc1} = \frac{g_{ab}^{inm} - g_{ac}^{inm} - g_{bc}^{inm}}{2}$$

(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 metrictad. Se utiliza en los coeficientes de conexión si `inonmet_flag` vale `true`. Se define como

$$\text{inmc2} = \frac{\begin{matrix} & c & c & cd \\ -\text{inm} & \text{kdelta} & -\text{kdelta} & \text{inm} + g & \text{inm} g \\ & a & b & a & b \\ c & & & & d \\ ab & & & & ab \end{matrix}}{2}$$

(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 torsión. Se define como

$$\text{ikt1} = \frac{\begin{matrix} & d & d & d \\ -g & \text{itr} & -g & \text{itr} & -\text{itr} & g \\ ad & cb & bd & ca & ab & cd \end{matrix}}{2}$$

(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 torsión. Se define como

$$\text{ikt2} = \frac{c}{ab} \frac{cd}{abd} \text{ikt1}$$

(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)
(%i3) covdiff(covdiff(f([],[]),i),j)
        -covdiff(covdiff(f([],[]),j),i)$
(%i4) ishow(%)$
(%t4)          f      ichr2      %4      %2
           ,%4      j i      ,%2      i j
(%i5) conform(%);
(%o5)          0
```

```

(%i6) itorsion_flag:true;
(%o6)                               true
(%i7) covdiff(covdiff(f[],[],i),j)
      -covdiff(covdiff(f[],[],j),i)$
(%i8) ishow(%)$
(%t8)          %8           %6
      f   icc2 - f   icc2 - f   + f
      ,%8   j i   ,%6   i j   ,j i   ,i j
(%i9) ishow(canform(%))$
(%t9)          %1           %1
      f   icc2 - f   icc2
      ,%1   j i   ,%1   i j
(%i10) ishow(canform(ev(%,icc2)))$
(%t10)          %1           %1
      f   ikt2 - f   ikt2
      ,%1   i j   ,%1   j i
(%i11) ishow(canform(ev(%,ikt2)))$
(%t11)          %2 %1           %2 %1
      f   g   ikt1 - f   g   ikt1
      ,%2   i j %1   ,%2   j i %1
(%i12) ishow(factor(canform(rename(expand(ev(%,ikt1))))))$
(%t12)          %3 %2           %1           %1
      f   g   g   (itr - itr )
      ,%3   %2 %1   j i   i j
-----^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(%)))$ 
(%t16)          %1
      - f   itr
      ,%1   i j

```

27.2.7 Álgebra exterior

Con el paquete **itensor** se pueden realizar operaciones en campos tensoriales covariantes antisimétricos. Un campo tensorial totalmente antisimétrico 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

$$\begin{matrix} \text{a a} & - & \text{a a} \\ & \text{i j} & \text{j i} \end{matrix}$$

$$a_i \wedge a_j = \frac{1}{2} a_{ij}$$

De forma más general, el producto de una p-forma por una q-forma se definiría como

$$A_{i_1..i_p} \wedge B_{j_1..j_q} = \frac{1}{(p+q)!} D^{k_1..k_p l_1..l_q} A_{i_1..i_p} B_{j_1..j_q k_1..k_p l_1..l_q}$$

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,

$$a_i a_j = a_{ij} - a_{ji}$$

y, en el caso general,

$$A_{i_1..i_p} \wedge B_{j_1..j_q} = \frac{1}{p! q!} D^{k_1..k_p l_1..l_q} A_{i_1..i_p} B_{j_1..j_q k_1..k_p l_1..l_q}$$

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 exterior: si `igeowedge_flag` vale `false` (el valor por defecto), se utiliza la primera definición, si vale `true`, la segunda.

\sim

Operador

El operador del producto exterior se representa por el símbolo \sim . Este es un operador binario. Sus argumentos deben ser expresiones que tengan escalares, tensores covariantes de rango uno o tensores covariantes de rango 1 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]))$
```

$$(t2) \frac{a_i b_j - b_i a_j}{2}$$

```
(%i3) decsym(a,2,0,[anti(all)],[]);
```

$$(t3) \text{done}$$

```
(%i4) ishow(a([i,j])~b([k]))$
```

$$(t4) \frac{a_i b_j + a_j b_i - a_k b_k}{3}$$

```
(%i5) igeowedge_flag:true;
```

$$(t5) \text{true}$$

```
(%i6) ishow(a([i])~b([j]))$
```

$$(t6) a_i b_j - b_i a_j$$

```
(%i7) ishow(a([i,j])^b([k]))$  

(%t7)          i   j      i   j  

           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)$  

(%t3)          v   a  

           %1  

           %1   j  

(%i4) ishow(a([j,i],[])|v)$  

(%t4)          - v   a  

           %1  

           %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))$  

(%t2)          v   - v  

           j,i   i,j  

           -----  

           2  

(%i3) decsym(a,2,0,[anti(all)],[]);  

(%o3)          done  

(%i4) ishow(extdiff(a([i,j]),k))$  

(%t4)          a   - a   + a  

           j   k,i   i   k,j   i   j,k  

           -----  

           3
```

```
(%i5) igeowedge_flag:true;
(%o5)                                true
(%i6) ishow(extdiff(v([i]),j))$
```

$$\frac{v_{j,i} - v_{i,j}}{2}$$

```
(%t6)
(%i7) ishow(extdiff(a([i,j]),k))$
```

$$- \left(a_{k,j,i} - a_{k,i,j} + a_{j,i,k} \right)$$

```
(%t7)
```

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) icrounter:100;
(%o4)                               100
(%i5) decsym(A,3,0,[anti(all)],[],[])$
```

$$A_{i,j,k}$$

```
(%i6) ishow(A([i,j,k],[]))$
```

$$\frac{\text{levi_civita}_{\{1,2,3,4\}} g_{\{1,102}} A_{\{2,3,4\}}}{6}$$

```
(%t6)
(%i7) ishow(canform(hodge(%)))$
```

$$\frac{\text{levi_civita}_{\{1,2,3,8\}} \text{levi_civita}_{\{2,107\}} g_{\{1,106\}}}{6}$$

$$g_{\{1,106\}} g_{\{3,108\}} g_{\{4,8\}} A_{\{5,6,7\}}$$

```
(%t7)
(%i8) ishow(canform(hodge(%)))$
```

$$\frac{\text{levi_civita}_{\{1,2,3,8\}} \text{levi_civita}_{\{2,107\}} g_{\{1,106\}}}{6}$$

$$g_{\{1,106\}} g_{\{3,108\}} g_{\{4,8\}} A_{\{5,6,7\}}$$

```
(%t8)
(%i9) lc2kdt(%)$
```

$$g_{\{1,106\}} g_{\{3,108\}} g_{\{4,8\}} A_{\{5,6,7\}}$$

```
(%i10) %,kdelta$
```

$$g_{\{1,106\}} g_{\{3,108\}} g_{\{4,8\}} A_{\{5,6,7\}}$$

```
(%i11) ishow(canform(contract(expand(%))))$
```

$$- A_{\{106,107,108\}}$$

```
(%t11)
```

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.

27.2.8 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 utilizar 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)
(%i4) ishow(icurvature([j,k,l],[i]))$  

          m
          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.

27.2.9 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]))$
```

$$\begin{matrix} & & k & & m & & l & k \\ & t & = & f & a & b & g \\ & i & j & , & j & i & l & m \end{matrix}$$

```
(%i3) ic_convert(eqn);
(%o3) for i thru dim do (for j thru dim do (
    for k thru dim do
        t : f sum(sum(diff(a , ct_coords ) b
           i, j, k
           m
           j i, l, k
           g , l, 1, dim), m, 1, dim)))
           l, m
(%i4) imetric(g);
(%o4)                               done
(%i5) metricconvert:true;
(%o5)                               true
(%i6) ic_convert(eqn);
(%o6) for i thru dim do (for j thru dim do (
    for k thru dim do
        t : f sum(sum(diff(a , ct_coords ) b
           i, j, k
           m
           j i, l, k
           lg , l, 1, dim), m, 1, dim)))
           l, m
```

27.2.10 Palabras reservadas

Las siguientes palabras son utilizadas por el paquete `itensor` internamente, por lo que no deberían ser modificadas por el usuario:

Palabra	Comentarios
<hr/>	
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
concat	
irpm	
lc0	

_lc2kdt0
_lcprod
_extlc

28 ctensor

28.1 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` 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);

(%t3)      a
            x
mcs      = ---
1, 1, 1   2 a

(%t4)      1
mcs      = -
1, 2, 2   x

(%t5)      1
mcs      = -
1, 3, 3   x

(%t6)      d
            x
mcs      = ---
1, 4, 4   2 d

(%t7)      x
mcs      = - -
2, 2, 1   a

(%t8)      cos(y)
mcs      = -----
2, 3, 3   sin(y)

(%t9)      2
            x sin (y)
mcs      = - -----
3, 3, 1   a

(%t10)     mcs      = - cos(y) sin(y)

```

```

3, 3, 2

d
x
(%t11) mcs      = ---
4, 4, 1   2 a
(%o11)          done

```

28.2 Funciones y variables para ctensor

28.2.1 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_coorsys (sistema_coordenadas, extra_arg)

Función

ct_coorsys (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
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 coordenadas bipolares
parabolic	2 [u,v]	Sistema de coordenadas parabólicas
cartesian3d	3 [x,y,z]	Sistema de coordenadas cartesianas
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
spherical4d	4	[r,theta,eta,phi]	Sistema de coordenadas esféricas
exteriororschwarzschild	4	[t,r,theta,phi]	Métrica de Schwarzschild
interiororschwarzschild	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           ]
                                         [ 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.

28.2.2 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
    \
     tracer - ein -- lein
       \
        riem -- lriem -- weyl

```

```
\uriem
```

El paquete `ctensor` también puede trabajar con sistemas de referencia móviles. Si `cframe_flag` vale `true`, se pueden calcular los siguientes tensores:

```
lfg -- ufg
 \
 fri -- fr -- lcs -- mcs -- lriem -- ric -- uric
   \
     lg -- ug           | \      \      \
                         |   weyl   tracer - ein -- lein
                         | \
                         | riem
                         |
                         \uriem
```

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.

leinsteinv (*dis*)

Función

Es el tensor covariante de Einstein. La función **leinsteinv** 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:

$$R[i,j,k,l] = R^1_{ijk} = |^{ -1}_{ij,k} - |^{ -1}_{ik,j} + |^{ -1}_{mk} |^{ -m}_{ij} - |^{ -1}_{mj} |^{ -m}_{ik}$$

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 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.

28.2.3 Desarrollo de Taylor

El paquete **ctensor** puede truncar resultados e interpretarlos como aproximaciones de Taylor. Este comportamiento se controla con la variable **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
<hr/>	
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))■
```

Esta capacidad del paquete `ctensor` puede ser muy útil; por ejemplo, cuando se trabaja en zonas del campo gravitatorio alejadas del origen de éste.

28.2.4 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:

$$\begin{array}{ccccccccc} & c & & c & & c & & d & e \\ ifb = & (\frac{ifri_1 - ifri_2}{ab}) & ifr_1 & & ifr_2 & & & & \\ & d,e & & e,d & & a & & b \end{array}$$

28.2.5 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 дем.`

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:

$$\begin{aligned} np_1 &= (fri_1 + fri_2) / \sqrt{2} \\ np_2 &= (fri_1 - fri_2) / \sqrt{2} \\ np_3 &= (fri_3 + %i fri_4) / \sqrt{2} \\ np_4 &= (fri_3 - %i fri_4) / \sqrt{2} \end{aligned}$$

psi (*dis*)

Función

Calcula los cinco coeficientes de Newman-Penrose `psi[0]...psi[4]`. Si `psi` 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(exterior schwarzschild,all);
(%o3)          done
(%i4) ug:invert(lg)$
(%i5) weyl(false);
(%o5)          done
(%i6) nptetrad(true);
(%t6) np =

```

$$\begin{bmatrix} \sqrt{r - 2m} & \sqrt{r} & 0 & 0 \\ \hline \sqrt{2} \sqrt{r} & \sqrt{2} \sqrt{r - 2m} & 0 & 0 \\ \sqrt{r - 2m} & \sqrt{r} & r & \frac{\sqrt{r} \sin(\theta)}{\sqrt{2}} \\ \hline \sqrt{2} \sqrt{r} & \sqrt{2} \sqrt{r - 2m} & \sqrt{2} & \sqrt{2} \\ \sqrt{r} & \sqrt{r - 2m} & \sqrt{2} & \frac{\sqrt{r} \sin(\theta)}{\sqrt{2}} \\ \hline 0 & 0 & 0 & -\frac{\sqrt{r} \sin(\theta)}{\sqrt{2}} \\ \sqrt{2} \sqrt{r - 2m} & \sqrt{2} \sqrt{r} & \sqrt{2} & \sqrt{2} \end{bmatrix}$$

```
(%t7) npi = matrix([- -----, -----, 0, 0], [
                      sqrt(2) sqrt(r - 2m)   sqrt(2) sqrt(r)

```

$$\begin{bmatrix} \sqrt{r} & \sqrt{r - 2m} & 0, 0 \\ \hline \sqrt{2} \sqrt{r - 2m} & \sqrt{2} \sqrt{r} & 0, 0 \\ \sqrt{r} & \sqrt{r - 2m} & 0, 0 \\ \hline \sqrt{2} \sqrt{r - 2m} & \sqrt{2} \sqrt{r} & \sqrt{2} \sqrt{r} \\ \sqrt{r} & \sqrt{r - 2m} & \sqrt{2} \sqrt{r} \\ \hline 1 & \frac{\sqrt{r} \sin(\theta)}{\sqrt{2} r} & [0, 0, \frac{1}{\sqrt{2} r}, \frac{\sqrt{r} \sin(\theta)}{\sqrt{2} r \sin(\theta)}] \\ \hline 1 & \frac{\sqrt{r} \sin(\theta)}{\sqrt{2} r} & [0, 0, \frac{1}{\sqrt{2} r}, \frac{\sqrt{r} \sin(\theta)}{\sqrt{2} r \sin(\theta)}] \end{bmatrix})$$

```
(%o7)          done
```

```

(%i7) psi(true);
(%t8)                               psi = 0
                                         0

(%t9)                               psi = 0
                                         1

(%t10)                             psi = -- m
                                         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.

28.2.6 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:

$$\begin{aligned}
& \text{kt} = \frac{-g_{im}^{m} tr_{kj}^{m} - g_{jm}^{m} tr_{ki}^{m} - tr_{ij}^{m} g_{km}^{m}}{2ijk}
\\[10pt]
& \text{kt}_{ij}^{k} = g_{ij}^{k} \text{kt}_{ijm}^{m}
\end{aligned}$$

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:

$$\text{nmc} = \frac{k_i^k - D_{ij}^k n_m^j + g_{ij} n_m^k}{2}$$

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*.

28.2.7 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);
(%o9) [dx - a d + d, 2 a d dxx - a (dx)2 xx - a d dxx x + 2 a d dxx2
      - 2 a dx, a xx2 + a - a]
(%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, 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(exterior schwarzschild,all);
(%o3)                               done
(%i4) depends(f,r);
(%o4)                               [f(r)]
(%i5) cograd(f,g1);
(%o5)                               done
(%i6) listarray(g1);
(%o6)                               [0, fr, 0, 0]
(%i7) contragrad(f,g2);
(%o7)                               done
(%i8) listarray(g2);
      fr - 2 frr m

```

```
(%o8) [0,  $\frac{r}{r}, \frac{r}{r}, 0, 0]$ 
```

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(exterior schwarzschild,all);
(%o3)          done
(%i4) depends(p,r);
(%o4)          [p(r)]
(%i5) factor(dscalar(p));

$$(\%o5) \frac{p_{rr}^2 - 2 m p_{rr} r + 2 p_{rr}^2 r - 2 m p_{rr}}{r^2}$$

```

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 $n = \text{dim}$.

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`.

28.2.8 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:

```

riem      = [
1, 2      [ 0           0           0   0 ]
            [ ]
            [ 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, 1      [          2             ]
            [ r (r - 2 m)       ]
            [ ]
            [ 0           0   0   0 ]
            [ ]
            [ 0           0   0   0 ]]

            [          2 m         ]
            [ ----- 0           0           0 ]
            [          2             ]
            [ r (r - 2 m)       ]
            [ ]
            [ 0           0           0           0 ]
            [ ]
            [ 0           0           0           0 ]]

riem      = [
2, 2      [          m           ]
            [ 0           0   - ----- 0 ]
            [          2             ]
            [ r (r - 2 m)       ]
            [ ]
            [ 0           0           0           0 ]
            [ ]
            [ 0           0           0           0 ]]

```

$$\begin{aligned}
 & \left[\begin{array}{cccc} & & 2 & \\ & & r & (r - 2m) \\ & & & \\ & & m & \\ 0 & 0 & 0 & - \frac{m}{r(r-2m)} \\ & & & 2 \\ & & & r(r-2m) \end{array} \right] \\
 \\
 \text{riem } 2, 3 = & \left[\begin{array}{cccc} 0 & 0 & 0 & 0 \\ & & m & \\ 0 & 0 & - \frac{m}{r(r-2m)} & 0 \\ & & 2 \\ & & r(r-2m) & \\ & & & \\ 0 & 0 & 0 & 0 \\ & & & \\ 0 & 0 & 0 & 0 \end{array} \right] \\
 \\
 \text{riem } 2, 4 = & \left[\begin{array}{ccccc} 0 & 0 & 0 & 0 & \\ & & m & \\ 0 & 0 & 0 & - \frac{m}{r(r-2m)} & \\ & & 2 \\ & & r(r-2m) & \\ & & & \\ 0 & 0 & 0 & 0 & \\ & & & \\ 0 & 0 & 0 & 0 & \end{array} \right] \\
 \\
 \text{riem } 3, 1 = & \left[\begin{array}{ccccc} 0 & 0 & 0 & 0 & \\ & & m & \\ - & 0 & 0 & 0 & \\ r & & & \\ & & \\ 0 & 0 & 0 & 0 & \end{array} \right] \\
 \\
 \text{riem } 3, 2 = & \left[\begin{array}{ccccc} 0 & 0 & 0 & 0 & \\ & & m & \\ 0 & - & 0 & 0 & \\ r & & & \\ & & \\ 0 & 0 & 0 & 0 & \end{array} \right]
 \end{aligned}$$

```

[   m          ]]
[ - - 0 0      0  ]
[ r           ]]
[             ]]
[   m          ]]
[ 0  - - 0 0    0  ]
[             r     ]]
riem = [             ]]
3, 3  [             ]]
[ 0 0 0      0  ]
[             ]]
[   2 m - r   ]]
[ 0 0 0  ----- + 1  ]
[             r     ]]

[ 0 0 0      0  ]
[             ]]
[ 0 0 0      0  ]
[             ]]
riem = [             2 m  ]
3, 4  [ 0 0 0  - ---  ]
[             r  ]]
[             ]]
[ 0 0 0      0  ]

[   0          0 0 0 0  ]
[             ]]
[   0          0 0 0 0  ]
[             ]]
riem = [             0 0 0 0  ]
4, 1  [             ]]
[   2          ]]
[ m sin (theta)  ]]
[ ----- 0 0 0  ]
[   r          ]]

[ 0          0 0 0  ]
[             ]]
[ 0          0 0 0  ]
[             ]]
riem = [ 0          0 0 0  ]
4, 2  [             ]]
[   2          ]]
[ m sin (theta)  ]]
[ 0 ----- 0 0  ]
[   r          ]]

[ 0 0          0 0  ]
[             ]]

```

```

[ 0 0 0 0 ]
[ ] [ ]
riem = [ 0 0 0 0 ]
4, 3 [ ] [ ]
[ ] [ 2 ]
[ ] [ 2 m sin (theta) ]
[ 0 0 - ----- 0 ]
[ ] [ r ] [ ]

[ 2 ]
[ m sin (theta) ] [ ]
[ - ----- 0 0 ] [ ]
[ r ] [ ] [ ]
[ ] [ 2 ]
[ ] [ m sin (theta) ] [ ]
riem = [ 0 - ----- 0 0 ]
4, 4 [ ] [ r ] [ ]
[ ] [ ] [ ]
[ ] [ 2 ]
[ ] [ 2 m sin (theta) ] [ ]
[ 0 0 0 ----- 0 ]
[ ] [ r ] [ ]
[ ] [ ] [ ]
[ 0 0 0 0 ] [ ]

```

(%o5) done

delelen (*L*, *n*)

Función

Devuelve una nueva lista consistente en *L* sin su *n*-ésimo elemento.**28.2.9 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 nmm 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: <code>true</code>	
Es una de las variables que controlan la simplificación de los tensores de Riemann; si vale <code>true</code> , entonces se llevará a cabo la simplificación racional; si <code>ratfac</code> vale <code>true</code> entonces las componentes también serán factorizadas.	
ratweyl	Variable opcional
Valor por defecto: <code>true</code>	
Si vale <code>true</code> , entonces la función <code>weyl</code> llevará a cabo la simplificación racional de los valores del tensor de Weyl. Si <code>ratfac</code> vale <code>true</code> 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 tetradiimensional de Lorentz con signatura $(+, +, +, -)$. Se utiliza cuando <code>cframe_flag</code> vale <code>true</code> .	
ufg	Variable
Es la métrica del sistema de referencia inverso. La calcula <code>lfg</code> cuando <code>cmetric</code> es invocada tomando <code>cframe_flag</code> el valor <code>true</code> .	
riem	Variable
Es el tensor $(3,1)$ de Riemann. Se calcula cuando se invoca la función <code>riemann</code> . Para información sobre el indexado, véase la descripción de <code>riemann</code> .	
Si <code>cframe_flag</code> vale <code>true</code> , <code>riem</code> se calcula a partir del tensor covariante de Riemann <code>lriem</code> .	
lriem	Variable
Es el tensor covariante de Riemann. Lo calcula la función <code>lriemann</code> .	
uriem	Variable
Es el tensor contravariante de Riemann. Lo calcula la función <code>uriemann</code> .	
ric	Variable
Es el tensor de Ricci. Lo calcula la función <code>ricci</code> .	
uric	Variable
Es el tensor contravariante de Ricci. Lo calcula la función <code>uricci</code> .	
lg	Variable
Es el tensor métrico. Este tensor se debe especificar (como matriz cuadrada de orden <code>dim</code>) antes de que se hagan otros cálculos.	
ug	Variable
Es la inversa del tensor métrico. Lo calcula la función <code>cmetric</code> .	

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 Kretschmann, tal como la calcula la función rinviant .	
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)); (%o39)	$\begin{bmatrix} 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
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 <code>ct_coords</code> contiene una lista de coordenadas. Aunque se define normalmente cuando se llama a la función <code>csetup</code> , también se pueden redefinir las coordenadas con la asignación <code>ct_coords: [j1, j2, ..., jn]</code> donde <code>j</code> es el nuevo nombre de las coordenadas. Véase también <code>csetup</code> .	

28.2.10 Nombres reservados

Los siguientes nombres se utilizan internamente en el paquete `ctensor` y no deberían redefinirse:

Nombre	Descripción
<code>_lg()</code>	Toma el valor <code>lfg</code> si se utiliza métrica del sistema de referencia, <code>lg</code> en otro caso
<code>_ug()</code>	Toma el valor <code>ufg</code> si se utiliza métrica del sistema de referencia, <code>ug</code> en otro caso
<code>cleanup()</code>	Elimina elementos de la lista <code>deindex</code>
<code>contract4()</code>	Utilizada por <code>psi()</code>
<code>filemet()</code>	Utilizada por <code>csetup()</code> cuando se lee la métrica desde un fichero
<code>findde1()</code>	Utilizada por <code>findde()</code>
<code>findde2()</code>	Utilizada por <code>findde()</code>
<code>findde3()</code>	Utilizada por <code>findde()</code>
<code>kdelt()</code>	Delta de Kronecker (no generalizada)
<code>newmet()</code>	Utilizada por <code>csetup()</code> para establecer una métrica interactivamente
<code>setflags()</code>	Utilizada por <code>init_ctensor()</code>
<code>readvalue()</code>	
<code>resimp()</code>	
<code>sermet()</code>	Utilizada por <code>csetup()</code> para definir una métrica como serie de Taylor
<code>txyzsum()</code>	
<code>tmetric()</code>	Métrica del sistema de referencia, utilizada por <code>cmetric()</code> cuando <code>cframe_flag:true</code>
<code>riemann()</code>	Tensor de Riemann en la base del sistema de referencia, se utiliza cuando <code>cframe_flag:true</code>
<code>tricci()</code>	Tensor de Ricci en la base del sistema de referencia, se utiliza cuando <code>cframe_flag:true</code>
<code>trrc()</code>	Coeficientes de rotación de Ricci, utilizada por <code>christof()</code>
<code>yesp()</code>	

29 atensor

29.1 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 commutativo ("."). 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 $\text{Cl}(0,2)$ con una base de dos vectores. Las tres unidades imaginarias son los dos vectores de la base junto con su producto:

$$\begin{array}{lll} i = v & j = v & k = v \cdot v \\ 1 & 2 & 1 \quad 2 \end{array}$$

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);
(%o5)
 [ 0  0  0  0 ]
 [               ]
 [ 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 ]
[                               ]
[      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.

29.2 Funciones y variables para atensor

init_atensor (alg-type, opt-dims)
init_atensor (alg-type)

Función
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 commutación. 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 commutación. 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)
(%i3) aform:matrix([0,3,-2],[-3,0,1],[2,-1,0]);
          3
          [  0   3   - 2 ]
          [                   ]
(%o3)          [ - 3   0   1 ]
          [                   ]
          [  2   - 1   0 ]
(%i4) asymbol:x;
(%o4)           x
(%i5) av(x[1],x[2]);
(%o5)           x
          3
```

abasep (*v*)

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**.

30 Series

30.1 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.

30.2 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)
(%i2) powerseries (f(x), x, 0);
(%o2)
(%i3) taylor (exp (sqrt (f(x))), x, 0, 4);
(%o3)/T/
```

$$\frac{1 + x + \frac{x^2}{18432} + \frac{3073 x^3}{12817} + \frac{12817 x^4}{307200} + \dots}{2}$$

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 no esté en uso. Si todas las variables de la lista han sido ya revisadas, se formarán nuevos índices 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)
(%i2) product (sum (f (foo + i*j*bar), foo, 1, inf), bar, 1, inf);
(%o2)
(%i3) niceindices (%);
(%o3)
```

$$\frac{\text{f}(\text{bar } i \text{ } j + \text{foo})}{\text{bar} = 1}$$

$$\text{foo} = 1$$

```
(%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);
(%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)]

```

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.

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).

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 progresá 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 expresión racional se muestra completamente expandida. La variable **ratexpand** tiene el mismo efecto.

Si **psexpand** vale **false**, las expresiones 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
          x      x      x      x      x
(%i5) taylor (log(x+1), x, 0, 6);
          2      3      4      5      6
          x      x      x      x      x
(%o5)/T/   x - --- + --- - --- + --- - --- + . . .
          2      3      4      5      6
(%i6) ratsimp (revert (t, x) - taylor (log(x+1), x, 0, 6));
(%o6)
(%i7) revert2 (t, x, 4);
          4      3      2
          x      x      x
(%o7)   - --- + --- - --- + x
          4      3      2

```

taylor (<i>expr</i> , <i>x</i> , <i>a</i> , <i>n</i>)	Función
taylor (<i>expr</i> , [<i>x</i> _1, <i>x</i> _2, ...], <i>a</i> , <i>n</i>)	Función
taylor (<i>expr</i> , [<i>x</i> , <i>a</i> , <i>n</i> , 'asymp])	Función
taylor (<i>expr</i> , [<i>x</i> _1, <i>x</i> _2, ...], [<i>a</i> _1, <i>a</i> _2, ...], [<i>n</i> _1, <i>n</i> _2, ...])	Función
taylor (<i>expr</i> , [<i>x</i> _1, <i>a</i> _1, <i>n</i> _1], [<i>x</i> _2, <i>a</i> _2, <i>n</i> _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 $n^{2^{\text{taylordepth}}}$.

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
          1 + 2.5 x + 3.375 x  + 6.5625 x  + . . .
(%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
- x - -- + . . .
           6
(%o8)/T/
(%i9) taylor ((cos(x) - sec(x))^3, x, 0, 5);
(%o9)/T/
(%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           2
           y   y           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
                                         + (--- + . . .) 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 $n \cdot 2^{\text{taylordepth}}$.

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
      + (1 - a - 2 a (y - a) - (y - a ) ) x
      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 <i>expr</i> . 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.	
taylorat (expr)	Función
Convierte <i>expr</i> 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 <i>expr</i> de tal forma como si sus sumas fuesen una serie truncada de Taylor. La expresión <i>expr</i> no sufre ninguna otra modificación.	
Ejemplo:	
<pre>(%i1) expr: x^2 + x + 1; (%o1) (%i2) trunc (expr); (%o2) (%i3) is (expr = trunc (expr)); (%o3)</pre>	
$\begin{aligned} &x^2 + x + 1 \\ &1 + x + x^2 + \dots \\ &\text{true} \end{aligned}$	

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);
(%o1)          g(p) :=  $\frac{p^4}{\binom{2n}{n}}$ 
(%i2) g(n^4);
(%o2)           $\frac{n^4}{\binom{4n}{4}}$ 
(%i3) nusum (% , n , 0 , n);
(%o3)           $\frac{2(n+1)(63n^4 + 112n^3 + 18n^2 - 22n + 3)}{693\binom{2n}{n}}$ 
(%i4) unsum (% , n);
(%o4)           $\frac{n^4}{\binom{4n}{4}}$ 
```

verbose

Variable opcional

Valor por defecto: **false**

Si **verbose** vale **true**, la función **powerseries** va imprimiendo mensajes durante su ejecución.

30.3 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 (<i>a, b</i>)	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 (<i>series, sinfn, cosfn</i>)	Función
Aplica las funciones <i>sinfn</i> a los términos sinusoidales y las funciones <i>cosfn</i> a los cosenoidales de la serie de Poisson dada. Tanto <i>sinfn</i> como <i>cosfn</i> son funciones de dos argumentos, los cuales son un coeficiente y una parte trigonométrica de un término de la serie.	
poisplus (<i>a, b</i>)	Función
Idéntico a intopois (<i>a + b</i>).	
poissimp (<i>a</i>)	Función
Convierte <i>a</i> en una serie de Poisson para <i>a</i> 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 (<i>a, b, c</i>)	Función
Sustituye <i>b</i> por <i>a</i> en <i>c</i> , donde <i>c</i> es una serie de Poisson.	
(1) Si <i>b</i> es una de las variables <i>u, v, w, x, y</i> o <i>z</i> , entonces <i>a</i> debe ser una expresión lineal en esas variables (por ejemplo, $6*u + 4*v$).	
(2) Si <i>b</i> no es ninguna de esas variables, entonces <i>a</i> no puede contener tampoco a ninguna de ellas, ni senos, ni cosenos.	
poistimes (<i>a, b</i>)	Función
Idéntico a intopois (<i>a*b</i>).	
printpois (<i>a</i>)	Función
Presenta una serie de Poisson en un formato legible. Conjuntamente con outofpois , si es necesario convertirá a primero en una codificación de Poisson.	

31 Teoría de Números

31.1 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  
          [1, - -, -, 0, - --, 0, --, 0, - --]  
          2   6   30  42   30  
(%i3) zerobern: false$  
(%i4) map (bern, [0, 1, 2, 3, 4, 5, 6, 7, 8]);  
          1   1   1   5   691   7   3617   43867  
          [1, - -, -, - --, --, - -----, -, - -----, -----]  
          2   6   30  66   2730  6   510   798
```

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.

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)
```

burn (*n*)

Función

Devuelve el *n*-ésimo número de Bernoulli del entero *n*. La función **burn** puede ser más eficiente que **bern** para *n* grande (mayor que 105, por ejemplo), pues **bern** calcula todos los números de Bernoulli hasta *n* antes de devolver el resultado.

La función **burn** se beneficia del hecho de que los números racionales de Bernoulli pueden aproximarse por funciones zeta con una eficiencia aceptable.

Es necesario cargar en memoria esta función haciendo `load ("bffac")`.

cf (*expr*)

Función

Transforma *expr* a fracciones continuas. La expresión *expr* debe contener fracciones continuas y raíces cuadradas de números enteros. Los operandos de la expresión pueden combinarse con operadores aritméticos. Además de fracciones continuas y raíces cuadradas, los factores de la expresión deben ser enteros o números racionales. Maxima no tiene más conocimiento sobre operaciones con fracciones continuas que el que aporta 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 $a + 1/(b + 1/(c + \dots))$ se representa como la lista `[a, b, c, ...]`, donde los elementos *a*, *b*, *c*, ... se evalúan como enteros. La expresión *expr* puede contener también **sqr** (*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, 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 $a + 1/(b + 1/(c + \dots))$ a partir de la representación en formato lista de la fracción continua $[a, b, c, \dots]$.

```
(%i1) cf ([1, 2, -3] + [1, -2, 1]);
(%o1) [1, 1, 1, 2]
(%i2) cfdisrep (%);
(%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, 2]
(%i5) cflength: 3$ 
(%i6) cf ((1 + sqrt(5))/2);
(%o6) [1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
```

divsum (n, k)
divsum (n)Función
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*.

Para la constante de Euler-Mascheroni, véase **%gamma**.

```
(%i1) map (euler, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
(%o1) [1, 0, - 1, 0, 5, 0, - 61, 0, 1385, 0, - 50521]
```

%gamma

Constante

Es la constante de Euler-Mascheroni, 0.5772156649015329

factorial (x) Función

Representa la función factorial. Maxima considera **factorial (x)** lo mismo que $x!$.
Véase !.

fib (n) Función

Devuelve el n -ésimo número de Fibonacci. La llamada **fib(0)** es igual a 0, **fib(1)** devuelve 1 y **fib (-n)** es igual a $(-1)^{n+1} * \text{fib}(n)$.

Después de llamar a **fib**, la variable **prevfib** toma el valor **fib (x - 1)**, que es el número de Fibonacci que precede al último calculado.

```
(%i1) map (fib, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
(%o1) [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

fibtophi (expr) 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));
(%o1) 
$$\frac{\frac{n}{\phi} - (1 - \phi)}{2\phi - 1}$$

(%i2) fib (n-1) + fib (n) - fib (n+1);
(%o2) 
$$-\text{fib}(n + 1) + \text{fib}(n) + \text{fib}(n - 1)$$

(%i3) fibtophi (%);
(%o3) 
$$\frac{\frac{n + 1}{\phi} - (1 - \phi)}{2\phi - 1} + \frac{\frac{n}{\phi} - (1 - \phi)}{2\phi - 1}$$


$$+ \frac{\frac{n - 1}{\phi} - (1 - \phi)}{2\phi - 1}$$

(%i4) ratsimp (%);
(%o4) 0
```

ifactors (n) Función

Devuelve la factorización del argumento n , siendo éste un número entero positivo. Si $n=p_1^{e_1}..p_k^{e_k}$ es la descomposición de n en números primos, **ifactors** devuelve $[[p_1, e_1], \dots, [p_k, e_k]]$.

Los métodos de factorización son las divisiones tentativas hasta el 9973, así como los métodos rho de Pollard y el de la curva elíptica.

```
(%i1) ifactors(51575319651600);
(%o1) [[2, 4], [3, 2], [5, 2], [1583, 1], [9050207, 1]]
(%i2) apply("*", map(lambda([u], u[1]^u[2]), %));
(%o2) 51575319651600
```

inrt (<i>x, n</i>)	Función
Devuelve la raíz entera <i>n</i> -ésima del valor absoluto de <i>x</i> .	
(%i1) 1: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]\$	
(%i2) map (lambda ([a], inrt (10^a, 3)), 1);	
(%o2) [2, 4, 10, 21, 46, 100, 215, 464, 1000, 2154, 4641, 10000]	
inv_mod (<i>n, m</i>)	Función
Calcula el inverso de <i>n</i> módulo <i>m</i> . La llamada inv_mod (<i>n,m</i>) devuelve false si <i>n</i> es un divisor nulo módulo <i>m</i> .	
(%i1) inv_mod(3, 41);	
(%o1) 14	
(%i2) ratsimp(3^-1), modulus=41;	
(%o2) 14	
(%i3) inv_mod(3, 42);	
(%o3) false	
jacobi (<i>p, q</i>)	Función
Devuelve el símbolo de Jacobi para <i>p</i> y <i>q</i> .	
(%i1) 1: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]\$	
(%i2) map (lambda ([a], jacobi (a, 9)), 1);	
(%o2) [1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0]	
lcm (<i>expr_1, ..., expr_n</i>)	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") .	
next_prime (<i>n</i>)	Función
Devuelve el menor de los primos mayores que <i>n</i> .	
(%i1) next_prime(27);	
(%o1) 29	
partfrac (<i>expr, var</i>)	Función
Expande la expresión <i>expr</i> en fracciones parciales respecto de la variable principal <i>var</i> . 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);	
(%o1)	$\frac{2}{x+2} - \frac{2}{x+1} + \frac{1}{(x+1)^2}$
(%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 $a^n \bmod m$, 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 10^{16} , 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 10^{16} `primep` realiza un número de pruebas de seudo-primalidad de Miller-Rabin igual a `primep_number_of_tests` y una prueba de seudo-primalidad de Lucas. La probabilidad de que 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 iguales a cero. See `bern`.**zeta (n)**

Función

Devuelve la función zeta de Riemann para x entero negativo, 0, 1 o número par positivo. No se evalúa `zeta (n)` para cualesquiera otros argumentos, incluyendo racionales no enteros, números en coma flotante o argumentos complejos.Véanse también `bfzeta` y `zeta%pi`.

```
(%i1) map (zeta, [-4, -3, -2, -1, 0, 1, 2, 3, 4, 5]);
              2          4
              1      1      1      %pi      %pi
(%o1) [0, ---, 0, - ---, - -, inf, -----, zeta(3), -----, zeta(5)]
         120      12      2      6          90
```

zeta%pi

Variable opcional

Valor por defecto: `true`Si `zeta%pi` vale `true`, `zeta` devuelve una expresión proporcional a π^n si n es un número par positivo. En caso contrario, `zeta` no se evalúa y devuelve la forma nominal `zeta (n)`.

```
(%i1) zeta%pi: true$
(%i2) zeta (4);
(%o2)                               4
                               %pi
                               -----
                               90
(%i3) zeta%pi: false$
(%i4) zeta (4);
(%o4)                               zeta(4)
```

32 Simetrías

Paquete escrito para Macsyma-Symbolics por Annick Valibouze¹. Los algoritmos están descritos en los siguientes artículos:

1. Fonctions symétriques et changements de bases². Annick Valibouze. EUROCAL'87 (Leipzig, 1987), 323–332, Lecture Notes in Comput. Sci 378. Springer, Berlin, 1989.
2. Résolvantes et fonctions symétriques³. 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 Macsyma⁴. 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

32.1 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
          2 a   b x   y + x
(%o1)
(%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 www-calfor.lip6.fr/~avb

² www.stix.polytechnique.fr/publications/1984-1994.html

³ 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 mayores, la función `tcontract` comprueba la simetría del polinomio.

```
(%i1) psym: explode (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 $lvar_1, \dots, lvar_n$, y en los polinomios p_1, \dots, 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
      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   2   4
(%o3) y - 2 e1 f1 y + ((2 e1 - 6 e2) f2 + (2 e2 + e1) f1 ) y
                                         3   3
                                         + ((9 e3 + 5 e1 e2 - 2 e1) f1 f2 + (- 2 e3 - 2 e1 e2) f1 ) y
                                         2   2   4
                                         + ((9 e2 - 6 e1 e2 + e1) f2
                                         2   2   2   2   2   4
                                         + (- 9 e1 e3 - 6 e2 + 3 e1 e2) f1 f2 + (2 e1 e3 + e2) f1 )
                                         2   2   3   2
                                         y + (((9 e1 - 27 e2) e3 + 3 e1 e2 - e1 e2) f1 f2)
```

$$\begin{aligned}
& + ((15 e_2 - 2 e_1) e_3 - e_1 e_2) f_1^2 f_2^3 - 2 e_2 e_3 f_1^5 y \\
& + (-27 e_3^2 + (18 e_1 e_2 - 4 e_1) e_3^3 - 4 e_2^3 + e_1 e_2^2) f_2^3 \\
& + (27 e_3^2 + (e_1 - 9 e_1 e_2) e_3^3 + e_2^2) f_1^3 f_2^2 \\
& + (e_1 e_2 e_3 - 9 e_3) f_1^2 f_2^4 + e_3^6 f_1^6
\end{aligned}$$

Búsqueda del polinomio cuyas raíces son la suma $a+u$ o a es la raíz de $z^2 - e_1 * z + e_2$ y u es la raíz de $z^2 - f_1 * z + f_2$

```
(%i1) ratsimp (direct ([z^2 - e1*z + e2, z^2 - f1*z + f2],
                      z, a + u, [[u], [a]]));
(%o1) y^4 + (-2 f1 - 2 e1) y^3 + (2 f2 + f1^2 + 3 e1 f1 + 2 e2
+ e1^2) y^2 + ((-2 f1 - 2 e1) f2 - e1 f1^3 + (-2 e2 - e1) f1^2
- 2 e1 e2) y^1 + f2^2 + (e1 f1^2 - 2 e2 + e1) f2^3 + e2 f1^2 + e1 e2 f1^2
+ e2^2
```

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`, `pui_reduc` 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 . $l = [n, e_{-1}, \dots, e_{-n}]$, 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`, `puireduce` y `schur2comp`.

elem (ele, sym, lvar)

Función

Descomponer 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 (%);
(%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`, `puireduce` y `schur2comp`.

explode (pc, lvar)

Función

Devuelve el polinomio simétrico asociado a la forma contraída `pc`. La lista `lvar` contiene las variables.

```
(%i1) explode (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 también `lgtreillis`, `treillis` y `treinat`.

mon2schur (*l*)

Función

La lista *l* representa la función de Schur S_l : Se tiene $l = [i_1, i_2, \dots, i_q]$ con $i_1 \leq i_2 \leq \dots \leq i_q$. La función de Schur es $S_{[i_1, i_2, \dots, i_q]}$, el menor de la matriz infinita (h_{i-j}) $i \geq 1, j \geq 1$ compuesto de las *q* primeras filas y columnas $i_1 + 1, i_2 + 2, \dots, i_q + q$.

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, \dots

```
(%i1) mon2schur ([1, 1, 1]);
(%o1)                                x1 x2 x3
(%i2) mon2schur ([3]);
(%o2)      x1 x2 x3 + x1^2 x2 + x1 x3^2
(%i3) mon2schur ([1, 2]);
(%o3)      2 x1 x2 x3 + x1^2 x2^2
```

Para 3 variables se tendrá:

$$\begin{aligned} & 2 x_1 x_2 x_3 + x_1^2 x_2 + x_2^2 x_1 + x_1^2 x_3 + x_3^2 x_1 \\ & + x_2^2 x_3 + x_3^2 x_2 \end{aligned}$$

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]]);
(%o1)           - 2 f2 + f1 (f1 + e1) - 3 e1 e2 + e1
(%i2) ratsimp (%);
(%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*, *puireduce* 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]]);
(%o1)           t2 + p1 t1 + ----- - -----
                  3   p1   p2   p1
                  2       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]);
(%o2)           [y^2 + 2 y, x^2 + 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]);
(%o1)           3 4
                  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);
(%o2)           7      5      3
                  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)
(%i2) pui ([3, a, b], u*x*y*z, [x, y, z]);
(%o2)
      2
      a (a - b) u   (a b - p3) u
      -----
      6           3
(%i3) ratsimp (%);
(%o3)
      3
      (2 p3 - 3 a b + a ) u
      -----
      6
```

Otras funciones para cambio de bases son: **comp2ele**, **comp2pui**, **ele2comp**, **ele2pui**, **elem**, **mon2schur**, **multi_elem**, **multi_pui**, **pui2comp**, **pui2ele**, **puireduce** 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, []);
(%o1)
      2
      p2 + p1
      [2, p1, -----]
      2
(%i2) pui2comp (3, [2, a1]);
(%o2)
      2
      a1 (p2 + a1 )
      2   p3 + ----- + a1 p2
      p2 + a1           2
      [2, a1, -----, -----]
```

```
(%i3) ratsimp (%);
(%o3)      [2, a1,  $\frac{p_2 + a_1}{2}$ ,  $\frac{2 p_3 + 3 a_1 p_2 + a_1}{6}]$ 
```

Otras funciones para cambio de bases son: `comp2ele`, `comp2pui`, `ele2comp`, `ele2pui`, `elem`, `mon2schur`, `multi_elem`, `multi_pui`, `pui`, `pui2ele`, `pui2pui` y `schur2comp`.

pui2ele (n, lpu) 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 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`, `pui2pui` y `schur2comp`.

pui2polynome (x, lpu) Función

Calcula el polinomio en x cuyas raíces tienen como funciones potenciales las dadas en la lista `lpu`.

```
(%i1) pui;
(%o1)
(%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, %);
(%o3)  $x^3 - 4x^2 + 5x - 1$ 
```

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 $lvar_1, \dots, lvar_n$. Sea c_i el número de variables en $lvar_i$ y SC el producto de los n grupos simétricos de grados c_1, \dots, 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 $c_1 \leq d_1, c_2 \leq d_2, \dots, c_n \leq d_n$. Por último, sea SD el producto de los grupos simétricos $S_{d1} \times S_{d2} \times \dots \times S_{dn}$.

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) 1: [[x, y], [a, b]];
(%o1) [[x, y], [a, b]]
(%i2) pui_direct (multi_orbit (a*x + b*y, 1), 1, [2, 2]);
(%o2) [a x, 4 a b x y + a x ]
```

```
(%i3) pui_direct (multi_orbit (a*x + b*y, 1), 1, [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]);
(%o1) [2, p1, p2, p1 p2 - -----
           2
           p1 (p1 - p2)
           2
(%i2) ratsimp (%);
(%o2) [2, p1, p2, -----
           3
           3 p1 p2 - p1
           2]
```

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

Función

Calcula la resolvente del polinomio *P* de variable *x* y grado *n* $\geq d$ 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
```

```
(%i17) 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 "
(%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 neutro 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` y `resolvante_diedrale`.

resolvante.Alterneel (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` y `resolvante_bipartite`.

resolvante_bipartite (P, x)

Función

Calcula la transformación de $P(x)$ de grado n (n par) por la función $x_1x_2\dots x_{\lfloor n/2 \rfloor} + x_{\lfloor n/2 \rfloor + 1} \dots x_n$

```
(%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` y `resolvante.Alterneel`.

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` y `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` y `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` y `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` y `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` y `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` y `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 variables 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 variables 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**.

trellis (*n*)

Función

Devuelve todas las particiones de pesos *n*.

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

Véanse también **lgtrellis**, **ltrellis** 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 **lgtrellis**, **ltrellis** y **treillis**.

33 Grupos

33.1 Funciones y variables para grupos

todd_coxeter (*relaciones, subgrupo*)
todd_coxeter (*relaciones*)

Función
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 $[H, Hg_2, Hg_3, \dots]$. 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
```


34 Entorno de Ejecución

34.1 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.

34.2 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 depurador 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.

34.3 Funciones y variables para el entorno de ejecución

`feature`

Declaración

Maxima trata con dos tipos diferentes de características, las del sistema y las correspondientes a expresiones matemáticas. Véase también `status` para información sobre características del sistema. Véanse asimismo `features` y `featurep` para información sobre características matemáticas.

`feature` no es el nombre de una función ni de una variable.

featurep (a, f)

Función

Intenta determinar si el objeto *a* tiene la característica *f* en base a la información en la base de datos actual. De ser así, devuelve **true**, en caso contrario **false**.

Nótese que **featurep** devuelve **false** cuando ni *f* ni la negación *f* puedan determinarse.

La función **featurep** evalúa sus argumentos.

Véanse también **declare** y **features**.

```
(%i1) declare (j, even)$
(%i2) featurep (j, integer);
(%o2)                                true
```

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 descripción 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.

status (feature)	Función
status (feature, putative_feature)	Función
status (status)	Función

Devuelve información sobre la presencia o ausencia de ciertas características 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, putative_feature)** devuelve **true** si *putative_feature* está en la lista de elementos retornados por **status (feature)** y **false** en otro caso. La función **status** no evalúa el argumento *putative_feature*. El operador comilla-comilla, ‘’, permite la evaluación. Una característica cuyo nombre contenga un carácter especial debe ser suministrada como un argumento del tipo cadena alfanumérica. Por ejemplo, **status (feature, "ansi-cl")**.
- La llamada **status (status)** devuelve una lista con dos elementos [**feature**, **status**]. Los elementos **feature** y **status** son los dos argumentos que acepta la función **status**; no está claro si esta lista tiene algún interés adicional.

La variable **features** contiene una lista de características que se aplican a expresiones matemáticas. Véanse **features** y **featurep** para más información.

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
--------------------	---------

Devuelve una cadena alfanumérica con la hora y fecha actuales. La cadena tiene el formato **HH:MM:SS Day, mm/dd/yyyy (GMT-n)**, donde los campos son: las horas, minutos, segundos, día de la semana, mes, día del mes, año y número de horas de diferencia con respecto a la hora GMT.

Ejemplo

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

absolute_real_time ()	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
```

35 Miscelánea de opciones

35.1 Introducción a la miscelánea de opciones

En esta sección se comentan varias opciones que tienen un efecto global sobre el comportamiento de Maxima. También se comentan varias listas, como la de las funciones definidas por el usuario.

35.2 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`.

La llamada `:lisp *maxima-sharedir*` muestra la localización del directorio "share" dentro del sistema de ficheros del usuario.

La llamada `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".

35.3 Funciones y variables para la miscelánea de opciones

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`.

alphabetic

Declaración

`alphabetic` es una declaración de tipo reconocida por `declare`. La expresión `declare(s, alphabetic)` indica a Maxima que reconozca como alfabéticos todos los caracteres de `s`, que debe ser una cadena alfanumérica.

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)                                [%y`xx, `xx@\@yy\~]
```

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`.

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.

inf

Constante

Símbolo que identifica al infinito positivo dentro de un contexto de números reales.

infinity

Constante

Símbolo que identifica al infinito complejo, una magnitud infinita con ángulo de fase arbitrario. Véanse también `inf` y `minf`.

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 <code>alias</code> , <code>ordergreat</code> o <code>orderless</code> o por haber declarado el átomo como <code>noun</code> (nombre) con <code>declare</code> .
dependencies	Átomos que tienen dependencias funcionales, creados por las funciones <code>depends</code> o <code>gradef</code> .
gradefs	Funciones que tienen derivadas definidas por el usuario, creadas por la función <code>gradef</code> .
props	Todos los átomos que tengan cualquier propiedad que no sea de las mencionadas hasta ahora, como las establecidas por <code>atvalue</code> , <code>matchdeclare</code> , etc., así como propiedades especificadas en la función <code>declare</code> .
let_rule_packages	Todos los paquetes de reglas <code>let</code> definidos por el usuario, junto con el paquete especial <code>default_let_rule_package</code> ; <code>default_let_rule_package</code> es el nombre del paquete de reglas utilizado cuando no se use ningún otro especificado por el usuario.

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

m1pbranch

Variable optativa

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

```

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]

```

properties (a)

Función

Devuelve una lista con los nombres de propiedades asociadas con el átomo *a*.

props

Símbolo especial

props son átomos que tienen cualquier propiedad diferente de las mencionadas explícitamente en **infolists**, tales como las especificadas por **atvalues**, **matchdeclares**, etc., así como las propiedades especificadas mediante la función **declare**.

propvars (*prop*)

Función

Devuelve una lista con aquellos átomos de la lista **props** que tienen la propiedad indicada por *prop*.

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.

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 modo que **put**, excepto que sus argumentos no son evaluados.

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
```

rem (*átomo, indicador*) Función

Elimina del *átomo* la propiedad indicada por *indicador*.

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`, o de las que define el usuario.

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 llamada `remove` devuelve siempre `done` independientemente que haya algún átomo con la propiedad especificada.

remvalue (*nombre_1, ..., nombre_n*)
remvalue (*all*)

Función
Función

Elimina del sistema los valores de las variables 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 ha 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 `pffformat: 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)`.

scalarmp (*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.

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)      -----
           2
           (%i4) legendre_p (1, %pi);
(%o4)                  %pi
(%i5) legendre_q (1, %pi);
           %pi + 1
           %pi log(-----)
           1 - %pi
(%o5)      -----
           2
```


36 Reglas y patrones

36.1 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`.

36.2 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. Despues 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, x₁, ..., x_n) Función
defmatch (progname, pattern) Función

Define una función `nombre_prog(expr, x1, ..., xn)` que analiza si `expr` coincide con el patrón.

El argumento patrón es una expresión que contiene los argumentos de patrón `x1, ..., xn` 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 `x1, ..., xn` 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` ningún 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) disp rule (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);
(%i3)      a1 a2! --> a1! where oneless(a2, a1)
(%i4) letrat: true$ 
(%i5) let (a1!/a1, (a1-1)!);
(%i5)      a1!
(%o5)      --- --> (a1 - 1) !
(%i6) letsimp (n*m!*(n-1)!/m);
(%o6)      (m - 1)! n!
(%i7) let (sin(a)^2, 1 - cos(a)^2);
(%o7)      sin (a) ^2 --> 1 - cos (a)^2
(%i8) letsimp (sin(x)^4);
(%o8)      4          2
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)!);
(%o2)      n!
(%o2)      -- --> (n - 1) !
(%i3) letrat: false$ 
(%i4) letsimp (a!/a);
(%o4)      a!
(%o4)      --
(%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 `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 (<i>expr</i>)	Función
letsimp (<i>expr, nombre_paquete</i>)	Función
letsimp (<i>expr, nombre_paquete_1, ..., nombre_paquete_n</i>)	Función
Aplica repetidamente las reglas definidas por <code>let</code> hasta que no se puedan hacer más cambios en <i>expr</i> .	
La llamada <code>letsimp (expr)</code> utiliza las reglas de <code>current_let_rule_package</code> .	
La llamada <code>letsimp (expr, nombre_paquete)</code> utiliza las reglas de <i>nombre_paquete</i> sin efectuar cambios en <code>current_let_rule_package</code> .	
La llamada <code>letsimp (expr, nombre_paquete_1, ..., nombre_paquete_n)</code> es equivalente a <code>letsimp (expr, nombre_paquete_1, ...)</code> seguida de <code>letsimp (% , nombre_paquete_2)</code> y así sucesivamente.	

let_rule_packages	Variable opcional
Valor por defecto: <code>[default_let_rule_package]</code>	
La variable <code>let_rule_packages</code> guarda una lista con todos los paquetes de reglas definidos por el usuario, junto con el paquete por defecto <code>default_let_rule_package</code> .	

matchdeclare (<i>a_1, pred_1, ..., a_n, pred_n</i>)	Función
Asocia un predicado <i>pred_k</i> con una variable o lista de variables <i>a_k</i> , de forma que <i>a_k</i> se comparará con expresiones para las cuales el predicado devuelva algo que no sea <code>false</code> .	
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, <code>true</code> o <code>all</code> . Cualquier expresión se hace coincidir con <code>true</code> o <code>all</code> .	
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 <code>matchdeclare</code> ; el predicado no se evalúa hasta que se ensaya una comparación.	
Un predicado puede devolver tanto una expresión booleana, como <code>true</code> o <code>false</code> . Las expresiones booleanas se evalúan con <code>is</code> dentro de la regla, por lo que no es necesario llamar a <code>is</code> 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` (vía `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)]

```

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 `displfun (l delimiter)` 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 prácticamente 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 ~;
(%o7)                                     -----  
                                >>z + foox, yoof + w<<  
                                @@p, q~
```

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

```
(%i1) matchfix ("!-", "-!");
(%o1)                                     " !-"
(%i2) !- x, y -! := x/y - y/x;
(%o2)                                     !-x, y-! := - - -  
                                         x   y  
                                         y   x
(%i3) define (!-x, y-!, x/y - y/x);
(%o3)                                     !-x, y-! := - - -  
                                         x   y  
                                         y   x
(%i4) define ("!-" (x, y), x/y - y/x);
(%o4)                                     !-x, y-! := - - -  
                                         x   y  
                                         y   x
(%i5) displfun ("!-");
(%t5)                                     !-x, y-! := - - -  
                                         x   y  
                                         y   x
(%o5)                                     done
```

(%i6) !-3, 5-!;	16
(%o6)	- --
	15
(%i7) "!"-(3, 5);	16
(%o7)	- --
	15

remlet (<i>prod, nombre</i>)	Función
remlet ()	Función
remlet (<i>all</i>)	Función
remlet (<i>all, nombre</i>)	Función

Elimina la última regla de sustitución `prod → repl` que haya sido definida por la función `let`. Si se suministrar 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 usuario (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:

```

(%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, foerule1);
(%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;
(%o11)          (sin(x) + 1)^2
(%i12) expand (%);
(%o12)          2 sin(x) - cos (x) + 2
(%i13) sin(x)^2;
(%o13)          1 - cos (x)^2
(%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;
(%o17)          1 - cos (y)^2

```

tellsimpafter (*patrón, reemplazamiento*)

Función

Define una regla de simplificación que el simplificador aplicará después de las reglas de simplificación propias 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 coincide 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 por 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);
(%o3)                                1 sqrt(2) sqrt(3)
                                         [-, -----, -----, 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 través 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 anticomutativo.

```
(%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 `^`.

37 Listas

37.1 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))
```

37.2 Funciones y variables para listas

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*.

atom (*expr*)

Función

Devuelve **true** si *expr* es un átomo (número, nombre o cadena alfanumérica) y **false** en caso contrario. 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).

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, x₁, list₁, ..., x_n, list_n*) Función
 Crea una lista mediante la evaluación de *form* con *x₁* tomando cada uno de los valores de *list₁*, para cada uno de estos valores liga *x₂* con cada elemento de *list₂*, 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.

Por ejemplo:

```
(%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.

Por ejemplo:

```
(%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 (*expr₁, expr₂*) Función
delete (*expr₁, expr₂, n*) Función

delete(*expr₁, expr₂*) elimina de *expr₂* cualesquiera argumentos del operador del nivel superior que sean iguales a *expr₁*. Nótese que los argumentos de las subexpresiones no se ven afectados por esta función.

expr₁ puede ser un átomo o una expresión no atómica. *expr₂* puede ser cualquier expresión no atómica. La función **delete** devuelve una nueva expresión sin modificar *expr₂*.

delete(*expr₁, expr₂, n*) elimina de *expr₂* los primeros *n* argumentos del operador del nivel superior que sean iguales a *expr₁*. 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]);
(%o6) [x^2 - y^2]
```

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.

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.

```
(%i1) put (%e, 'transcendental, 'type);
(%o1)                      transcendental
(%i2) put (%pi, 'transcendental, 'type)$
(%i3) put (%i, 'algebraic, 'type)$
(%i4) typeof (expr) := block ([q],
                                if numberp (expr)
                                then return ('algebraic),
                                if not atom (expr)
                                then return (maplist ('typeof, expr)),
                                q: get (expr, 'type),
                                if q=false
                                then errcatch (error(expr,"is not numeric.")) else q)$
(%i5) typeof (2*%e + x*%pi);
x is not numeric.
(%o5)  [[transcendental, []], [algebraic, transcendental]]
(%i6) typeof (2*%e + %pi);
(%o6)      [transcendental, [algebraic, transcendental]]
```

join (l, m)

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 (*expr*, *i*, *i_0*, *i_1*) Función
makelist (*expr*, *x*, *list*) Función

Construye y devuelve una lista, siendo cada uno de sus elementos generados por *expr*.

La llamada **makelist** (*expr*, *i*, *i_0*, *i_1*) devuelve una lista cuyo *j*-ésimo elemento es igual a **ev** (*expr*, *i=j*), tomando *j* los valores enteros entre *i_0* y *i_1*.

La llamada **makelist** (*expr*, *x*, *list*) devuelve una lista cuyo *j*-ésimo elemento es igual a **ev** (*expr*, *x=list[j]*), tomando *j* los valores enteros entre 1 through **length** (*list*).

Ejemplos:

```
(%i1) makelist(concat(x,i),i,1,6);
(%o1) [x1, x2, x3, x4, x5, x6]
(%i2) makelist(x=y,y,[a,b,c]);
(%o2) [x = a, x = b, x = c]
```

member (*expr_1*, *expr_2*) 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);
(%o6)          1   1   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.

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]
```

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.

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]
```

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.

38 Conjuntos

38.1 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 objetos 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.

38.1.1 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)
(%i3) set(a, set(b));
(%o3)
(%i4) set(a, [b]);
(%o4)
(%i5) {};
(%o5)
(%i6) {a, b, a};
(%o6)
(%i7) {a, {b}};
(%o7)
(%i8) {a, [b]};
(%o8)
```

Dos elementos candidatos a formar parte de un conjunto, `x` e `y`, son redundantes, esto es, se consideran el mismo elemento a efectos de construir 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;
(%o1)
(%i2) y: a/c + b/c;
```

$$\begin{array}{r} \text{b} \quad \text{a} \\ - \quad + \quad - \\ \text{c} \quad \text{c} \end{array}$$

```

(%o2)      b   a
           - + -
           c   c
(%i3) z: (a + b)/c;
(%o3)      b + a
           -----
                           c
(%i4) is (x = y);
(%o4)      true
(%i5) is (y = z);
(%o5)      false
(%i6) is (equal (y, z));
(%o6)      true
(%i7) y - z;
(%o7)      b + a   b   a
           - ----- + - + -
                           c       c   c
(%i8) ratsimp (%);
(%o8)      0
(%i9) {x, y, z};
(%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};
(%o1)      {(x - 1) (x + 1), x^2 - 1}

```

Para reducir este conjunto a otro unitario, aplicar **rat** a cada elemento del conjunto:

```

(%i1) {(x - 1)*(x + 1), x^2 - 1};
(%o1)      {(x - 1) (x + 1), x^2 - 1}
(%i2) map (rat, %);
(%o2)/R/      {x^2 - 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};
(%o1)      {1, sin^2(x) + cos^2(x)}

```

```
(%i2) map (trigsimp, %);
(%o2) {1}
```

Se entiende que un conjunto está simplificado cuando entre sus elementos no hay redundancias y se hayan ordenados. 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.

38.1.2 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.

38.1.3 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`.

38.1.4 Autores

Stavros Macrakis de Cambridge, Massachusetts y Barton Willis de la University of Nebraska at Kearney (UNK).

38.2 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 $[x_1, \dots, x_n]$, siendo x_1, \dots, x_n elementos de los conjuntos b_1, \dots, 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]));
(%o3)           [ [ divisors(a)  divisors(b) ]
(%o3)           [ [ ] ]
(%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]
```

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},
(%o1)           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*)
every (*f*, *L*₁, ..., *L*_{*n*})

Función
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*)
extremal_subset (*s, f, min*)

Función
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 $[a_1, \dots, a_m]$ es una partición de un entero no negativo *n* si (1) cada a_i es entero no nulo y (2) $a_1 + \dots + a_m = n$. 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 (*x, y*)

Función

Es la función delta de Kronecker.

La función **kron_delta** devuelve 1 cuando *x* e *y* son idénticos o equivalentes, devolviendo 0 si *x* e *y* no son idénticos. Cuando no está claro si ambas expresiones son equivalentes, **kron_delta** devuelve una forma nominal. Si la diferencia *x - y* es un número decimal en coma flotante, **kron_delta** dará como resultado una expresión nominal, aún cuando *x* e *y* sean aparentemente equivalentes.

Concretando, **kron_delta**(*x, y*) devuelve 1 si **is(x = y)** retorna **true**; **kron_delta** también devuelve 1 si **sign(abs(x - y))** retorna **zero** y *x - y* no es un decimal en coma flotante (ni ordinario ni *bigfloat*); **kron_delta** devuelve 0 si **sign(abs(x - y))** retorna **pos**.

Cuando **sign(abs(x - y))** es cualquier otra cosa diferente a **pos** o **zero**, o si es **zero** siendo *x - y* un decimal en coma flotante, entonces **kron_delta** devuelve una expresión nominal.

La función **kron_delta** está declarada como simétrica, esto es, **kron_delta**(*x, y*) es igual a **kron_delta**(*y, x*).

Ejemplos:

Si los argumentos de **kron_delta** son idénticos, **kron_delta** devuelve 1,

```
(%i1) kron_delta (a, a);
(%o1) 1
(%i2) kron_delta (x^2 - y^2, x^2 - y^2);
(%o2) 1
(%i3) float (kron_delta (1/10, 0.1));
(%o3) 1
```

Si los argumentos de **kron_delta** son equivalentes y su diferencia no es un decimal en coma flotante, **kron_delta** devuelve 1,

```
(%i1) assume (equal (x, y));
(%o1) [equal(x, y)]
(%i2) kron_delta (x, y);
```

```
(%o2) 1
```

Si los argumentos de `kron_delta` no son equivalentes, `kron_delta` devuelve 0,

```
(%i1) kron_delta (a + 1, a);
(%o1) 0
(%i2) assume (a > b)$
(%i3) kron_delta (a, b);
(%o3) 0
(%i4) kron_delta (1/5, 0.7);
(%o4) 0
```

Si no es posible determinar la equivalencia de los argumentos de `kron_delta`, ésta devuelve una forma nominal,

```
(%i1) kron_delta (a, b);
(%o1) kron_delta(a, b)
(%i2) assume(x >= y)$
(%i3) kron_delta (x, y);
(%o3) kron_delta(x, y)
```

Si los argumentos de `kron_delta` son equivalentes, pero su diferencia es un decimal en coma flotante, `kron_delta` devuelve forma nominal,

```
(%i1) 1/4 - 0.25;
(%o1) 0.0
(%i2) 1/10 - 0.1;
(%o2) 0.0
(%i3) 0.25 - 0.25b0;
Warning: Float to bigfloat conversion of 0.25
(%o3) 0.0b0
(%i4) kron_delta (1/4, 0.25);
(%o4) kron_delta(-, 0.25)
        1
        4
(%i5) kron_delta (1/10, 0.1);
(%o5) kron_delta(--, 0.1)
        1
        10
(%i6) kron_delta (0.25, 0.25b0);
Warning: Float to bigfloat conversion of 0.25
(%o6) kron_delta(0.25, 2.5b-1)
```

La función `kron_delta` es simétrica.

```
(%i1) kron_delta (x, y);
(%o1) kron_delta(x, y)
(%i2) kron_delta (y, x);
(%o2) kron_delta(x, y)
(%i3) kron_delta (x, y) - kron_delta (y, x);
(%o3) 0
(%i4) is (equal (kron_delta (x, y), kron_delta (y, x)));
(%o4) true
(%i5) is (kron_delta (x, y) = kron_delta (y, x));
(%o5) true
```

listify (a)

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 $F(\dots F(F(s_1, s_2), s_3), \dots s_n)$. 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]));  
(%o8)           [ - 1  0 ]  
                  [         ]  
                  [ - 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 $a_1 + \dots + a_n$ objetos diferentes en n cajas con a_k elementos en la k -ésima caja. En general, **multinomial_coeff (a_1, ..., a_n)** calcula $(a_1 + \dots + a_n)!/(a_1! \dots a_n!)$.

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);
(%o1) 
$$\frac{(x + 3)!}{2 x!}$$

(%i2) minfactorial (%);
(%o2) 
$$\frac{(x + 1)(x + 2)(x + 3)}{2}$$

(%i3) multinomial_coeff (-6, 2);
(%o3) 
$$\frac{(-4)!}{2(-6)!}$$

(%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, \dots, k_m tales que $n = k_1 + \dots + k_m$.

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 $2^{\text{cardinality}(a)}$ 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 $F(s_{-1}, \dots, F(s_{-n+2}, F(s_{-n+1}, s_n)))$. 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
          ---
          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, \dots, 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, \dots, L_n) devuelve **true** si $f(x_1, \dots, x_n)$ devuelve también **true** para al menos un x_1, \dots, x_n de L_1, \dots, L_n , respectivamente; puede ser que **some** no evalúe f para todos las combinaciones x_1, \dots, 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, \dots, 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. $\text{stirling1}(0, n) = \text{krondelta}(0, n)$ (Ref. [1])
2. $\text{stirling1}(n, n) = 1$ (Ref. [1])
3. $\text{stirling1}(n, n - 1) = \text{binomial}(n, 2)$ (Ref. [1])
4. $\text{stirling1}(n + 1, 0) = 0$ (Ref. [1])
5. $\text{stirling1}(n + 1, 1) = n!$ (Ref. [1])

6. $\text{stirling1}(n + 1, 2) = 2^n - 1$ (Ref. [1])

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 aplica algunas identidades a **stirling1**,

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling1 (n + 1, n);
(%o3)          
$$\frac{n(n+1)}{2}$$

(%i4) stirling1 (n + 1, 1);
(%o4)          
$$\frac{n!}{n!}$$

```

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. $\text{stirling2}(0, n) = \text{krondelta}(0, n)$ (Ref. [1])
2. $\text{stirling2}(n, n) = 1$ (Ref. [1])
3. $\text{stirling2}(n, n - 1) = \text{binomial}(n, 2)$ (Ref. [1])
4. $\text{stirling2}(n + 1, 1) = 1$ (Ref. [1])
5. $\text{stirling2}(n + 1, 2) = 2^n - 1$ (Ref. [1])
6. $\text{stirling2}(n, 0) = \text{krondelta}(n, 0)$ (Ref. [2])
7. $\text{stirling2}(n, m) = 0$ when $m > n$ (Ref. [2])
8. $\text{stirling2}(n, m) = \sum((-1)^{m-k} \text{binomial}(mk) k^n, i, 1, m) / m!$ 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 aplica algunas identidades a **stirling2**,

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling2 (n + 9, n + 8);
(%o3)          
$$\frac{(n + 8)(n + 9)}{2}$$

(%i4) stirling2 (n + 1, 2);
(%o4)          
$$\frac{n}{2 - 1}$$

```

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 returned 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 el conjunto de elementos que pertenecen a un único conjunto de los a_k .

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, z}
(%i8) symmdifference ({}, S_1, S_2, S_3);
(%o8)                                {1, 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 siguiente: Aplicar F a pares sucesivos de elementos para formar una nueva lista $[F(s_1, s_2), F(s_3, s_4), \dots]$, 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 sola 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]
```

39 Definición de Funciones

39.1 Introducción a la definición de funciones

39.2 Funciones

39.2.1 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), 1)
```

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 función 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`.

39.2.2 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 `disfun` 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.

39.3 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;
(%o1)           g(x) ::=  $\frac{x}{99}$ 
(%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;
(%o1)          g(x) ::= -- x
                           99
(%i2) h (x) ::= buildq ([x], g (x - a));
(%o2)          h(x) ::= buildq([x], g(x - a))
(%i3) a: 1234;
(%o3)                               1234
(%i4) 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 admite 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<>
```

39.4 Funciones y variables para la definición de funciones

apply (*F*, [*x₁*, ..., *x_n*])

Función

Construye y evalúa la expresión *F(arg₁, ..., 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;
(%o1)           F(x) :=  $\frac{x}{1729}$ 
(%i2) fname : F;
(%o2)           F
(%i3) dispfun (F);
```

```
(%t3)          x
               F(x) := ----
                           1729

(%o3)          [%t3]
(%i4) dispfun (fname);
fname is not the name of a user function.
-- an error. Quitting. To debug this try debugmode(true);
(%i5) apply (dispfun, [fname]);
(%t5)          x
               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, \dots, v_m], $expr_1, \dots, expr_n$)

Función
Función

block ($expr_1, \dots, expr_n$)

La función `block` evalúa $expr_1, \dots, expr_n$ 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, \dots, v_m son locales en el bloque; éstas se distinguen 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, \dots, v_m se considera global.

La función `block` guarda los valores actuales de las variables v_1, \dots, 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, \dots, v_m)` dentro de un bloque almacena las propiedades asociadas a los símbolos v_1, \dots, 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 utilizar `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

Evaluá *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(1) := catch (map (''% , 1))$  
(%i3) g ([1, 2, 3, 7]);  
(%o3) [f(1), f(2), f(3), f(7)]  
(%i4) g ([1, 2, -3, 7]);  
(%o4) - 3
```

La función `g` devuelve las imágenes por `f` de todos los elementos de la lista `1` 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`.

<code>compile (f_1, ..., f_n)</code>	Función
<code>compile (functions)</code>	Función
<code>compile (all)</code>	Función

Traduce las funciones de Maxima f_1, \dots, 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.

<code>define (f(x_1, ..., x_n), expr)</code>	Función
<code>define (f[x_1, ..., x_n], expr)</code>	Función
<code>define (funmake (f, [x_1, ..., x_n]), expr)</code>	Función
<code>define (arraymake (f, [x_1, ..., x_n]), expr)</code>	Función
<code>define (ev (expr_1), expr_2)</code>	Función

Define una función de nombre f con argumentos x_1, \dots, 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 $x_1, \dots, x_{(n - 1)}$, 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 $f(x_1, \dots, x_n)$ o $f[x_1, \dots, x_n]$, 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.");
(%i3) 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 (*f₁, ..., f_n*)
dispfun (*all*)

Función
Función

Muestra la definición de las funciones de usuario *f₁, ..., 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);
(%o1)                                m(x, y) := x
(%i2) f(x, y) := x^(-y);
(%o2)                                f(x, y) := x
(%i3) g[x, y] := x^(-y);
(%o3)                                g      := x
                                         x, y
(%i4) h[x](y) := x^(-y);
(%o4)                                h(y) := x
                                         x
(%i5) i[8](y) := 8^(-y);
(%o5)                                i(y) := 8
                                         8
(%i6) dispfun (m, f, g, h, h[5], h[10], i[8]);
(%t6)                                m(x, y) := x
                                         - y
                                         x
(%t7)                                f(x, y) := x
                                         - y
                                         x
(%t8)                                g      := x
                                         x, y
                                         - y
                                         x
(%t9)                                h(y) := x
                                         x
(%t10)                               h(y) := -- 1
                                         5      y
                                         5
(%t11)                               h(y) := --- 1
                                         10     y
                                         10
(%t12)                               i(y) := 8
                                         8
                                         - y
(%o12)      [%t6, %t7, %t8, %t9, %t10, %t11, %t12]
(%i12)  ''%;
```

```
(%o12) [m(x, y) := x-y, f(x, y) := x-y, g(x, y) := x-y,
      h(y) := x-y, h(y) := 51, h(y) := y10, i(y) := 8-y ]
           x          5          y          10          y          8
                           5          10
```

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 subindividuadas, 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;
(%o2)                               F_2(x, y) := -x
                                         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;
(%o5)                               G_2 := -x
                                         y
(%i6) define (G_3 [x], sqrt (x));
(%o6)                               G_3 := sqrt(x)
                                         x
(%i7) H_1 [x] (y) := x^y;
(%o7)                               H_1 (y) := x^y
                                         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 $F(arg_1, \dots, arg_n)$. 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 $F(\dots)$, 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;
(%o1) F(x, y) := y^2 - x^2
(%i2) funmake (F, [a + 1, b + 1]);
(%o2) F(a + 1, b + 1)
(%i3) '';
(%o3) (b + 1)^2 - (a + 1)^2
```

La función `funmake` aplicada a una macro.

```
(%i1) G (x) ::= (x - 1)/2;
(%o1) G(x) ::=  $\frac{x - 1}{2}$ 
(%i2) funmake (G, [u]);
(%o2) G(u)
(%i3) '';
(%o3)  $\frac{u - 1}{2}$ 
```

La función `funmake` aplicada a una función subindicada.

```
(%i1) H [a] (x) := (x - 1)^a;
(%o1)                                H (x) := (x - 1)
(%i2) funmake (H [n], [%e]);
(%o2)          lambda([x], (x - 1)^(%e))
(%i3) '';
(%o3)                                (%e - 1)
(%i4) funmake ('(H [n]), [%e]);
(%o4)          H (%e)
(%i5) '';
(%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 returnedo.

```
(%i1) det(a,b,c) := b^2 -4*a*c;
(%o1)          det(a, b, c) := b^2 - 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 returnedo de **funmake**.

```
(%i1) funmake (sin, [%pi / 2]);
(%o1)          1
```

lambda ([x₁, ..., x_m], expr₁, ..., expr_n)
lambda ([[L]], expr₁, ..., expr_n)
lambda ([x₁, ..., x_m, L], expr₁, ..., expr_n)

Función
Function
Function

Define y devuelve una expresión lambda (es decir, una función anónima). La función puede tener argumentos x₁, ..., 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)
(%i2) f(a);
(%o2)
```

- Una expresión `lambda` puede aparecer en algunos contextos en los que sea necesario un nombre de función.

```
(%i3) lambda ([x], x^2) (a);
(%o3)
(%i4) apply (lambda ([x], x^2), [a]);
(%o4)
(%i5) map (lambda ([x], x^2), [a, b, c, d, e]);
(%o5)
```

- 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)
(%i9) b: %gamma$
(%i10) g(1/2);
```

```
(%o10) %gamma
-----
2
(%i11) g2: lambda ([a], a*'b);
(%o11)           lambda([a], a %gamma)
(%i12) b: %e$ 
(%i13) g2(1/2);
(%o13) %gamma
-----
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));
(%o14)      lambda([a, b], h2 : lambda([a], a b), h2(-))
2
(%i15) h(%pi, %gamma);
(%o15) %gamma
-----
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);
(%o19)           lambda([x], -)
2
(%i20) i2(1/2)(%pi);
(%o20) %pi
-----
```

- 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 (*v₁*, ..., *v_n*)

Función

La declaración **local**(*v₁*, ..., *v_m*) dentro de un bloque almacena las propiedades asociadas a los símbolos *v₁*, ..., *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 sola 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; cualesquier 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);
(%o1)          f(x) := -----
                           h(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
(%o5)          a b - 99
                   -----
                   a b + 99
(%i6) dispfun (f);
(%t6)          f(x) := -----
                           h(x)
                                 g(x)
(%o6)                               done
(%i7) f (a * b);
x - 99 is equal to x
x + 99 is equal to x
(%o7)          a b - 99
                   -----
                   a b + 99
```

Si `macroexpansion` vale `expand`, una función macro tan solo es llamada una vez.

```
(%i1) f (x) := h (x) / g (x);
(%o1)          f(x) := -----
                           h(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);
(%o1)                                f(x) :=  $\frac{h(x)}{g(x)}$ 
(%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),
(%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),
(%i4) macroexpansion: displace;
(%o4)                                displace
(%i5) f (a * b);
x - 99 is equal to x
x + 99 is equal to x
(%o5) 
$$\frac{a b - 99}{a b + 99}$$

(%i6) dispfun (f);
(%t6) f(x) := 
$$\frac{x - 99}{x + 99}$$

(%o6) done
(%i7) f (a * b);

```

```
(%o7)      a b - 99
-----  
           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.**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 (<i>f_1, ..., f_n</i>)	Función
translate (<i>functions</i>)	Función
translate (<i>all</i>)	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 rápidamente 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 (<i>nombre_fichero_maxima</i>)	Función
translate_file (<i>nombre_fichero_maxima, nombre_fichero_lisp</i>)	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 fichero 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`, `tr_warn_undefined_variable`, y `tr_windy`.

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: <code>false</code>	
Si <code>tr_file_tty_messagesp</code> vale <code>true</code> , los mensajes generados por <code>translate_file</code> durante la traducción de un fichero se muestran en la consola y se insertan en el fichero UNLISP. Si vale <code>false</code> , los mensajes sobre la traducción del fichero sólo se incorporan al fichero UNLISP.	
tr_float_can_branch_complex	Variable opcional
Valor por defecto: <code>true</code>	
Le dice al traductor de Maxima a Lisp que las funciones <code>acos</code> , <code>asin</code> , <code>asec</code> y <code>acsc</code> pueden devolver valores complejos.	
tr_function_call_default	Variable opcional
Valor por defecto: <code>general</code>	
El valor <code>false</code> significa llama a <code>meval</code> , <code>expr</code> significa que Lisp asignó los argumentos de la función, <code>general</code> , el valor por defecto, devuelve código apropiado para <code>mexprs</code> y <code>mlexprs</code> pero no para <code>macros</code> . La opción <code>general</code> asegura que las asignaciones de las variables son correctas en el código compilado. En modo <code>general</code> , cuando se traduce <code>F(X)</code> , si <code>F</code> es una variable con valor, entonces se entiende que se quiere calcular <code>apply (f, [x])</code> , 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: <code>false</code>	
Si <code>tr_numer</code> vale <code>true</code> se utilizan las propiedades numéricas en aquellos átomos que las posean, como en <code>%pi</code> .	
tr_optimize_max_loop	Variable opcional
Valor por defecto: 100	
El valor de <code>tr_optimize_max_loop</code> 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: <code>false</code>	
Si <code>tr_semicompile</code> vale <code>true</code> , las salidas de <code>translate_file</code> y <code>compfile</code> 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: <code>true</code>	
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: <code>compfile</code>	
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: <code>compfile</code>	
Devuelve un aviso si la función <code>meval</code> es llamada. Si <code>meval</code> es invocada, es señal de la presencia de problemas en la traducción.	
tr_warn_mode	Variable opcional
Valor por defecto: <code>all</code>	
Devuelve un aviso cuando a las variables se les asignan valores incompatibles con su modo.	
tr_warn_undeclared	Variable opcional
Valor por defecto: <code>compile</code>	
Determina cuando enviar mensajes sobre variables no declaradas.	
tr_warn_undefined_variable	Variable opcional
Valor por defecto: <code>all</code>	
Devuelve un aviso cuando se detectan variables globales no definidas.	
tr_windy	Variable opcional
Valor por defecto: <code>true</code>	
Genera comentarios de ayuda y consejos sobre programación.	

compile_file (<i>nombre_fich</i>)	Función
compile_file (<i>nombre_fich, nombre_fich_compilado</i>)	Función
compile_file (<i>nombre_fich, nombre_fich_compilado, nombre_fich_lisp</i>)	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 (<i>f_1, f_2, ...</i>)	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.

40 Programación

40.1 Introducción a la programación

Maxima dispone de los bucles `do` para hacer iteraciones, así como estructuras más primitivas del estilo de `go`.

40.2 Funciones y variables para la programación

<code>backtrace ()</code>	Función
<code>backtrace (n)</code>	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

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

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)
(%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
      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)  

(%t3) %pi  

      ---  

      4  

(%i4) ev(%t3,numer);  

(%o4) 0.78539816
```

errcatch (expr_1, ..., expr_n)

Función

Evaluá 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*)
error

Función

Variable del sistema

Calcula y devuelve *expr_1, ..., expr_n*, enviando posteriormente una señal 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.

errormsg ()

Función

Reimprime el mensaje de error más reciente. La variable `error` guarda el mensaje y `errormsg` lo formatea e imprime.

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	<code><=</code>	operador relacional infijo
igualdad (sintáctica)	=	operador relacional infijo
negación de =	#	operador relacional infijo
igualdad (por valor)	<code>equal</code>	operador relacional infijo
negación de equal	<code>notequal</code>	operador relacional infijo
mayor o igual que	<code>>=</code>	operador relacional infijo
mayor que	>	operador relacional infijo
y	<code>and</code>	operador lógico infijo
o	<code>or</code>	operador lógico infijo
no	<code>not</code>	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));
(%o2)          1      1      1
           ----- - ----- + ----- + x
           x + 2    x + 1        2
                                         (x + 1)
(%i3) ratsimp( x/(x^2+x)+(y^2+y)/y);
(%o3)          1
           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`, `map1` 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 o 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);  

(%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));  

(%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);  

(%o5) f(f(f(a) f(x)) + f(b))  

      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, $x[i_1] \dots [i_m]$ tiene la misma estructura que la segunda lista, lista anidada o matriz, $x[i_1] \dots [i_m][j_1] \dots [j_n]$ tiene la misma estructura que la tercera lista, lista anidada o matriz, y así sucesivamente, siendo m, n, \dots 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, mientras 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]));
(%o2) [
      [ [ F(a, 1) F(a, 2) ] [ F(b, 1) F(b, 2) ]
        [ [ ] [ ] ]
        [ [ F(a, 3) F(a, 4) ] [ F(b, 3) F(b, 4) ]
          [ [ ] [ ]
            [ [ 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 devuelto 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]);
(%o1) [ 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 ]
              [ [ xx + -- yy + -- ], [ xx + -- yy + -- ] ]
              [ [      11      11 ] [      22      22 ] ]
(%i6) xx_2 : xx_0 [1][1] [1];
              [      a      a ]
              [ xx + -- yy + -- ]
              [ [      11      11 ] ]
(%i7) xx_3 : xx_0 [1][1] [1] [1];
              a
              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
```


41 Depurado

41.1 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 compilación 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`.

41.2 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.

41.3 Funciones y variables para depurado

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 entonces 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 (<i>f_1, ..., f_n</i>)	Función
timer (<i>all</i>)	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 (<i>f_1, ..., f_n</i>)	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 (<i>f_1, ..., f_n</i>)	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 *f₁*, ..., *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 rastreadas.

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 $\rightarrow \text{true}$ tanto a la entrada como a la salida de la función.
- **errorcatch:** Detecta errores, otorgando la posibilidad de marcar un error, reiniciar 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, \dots, f_n)

Función

untrace ()

Función

Dadas las funciones f_1, \dots, 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.

42 augmented_lagrangian

42.1 Funciones y variables para augmented_lagrangian

augmented_lagrangian_method (<i>FOM</i> , <i>xx</i> , <i>C</i> , <i>yy</i>)	Función
augmented_lagrangian_method (<i>FOM</i> , <i>xx</i> , <i>C</i> , <i>yy</i> , <i>optional_args</i>)	Función
augmented_lagrangian_method ([<i>FOM</i> , <i>grad</i>], <i>xx</i> , <i>C</i> , <i>yy</i>)	Función
augmented_lagrangian_method ([<i>FOM</i> , <i>grad</i>], <i>xx</i> , <i>C</i> , <i>yy</i> , <i>optional_args</i>)	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.
lbfsgs_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 `lbfsgs`.

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 (lbfsgs);
(%o1)      /maxima/share/lbfsgs/lbfsgs.mac
(%i2) load (augmented_lagrangian);
(%o2)
```

```
/maxima/share/contrib/augmented_lagrangian.mac
(%i3) FOM: x^2 + 2*y^2;
(%o3)
(%o4) xx: [x, y];
(%o4)
(%i5) C: [x + y - 1];
(%o5)
(%i6) yy: [1, 1];
(%o6)
(%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;
(%o3)
(%i4) FOM: x^2 + 2*y^2;
(%o4)
(%i5) xx: [x, y];
(%o5)
(%i6) grad : [2*x, 4*y];
(%o6)
(%i7) C: [x + y - 1];
(%o7)
(%i8) yy: [1, 1];
(%o8)
(%i9) augmented_lagrangian_method ([FOM, grad], xx, C, yy, iprint = [-1, 0]);
(%o9) [[x = 0.666659841080025, y = .3333402724554462],
%lambda = [- 1.333337940892543]]
```

43 bode

43.1 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`.

44 contrib_ode

44.1 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 syntaxis 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;

(%o2)

$$x \frac{dy^2}{dx^2} - (x y + 1) \frac{dy}{dx} + y = 0$$


(%i3) contrib_ode(eqn,y,x);

(%o3)

$$[y = \log(x) + %c, y = %c e^{\frac{x}{x}}]$$

(%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;

(%o2)

$$\frac{dy}{dx}^2 + x \frac{dy}{dx} - y = 0$$


(%i3) contrib_ode(eqn,y,x);

(%o3)

$$[y = \%c x^2 + \%c, \quad y = -\frac{x^2}{4}]$$


(%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;

(%o2)

$$\frac{dy}{dx}^2 = (y + x)^2$$


(%i3) contrib_ode(eqn,y,x);

(%o3)

$$[[x = \%c - \text{atan}(\sqrt(%t)), \quad y = -x - \sqrt(%t)], \\ [x = \text{atan}(\sqrt(%t)) + \%c, \quad y = \sqrt(%t) - x]]$$


(%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)

$$[x > 0, \quad y > 0]$$


(%i3) eqn:x*'diff(y,x)-x*sqrt(y^2+x^2)-y;

(%o3)

$$x \frac{dy}{dx}^2 - x \sqrt{y^2 + x^2} - y$$


(%i4) contrib_ode(eqn,y,x);

(%o4)

$$[x - \frac{\text{asinh}(-\frac{y}{x})}{x} = \%c]$$

```

```
(%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;

(%o2)

$$\frac{2 \frac{dy}{dx}}{x} = c x^n y^2 + b x^n + a$$


(%i3) contrib_ode(eqn,y,x);


$$\frac{\frac{d\%u}{dx}}{\frac{\%u^2}{x^2}} = \frac{(b x^{n-2} + a)}{x^2} + \frac{\frac{d\%u}{dx}}{x^2}$$


(%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)`.

44.2 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` resuelve 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;

(%o2)          2
              d y
              --- + (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)];

(%o3) [y = bessel_y(-, 1 2 (a x + b)
           3           3 a) %k2 sqrt(a x + b)
           + bessel_j(-, 1 2 (a x + b)
           3           3 a) %k1 sqrt(a x + b)]
(%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 $x(1-x) \operatorname{diff}(y,x,2) + [c-(a+b+1)x \operatorname{diff}(y,x) - aby = 0$ (A&S 15.5.1). El único uso que se hace de estas funciones es en las soluciones de EDOs que devuelven <code>odelin</code> y <code>contrib_ode</code> . La definición y utilización de estas funciones puede cambiar en futuras distribuciones de Maxima.	
Véanse también <code>gauss_b</code> , <code>dgauss_a</code> y <code>gauss_b</code> .	
gauss_b (a, b, c, x)	Función
Véase también <code>gauss_a</code> .	
dgauss_a (a, b, c, x)	Función
The derivative with respect to x of <code>gauss_a(a,b,c,x)</code> .	
dgauss_b (a, b, c, x)	Función
Derivada de <code>gauss_b(a,b,c,x)</code> respecto de x.	
kummer_m (a, b, x)	Función
Función M de Kummer, tal como la definen Abramowitz y Stegun, <i>Handbook of Mathematical Functions</i> , Sección 13.1.2.	
El único uso que se hace de esta función es en las soluciones de EDOs que devuelven <code>odelin</code> y <code>contrib_ode</code> . La definición y utilización de estas funciones puede cambiar en futuras distribuciones de Maxima.	
Véanse también <code>kummer_u</code> , <code>dkummer_m</code> y <code>dkummer_u</code> .	
kummer_u (a, b, x)	Función
Función U de Kummer, tal como la definen Abramowitz y Stegun, <i>Handbook of Mathematical Functions</i> , Sección 13.1.3.	
Véase también <code>kummer_m</code> .	
dkummer_m (a, b, x)	Función
Derivada de <code>kummer_m(a,b,x)</code> respecto de x.	
dkummer_u (a, b, x)	Función
Derivada de <code>kummer_u(a,b,x)</code> respecto de x.	

44.3 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 errores, otros devuelven respuestas muy complejas.
- Hacer más pruebas.

44.4 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, incluidas 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.

44.5 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. Koloknikov, First Order ODEs, Symmetries and Linear Transformations, European Journal of Applied Mathematics, Vol. 14, No. 2, pp. 231-246 (2003). (<http://arxiv.org/abs/math-ph/0007023>) (http://lie.uwaterloo.ca/papers/ode_iv.pdf)

8. G. W. Bluman, S. C. Anco, Symmetry and Integration Methods for Differential Equations, Springer, (2002)
9. M Bronstein, S Lafaille, Solutions of linear ordinary differential equations in terms of special functions, Proceedings of ISSAC 2002, Lille, ACM Press, 23-28. (<http://www-sop.inria.fr/cafe/Manuel.Bronstein/publications/issac2002.pdf>)

45 descriptive

45.1 Introducción a descriptive

El paquete **descriptive** contiene funciones para realizar cálculos 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 '*mario AT edu DOT xunta DOT es*'.

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 univariante */ mean ([a, b, c]);
          c + b + a
(%o2)
          -----
          3
(%i3) matrix ([a, b], [c, d], [e, f]);
          [ a   b ]
          [       ]
          [ 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 ]
```

```
(%o1) [ [ ] [ 14.71 16.88 ] [ ] [ 18.5 16.88 ] [ ] [ ] [ 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]);
(%o2)
[ 1   3 ]
[       ]
[ b   5 ]
```

45.2 Funciones y variables para el tratamiento de datos

continuous_freq (*list*)

Función

continuous_freq (*list, m*)

Función

El argumento de **continuous_freq** debe ser una lista de números, los cuales serán luego agrupados en intervalos y hecho el recuento de cuántos hay en cada grupo. Opcionalmente, la función **continuous_freq** admite un segundo argumento para indicar el número de clases a considerar, siendo 10 su valor por defecto.

```
(%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]]
```

La primera lista contiene los extremos de los intervalos y la segunda los resultados de los recuentos: hay 16 dígitos dentro del intervalo [0, 1.8], esto es ceros y unos, 24 dígitos en (1.8, 3.6], es decir doses y treses, y así sucesivamente.

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.

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 argumentosopcionales 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"))$
```

$$\begin{aligned} (%i3) \text{subsample} (s2, \lambda([v], v[1] > 18)); \\ & [19.38 \quad 15.37 \quad 15.12 \quad 23.09 \quad 25.25] \\ & [\quad \quad \quad \quad \quad \quad] \\ & [18.29 \quad 18.66 \quad 19.08 \quad 26.08 \quad 27.63] \\ (%o3) & [\quad \quad \quad \quad \quad \quad] \\ & [20.25 \quad 21.46 \quad 19.95 \quad 27.71 \quad 23.38] \\ & [\quad \quad \quad \quad \quad \quad] \\ & [18.79 \quad 18.96 \quad 14.46 \quad 26.38 \quad 21.84] \end{aligned}$$

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"))$
```

$$\begin{aligned} (%i3) g(x) := x[1] \geq 16 \text{ and } x[4] < 25 \$ \\ (%i4) \text{subsample} (s2, g, 1, 2, 5); \\ & [19.38 \quad 15.37 \quad 25.25] \\ & [\quad \quad \quad \quad \quad \quad] \\ & [17.33 \quad 14.67 \quad 19.58] \\ (%o4) & [\quad \quad \quad \quad \quad \quad] \\ & [16.92 \quad 13.21 \quad 21.21] \\ & [\quad \quad \quad \quad \quad \quad] \\ & [17.25 \quad 18.46 \quad 23.87] \end{aligned}$$

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 años,

```
(%i1) load (descriptive)$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
```

$$\begin{aligned} (%i3) h(u) := u[1] = B \text{ and } u[2] > 38 \$ \\ (%i4) \text{subsample} (s3, h); \\ & [B \quad 39 \quad 28.0 \quad 102.3 \quad 17.1 \quad 146] \\ & [\quad \quad \quad \quad \quad \quad \quad] \\ & [B \quad 39 \quad 21.0 \quad 92.4 \quad 10.3 \quad 197] \\ & [\quad \quad \quad \quad \quad \quad \quad] \\ & [B \quad 39 \quad 23.0 \quad 111.5 \quad 10.0 \quad 133] \\ & [\quad \quad \quad \quad \quad \quad \quad] \\ & [B \quad 39 \quad 26.0 \quad 92.6 \quad 12.3 \quad 196] \\ (%o4) & [\quad \quad \quad \quad \quad \quad \quad] \\ & [B \quad 39 \quad 25.0 \quad 98.7 \quad 10.0 \quad 174] \\ & [\quad \quad \quad \quad \quad \quad \quad] \\ & [B \quad 39 \quad 21.0 \quad 93.2 \quad 5.9 \quad 181] \\ & [\quad \quad \quad \quad \quad \quad \quad] \\ & [B \quad 39 \quad 18.0 \quad 95.0 \quad 11.3 \quad 66] \end{aligned}$$

```
[      ]
[ 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
```

45.3 Funciones y variables de valores 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*)
std (*matrix*)

Función
 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*)
std1 (*matrix*)

Función
 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*)

noncentral_moment (*matrix*, *k*)

Función
 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*)
central_moment (*matrix, k*)

Función
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*)
cv (*matrix*)

Función
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`.

mini (*list*)
mini (*matrix*)

Función
Función

Es el valor mínimo de la muestra *list*,

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) mini (s1);
(%o3) 0
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) mini (s2);
(%o5) [0.58, 0.5, 2.67, 5.25, 5.17]
```

Véase también `maxi`.

maxi (*list*)

Función

maxi (*matrix*)

Función

Es el valor máximo de la muestra *list*,

```
(%i1) load (descriptive)$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) maxi (s1);
(%o3) 9
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) maxi (s2);
(%o5) [20.25, 21.46, 20.04, 29.63, 27.63]
```

Véase también `mini`.

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,
 $\text{quantile}(\text{list}, 3/4) - \text{quantile}(\text{list}, 1/4)$,

```
(%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, 8.72999999999999,
       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*)
median_deviation (*matrix*)

Función
Función

Es la desviación mediana, definida como

$$\frac{1}{n} \sum_{i=1}^n |x_i - \text{med}|$$

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*)
harmonic_mean (*matrix*)

Función
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*)
geometric_mean (*matrix*)

Función
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 (<i>list</i>)	Función
pearson_skewness (<i>matrix</i>)	Función

Es el coeficiente de asimetría de Pearson, definido como

$$\frac{3(\bar{x} - \text{med})}{s}$$

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 (<i>list</i>)	Función
quartile_skewness (<i>matrix</i>)	Función

Es el coeficiente de asimetría cuartílico, definido como

$$\frac{c_{\frac{3}{4}} - 2c_{\frac{1}{2}} + c_{\frac{1}{4}}}{c_{\frac{3}{4}} - c_{\frac{1}{4}}}$$

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**.

45.4 Funciones y variables de valores descriptivos multivariantes

cov (<i>matrix</i>)	Función
Es la matriz de covarianzas de una muestra multivariante, definida como	

$$S = \frac{1}{n} \sum_{j=1}^n (X_j - \bar{X}) (X_j - \bar{X})'$$

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

$$\frac{1}{n-1} \sum_{j=1}^n (X_j - \bar{X}) (X_j - \bar{X})'$$

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, logical_value)

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`.

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]
```

La función `global_variances` tiene un argumento lógico opcional: `global_variances(x,true)` indica a Maxima que x es la matriz de datos, calculando entonces lo mismo que `global_variances(x)`. Por otro lado, `global_variances(x,false)` significa que x no es la matriz de datos, sino la de covarianzas, evitando así recalcularla,

```
(%i1) load (descriptive)$
(%i2) s2 : read_matrix (file_search ("wind.data"))$ 
(%i3) s : cov1 (s2)$
(%i4) global_variances (s, false);
(%o4) [105.338342060606, 21.06766841212119, 12874.34690469686,
      113.4651792608502, 6.636590811800794, 2.576158149609762]
```

Véanse también `cov` y `cov1`.

cor (<i>matrix</i>)	Función
cor (<i>matrix, logical_value</i>)	Función

Es la matriz de correlaciones de la muestra multivariante.

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 ]
      [                   ]
      [   .8803515   .8735834     1.0     .7764065   .8323358 ]
      [                   ]
      [   .8239624   .6902622   .7764065     1.0     .7293848 ]
      [                   ]
      [   .7519506   0.782502   .8323358   .7293848     1.0 ]
```

La función `cor` tiene un argumento lógico opcional: `cor(x,true)` indica a Maxima que x es la matriz de datos, calculando entonces lo mismo que `cor(x)`. Por otro lado, `cor(x,false)` significa que x no es la matriz de datos, sino la de covarianzas, evitando así recalcularla,

```
(%i1) load (descriptive)$
(%i2) fpprintprec:7$
(%i3) s2 : read_matrix (file_search ("wind.data"))$ 
(%i4) s : cov1 (s2)$
(%i5) cor (s, false); /* this is faster */
[ 1.0   .8476339  .8803515  .8239624  .7519506 ]
[ ]
[ .8476339   1.0   .8735834  .6902622  0.782502 ]
[ ]
(%o5) [ .8803515   .8735834   1.0   .7764065  .8323358 ]
[ ]
[ .8239624   .6902622   .7764065   1.0   .7293848 ]
[ ]
[ .7519506   0.782502   .8323358   .7293848   1.0 ]
```

Véanse también `cov` y `cov1`.

list_correlations (*matrix*)
list_correlations (*matrix, logical_value*)

Función
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 ,

$$S_1^{-1} = (s^{ij})_{i,j=1,2,\dots,p}$$

- *multiple correlation vector*: $(R_1^2, R_2^2, \dots, R_p^2)$, 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

$$r_{ij.rest} = -\frac{s^{ij}}{\sqrt{s^{ii}s^{jj}}}$$

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 ]]
```

La función `list_correlations` tiene un argumento lógico opcional: `list_correlations(x,true)` indica a Maxima que `x` es la matriz de datos, calculando entonces lo mismo que `list_correlations(x)`. Por otro lado, `list_correlations(x,false)` significa que `x` no es la matriz de datos, sino la de covarianzas, evitando así recalcularla.

Véanse también `cov` y `cov1`.

45.5 Funciones y variables para gráficos estadísticos

<code>histogram (list)</code>	Función
<code>histogram (list, option_1, option_2, ...)</code>	Función
<code>histogram (one_column_matrix)</code>	Función
<code>histogram (one_column_matrix, option_1, option_2, ...)</code>	Función
<code>histogram (one_row_matrix)</code>	Función
<code>histogram (one_row_matrix, option_1, option_2, ...)</code>	Función

Esta función dibuja el histograma de una muestra constante. Los datos muestrales se deben almacenar en una lista de números o en una matriz unidimensional.

Opciones disponibles:

- Las definidas en el paquete `draw`. Véanse también `bars` y `barsplot`.
- `nclasses`: número de clases del histograma (por defecto, 10).

Véanse también `discrete_freq` y `continuous_freq` para recuentos, y `bars` y `barsplot` para representar gráficos.

Ejemplos:

Un histograma de ocho 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)$

scatterplot (list)                                         Función
scatterplot (list, option_1, option_2, ...)                 Función
scatterplot (matrix)                                       Función
scatterplot (matrix, option_1, option_2, ...)                Función
```

Dibuja diagramas de dispersión, tanto de muestras univariantes (*list*) como multivariantes (*matrix*).

Opciones disponibles:

- Las definidas en el paquete **draw**.
- *nclases*: número de clases del histograma (por defecto, 10).

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,
    terminal   = eps,
    eps_width  = 10,
    eps_height = 2)$
```

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))$
```

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,
  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**. Para las opciones relacionadas con los histogramas, véase **bars**.

Véase también **histogram**.

barsplot (*data1, data2, ..., option_1, option_2, ...*)

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:

- Las definidas en el paquete **draw**.
- **box_width**: ancho relativo de los rectángulos (3/4 por defecto). Este valor debe pertenecer al rango [0,1].
- **groups_gap**: un número positivo que representa la separación entre dos grupos consecutivos de barras. El valor por defecto es 1.
- **bars_colors**: una lista de colores para múltiples muestras. El valor por defecto es la lista vacía []. Cuando el número de muestras sea mayor que el de colores especificados, los colores adicionales necesarios se seleccionan aleatoriamente. Véase **color** para más información.
- **relative_frequencies**: si vale **false**, se utilizarán frecuencias absolutas; si vale **true**, las marcas del eje-y serán frecuencias relativas. El valor por defecto es **false**.
- **ordering**: los valores admitidos para esta opción son: **orderlessp** y **ordergreatp**, indicando cómo se deben ordenar los resultados muestrales sobre el eje-x. El valor por defecto es **orderlessp**.
- **sample_keys**: es una lista de cadenas de texto a usar como leyendas. Su valor por defecto es la lista vacía []. Cuando la lista tenga una longitud diferente de cero o del número de muestras, se devolverá un mensaje de error.

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],
    relative_frequencies = true,
    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)$
```

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) draw_compound : false $
(%i5) bp1 :
    barsplot(
        l1,
        box_width = 1,
        fill_density = 0.5,
        bars_colors = [blue],
        relative_frequencies = true)$
(%i6) bp2 :
    barsplot(
        l2,
        box_width = 1,
        fill_density = 0.5,
        bars_colors = [red],
        relative_frequencies = true)$
(%i7) draw(apply(gr2d,bp1),
    apply(gr2d,bp2)) $
```

Para las opciones relacionadas con los diagramas de barras, véase **bars** del paquete **draw**.

Véanse también las funciones **histogram** y **piechart**.

piechart (<i>list</i>)	Función
piechart (<i>list, option_1, option_2, ...</i>)	Función
piechart (<i>one_column_matrix</i>)	Función
piechart (<i>one_column_matrix, option_1, option_2, ...</i>)	Función
piechart (<i>one_row_matrix</i>)	Función
piechart (<i>one_row_matrix, option_1, option_2, ...</i>)	Función

Similar a **barsplot**, pero dibuja sectores en lugar de rectángulos.

Opciones disponibles:

- Las definidas en el paquete **draw**.
- *pie_center*: centro del diagrama (por defecto, [0,0]).
- *pie_radius*: radio del diagrama (por defecto, 1).

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],
      axis_top    = false,
      axis_right  = false,
      axis_left   = false,
      axis_bottom = false,
      xtics       = none,
      ytics       = none,
      title       = "Digit frequencies in pi")$
```

Véase también la función **barsplot**.

boxplot (<i>data</i>)	Función
boxplot (<i>data, option_1, option_2, ...</i>)	Función

Dibuja diagramas de cajas (box-and-whisker). 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.

Opciones disponibles:

- Las definidas en el paquete **draw**.
- *box_width*: ancho relativo de las cajas (por defecto, 3/4). Este valor debe estar en el rango [0,1].

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)$
```

46 diag

46.1 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)      [     ]
```

$$\begin{bmatrix} 0 & 3 \end{bmatrix}$$

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   0   2   0 ]  

      [                               ]  

      [ 0   0   0   0   0   0   0   2 ]  

      [                               ]  

      [ 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) $\begin{bmatrix} [-1, 1], [1, 3] \end{bmatrix}$

(%i4) M: ModeMatrix(a,%);

$$\begin{bmatrix} 1 & -1 & 1 & 1 \\ 9 & 1 & 0 & 0 \\ 13 & -1 & -1 & 0 \\ 9 & 17 & -1 & 1 \\ 9 & \end{bmatrix}$$

(%o4)

(%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 $f(\text{mat})$, 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 ,
```

$$2 - t \quad 2 - t \quad - t \quad - t \quad - t$$

Σ $\gamma_0 \Sigma$ $\gamma_0 \Xi$ Ξ Ξ $\gamma_0 \Xi$

```

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
          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 ]
[           ]
[           ]
[           ]
[ 2      - t ]
[ (t - 2 t) %e      ]
[ ----- ]
[           2             ]
[           ]
[ 2      - t      ]
[ (t + t) %e      ]
[           ]
[           ]
Col 1 = [ 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)
(%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   ]
[ 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")`.

47 distrib

47.1 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 definiciones 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

$$F(x) = \int_{-\infty}^x f(u) du$$

que es igual a la probabilidad $Pr(X \leq x)$.

La *media* es un parámetro de localización y se define como

$$E[X] = \int_{-\infty}^{\infty} x f(x) dx$$

La *varianza* es una medida de dispersión,

$$V[X] = \int_{-\infty}^{\infty} f(x) (x - E[X])^2 dx$$

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,

$$SK[X] = \frac{\int_{-\infty}^{\infty} f(x) (x - E[X])^3 dx}{D[X]^3}$$

Y el *coeficiente de curtosis* mide el apuntamiento de la densidad,

$$KU[X] = \frac{\int_{-\infty}^{\infty} f(x) (x - E[X])^4 dx}{D[X]^4} - 3$$

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 *probabilidad*, $f(x)$ 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

$$F(x) = \sum_{x_i \leq x} f(x_i)$$

La media, varianza, desviación típica y los coeficientes de asimetría y curtosis adquieren las formas

$$E[X] = \sum_{x_i} x_i f(x_i),$$

$$V[X] = \sum_{x_i} f(x_i) (x_i - E[X])^2,$$

$$D[X] = \sqrt{V[X]},$$

$$SK[X] = \frac{\sum_{x_i} f(x_i) (x_i - E[X])^3}{D[X]^3} dx$$

y

$$KU[X] = \frac{\sum_{x_i} f(x_i) (x_i - E[X])^4}{D[X]^4} dx - 3,$$

respectivamente.

Por favor, consultese 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)

```

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 '*mario AR-ROBA edu PUNTO xunta PUNTO es*'.

47.2 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) assume(s>0)$ cdf_normal(x,m,s);
           x - m
           erf(-----)
           sqrt(2) s   1
(%o3)      ----- + -
           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) assume(n>2)$  var_student_t(n);
          n
(%o3)      -----
          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));
      .01370030107589574 sqrt(5)
(%o2) -----
      sqrt(2) sqrt(14) sqrt(%pi)
      1.654562884111515E-4 sqrt(5)
+ -----
      sqrt(%pi)
      .02434921505438663 sqrt(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 si la variable global `numer` vale `true` o si alguno de sus argumentos es un número decimal, en otro caso devuelve una expresión nominal. Para hacer uso de esta función, ejecútese primero `load(distrib)`.

```
(%i1) load (distrib)$
(%i2) cdf_noncentral_student_t(-2,5,-5);
(%o2) cdf_noncentral_student_t(- 2, 5, - 5)
(%i3) cdf_noncentral_student_t(-2.0,5,-5);
(%o3)          .9952030093319743
```

quantile_noncentral_student_t (q,n,ncp)

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) (assume(df>1), mean_noncentral_student_t(df,k));
          df - 1
          gamma(-----) sqrt(df) k
                  2
(%o2)      -----
                      df
          sqrt(2) gamma(--)
                  2
```

var_noncentral_student_t (*n,ncp*) 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 $\text{Chi}^2(n)$, con $n > 0$.

La variable aleatoria $\text{Chi}^2(n)$ equivale a una $\text{Gamma}(n/2, 2)$, por lo que cuando Maxima no tiene suficiente información para obtener el resultado, devuelve una forma nominal basada en la función de densidad de la gamma.

```
(%i1) load (distrib)$
(%i2) pdf_chi2(x,n);
(%o2)                               n
                           pdf_gamma(x, -, 2)
                           2
(%i3) assume(x>0, n>0)$ pdf_chi2(x,n);
                           n/2 - 1 - x/2
                           x      %e
(%o4) -----
                           n/2      n
                           2      gamma(-)
                           2
```

cdf_chi2 (x,n)

Función

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria chi-cuadrado $\text{Chi}^2(n)$, con $n > 0$.

```
(%i1) load (distrib)$
(%i2) cdf_chi2(3,4);
(%o2)      1 - gamma_incomplete_regularized(2, -)
                           3
                           2
(%i3) float(%);
(%o3)      .4421745996289256
```

quantile_chi2 (q,n)

Función

Devuelve el q -cuantil de una variable aleatoria $\text{Chi}^2(n)$, 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 si la variable global **numer** vale **true**, en otro caso devuelve una expresión nominal basada en la función cuantil de la gamma, puesto que la variable aleatoria $\text{Chi}^2(n)$ equivale a una $\text{Gamma}(n/2, 2)$.

```
(%i1) load (distrib)$
(%i2) quantile_chi2(0.99,9);
(%o2)      21.66599433346194
(%i3) quantile_chi2(0.99,n);
(%o3)      n
                           quantile_gamma(0.99, -, 2)
                           2
```

mean_chi2 (n)

Función

Devuelve la media de una variable aleatoria $\text{Chi}^2(n)$, con $n > 0$.

La variable aleatoria $\text{Chi}^2(n)$ equivale a una $\text{Gamma}(n/2, 2)$, por lo que cuando Maxima no tiene suficiente información para obtener el resultado, devuelve una forma nominal basada en la media de la gamma.

```
(%i1) load (distrib)$
(%i2) mean_chi2(n);
          n
(%o2)           mean_gamma(-, 2)
                  2
(%i3) assume(n>0)$ mean_chi2(n);
(%o4)                   n
```

var_chi2 (n)

Función

Devuelve la varianza de una variable aleatoria $\text{Chi}^2(n)$, con $n > 0$.

La variable aleatoria $\text{Chi}^2(n)$ equivale a una $\text{Gamma}(n/2, 2)$, por lo que cuando Maxima no tiene suficiente información para obtener el resultado, devuelve una forma nominal basada en la varianza de la gamma.

```
(%i1) load (distrib)$
(%i2) var_chi2(n);
          n
(%o2)      var_gamma(-, 2)
                  2
(%i3) assume(n>0)$ var_chi2(n);
(%o4)           2 n
```

std_chi2 (n)

Función

Devuelve la desviación típica de una variable aleatoria $\text{Chi}^2(n)$, con $n > 0$.

La variable aleatoria $\text{Chi}^2(n)$ equivale a una $\text{Gamma}(n/2, 2)$, por lo que cuando Maxima no tiene suficiente información para obtener el resultado, devuelve una forma nominal basada en la desviación típica de la gamma.

```
(%i1) load (distrib)$
(%i2) std_chi2(n);
          n
(%o2)      std_gamma(-, 2)
                  2
(%i3) assume(n>0)$ std_chi2(n);
(%o4)           sqrt(2) sqrt(n)
```

skewness_chi2 (n)

Función

Devuelve el coeficiente de asimetría de una variable aleatoria $\text{Chi}^2(n)$, con $n > 0$.

La variable aleatoria $\text{Chi}^2(n)$ equivale a una $\text{Gamma}(n/2, 2)$, por lo que cuando Maxima no tiene suficiente información para obtener el resultado, devuelve una forma nominal basada en el coeficiente de asimetría de la gamma.

```
(%i1) load (distrib)$
(%i2) skewness_chi2(n);
          n
(%o2)      skewness_gamma(-, 2)
```

```

2
(%i3) assume(n>0)$ skewness_chi2(n);
          2 sqrt(2)
(%o4)
-----  

          sqrt(n)

```

kurtosis_chi2 (n)

Función

Devuelve el coeficiente de curtosis una variable aleatoria $Chi^2(n)$, con $n > 0$.

La variable aleatoria $Chi^2(n)$ equivale a una $Gamma(n/2, 2)$, por lo que cuando Maxima no tiene suficiente información para obtener el resultado, devuelve una forma nominal basada en el coeficiente de curtosis de la gamma.

```

(%i1) load (distrib)$
(%i2) kurtosis_chi2(n);
           n
(%o2)      kurtosis_gamma(-, 2)
           2
(%i3) assume(n>0)$ kurtosis_chi2(n);
           12
(%o4)
---  

           n

```

random_chi2 (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 $ncChi^2(n, ncp)$, con $n > 0$ y parámetro de no centralidad $ncp \geq 0$. 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 $ncChi^2(n, ncp)$, con $n > 0$ y parámetro de no centralidad $ncp \geq 0$.

quantile_noncentral_chi2 (q,n,ncp)

Función

Devuelve el q -cuantil de una variable aleatoria chi-cuadrado no centrada $ncChi^2(n, ncp)$, con $n > 0$ y parámetro de no centralidad $ncp \geq 0$; 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 si la variable global **numer** vale **true**, en otro caso devuelve una expresión nominal.

mean_noncentral_chi2 (<i>n,ncp</i>)	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 $ncp \geq 0$.	
var_noncentral_chi2 (<i>n,ncp</i>)	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 $ncp \geq 0$.	
std_noncentral_chi2 (<i>n,ncp</i>)	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 $ncp \geq 0$.	
skewness_noncentral_chi2 (<i>n,ncp</i>)	Función
Devuelve el coeficiente de asimetría de una variable aleatoria chi-cuadrado no centrada $nc_Chi^2(n, ncp)$, con $n > 0$ y parámetro de no centralidad $ncp \geq 0$.	
kurtosis_noncentral_chi2 (<i>n,ncp</i>)	Función
Devuelve el coeficiente de curtosis una variable aleatoria chi-cuadrado no centrada $nc_Chi^2(n, ncp)$, con $n > 0$ y parámetro de no centralidad $ncp \geq 0$.	
random_noncentral_chi2 (<i>n,ncp</i>)	Función
random_noncentral_chi2 (<i>n,ncp,m</i>)	Función
Devuelve un valor aleatorio $nc_Chi^2(n, ncp)$, con $n > 0$ y parámetro de no centralidad $ncp \geq 0$. Llamando a random_noncentral_chi2 con un tercer argumento <i>m</i> , se simulará una muestra aleatoria de tamaño <i>m</i> .	
Para hacer uso de esta función, ejecútese primero load(distrib) .	
pdf_f (<i>x,m,n</i>)	Función
Devuelve el valor correspondiente a <i>x</i> 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 (<i>x,m,n</i>)	Función
Devuelve el valor correspondiente a <i>x</i> 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 8 2 11
(%o2) 1 - beta_incomplete_regularized(-, -, --)	
(%i3) float(%);	
(%o3)	0.66756728179008
quantile_f (<i>q,m,n</i>)	Función
Devuelve el <i>q</i> -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 <i>q</i> debe ser un número de $[0, 1]$.	

Esta función no tiene expresión compacta, por lo que es evaluada numéricamente si la variable global `numer` vale `true`, en caso contrario devuelve una forma nominal.

```
(%i1) load (distrib)$
(%i2) quantile_f(2/5,sqrt(3),5);
          2
(%o2)           quantile_f(-, sqrt(3), 5)
          5
(%i3) %,numer;
(%o3)          0.518947838573693
```

mean_f (m,n)

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 $Weibull(1, 1/m)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en la densidad de Weibull.

```
(%i1) load (distrib)$
(%i2) pdf_exp(x,m);
(%o2)                               
$$\frac{1}{m} e^{-\frac{x}{m}}$$

(%i3) assume(x>0,m>0)$ pdf_exp(x,m);
(%o3)                               
$$\frac{1}{m} e^{-\frac{x}{m}}$$

(%o4)                               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 $Weibull(1, 1/m)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en la distribución de Weibull.

```
(%i1) load (distrib)$
(%i2) cdf_exp(x,m);
(%o2)                               
$$\frac{1}{m} e^{-\frac{x}{m}}$$

(%i3) assume(x>0,m>0)$ cdf_exp(x,m);
(%o3)                               
$$\frac{1}{m} e^{-\frac{x}{m}}$$

(%o4)                               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 $Weibull(1, 1/m)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en el cuantil de la Weibull.

```
(%i1) load (distrib)$
(%i2) quantile_exp(0.56,5);
(%o2)                               .1641961104139661
(%i3) quantile_exp(0.56,m);
(%o3)                               
$$\frac{1}{m} \ln(2)$$

```

mean_exp (m)

Función

Devuelve la media de una variable aleatoria $Exponencial(m)$, con $m > 0$.

La variable aleatoria $Exponencial(m)$ equivale a una $Weibull(1, 1/m)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en la media de la Weibull.

```
(%i1) load (distrib)$
(%i2) mean_exp(m);
```

```
(%o2)           1
               mean_weibull(1, -)
                   m
(%i3) assume(m>0)$ mean_exp(m);
(%o4)           1
                   -
                   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*), por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en la varianza de la Weibull.

```
(%i1) load (distrib)$
(%i2) var_exp(m);
(%o2)           1
               var_weibull(1, -)
                   m
(%i3) assume(m>0)$ var_exp(m);
(%o4)           1
                   --
                   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*), por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en la desviación típica de la Weibull.

```
(%i1) load (distrib)$
(%i2) std_exp(m);
(%o2)           1
               std_weibull(1, -)
                   m
(%i3) assume(m>0)$ std_exp(m);
(%o4)           1
                   -
                   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*), por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en el coeficiente de asimetría de la Weibull.

```
(%i1) load (distrib)$
(%i2) skewness_exp(m);
```

```
(%o2)      1
           skewness_weibull(1, -)
           m
(%i3) assume(m>0)$ skewness_exp(m);
(%o4)      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 $Weibull(1, 1/m)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en el coeficiente de curtosis de la Weibull.

```
(%i1) load (distrib)$
(%i2) kurtosis_exp(m);
(%o2)      1
           kurtosis_weibull(1, -)
           m
(%i3) assume(m>0)$ kurtosis_exp(m);
(%o4)      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
(%o3)      ----- + -
           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 (<i>m,s</i>)	Función
Devuelve la media de una variable aleatoria $\text{Lognormal}(m, s)$, con $s > 0$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
var_lognormal (<i>m,s</i>)	Función
Devuelve la varianza de una variable aleatoria $\text{Lognormal}(m, s)$, con $s > 0$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
std_lognormal (<i>m,s</i>)	Función
Devuelve la desviación típica de una variable aleatoria $\text{Lognormal}(m, s)$, con $s > 0$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
skewness_lognormal (<i>m,s</i>)	Función
Devuelve el coeficiente de asimetría de una variable aleatoria $\text{Lognormal}(m, s)$, con $s > 0$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
kurtosis_lognormal (<i>m,s</i>)	Función
Devuelve el coeficiente de curtosis una variable aleatoria $\text{Lognormal}(m, s)$, con $s > 0$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
random_lognormal (<i>m,s</i>)	Función
random_lognormal (<i>m,s,n</i>)	Función
Devuelve un valor aleatorio $\text{Lognormal}(m, s)$, con $s > 0$. Llamando a random_lognormal con un tercer argumento <i>n</i> , se simulará una muestra aleatoria de tamaño <i>n</i> .	
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 (<i>x,a,b</i>)	Función
Devuelve el valor correspondiente a <i>x</i> de la función de densidad de una variable aleatoria $\text{Gamma}(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
cdf_gamma (<i>x,a,b</i>)	Función
Devuelve el valor correspondiente a <i>x</i> de la función de distribución de una variable aleatoria $\text{Gamma}(a, b)$, con $a, b > 0$.	
(%i1) load (distrib)\$	
(%i2) cdf_gamma(3,5,21);	
(%o2) $1 - \frac{\text{gamma_incomplete_regularized}(5, \frac{1}{7})}{\text{gamma_incomplete_regularized}(5, \frac{1}{7})}$	
(%i3) float(%);	
(%o3) $4.402663157376807\text{E}-7$	

quantile_gamma (q, a, b)	Función
Devuelve el q -cuantil de una variable aleatoria $\text{Gamma}(a, b)$, 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 $\text{Gamma}(a, b)$, 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 $\text{Gamma}(a, b)$, 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 $\text{Gamma}(a, b)$, 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 $\text{Gamma}(a, b)$, 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 $\text{Gamma}(a, b)$, 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 $\text{Gamma}(a, b)$, 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 $a \geq 1$, Cheng, R.C.H. y Feast, G.M. (1979). <i>Some simple gamma variate generators</i> . Appl. Stat., 28, 3, 290-295.	
Para $0 < a < 1$, Ahrens, J.H. y Dieter, U. (1974). <i>Computer methods for sampling from gamma, beta, poisson and binomial distributions</i> . 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 $\text{Beta}(a, b)$, 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 $\text{Beta}(a, b)$, 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 descrito 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 <i>Uniforme Continua</i> (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 <i>Uniforme Continua</i> (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 <i>Uniforme Continua</i> (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 <i>Uniforme Continua</i> (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 <i>Uniforme Continua</i> (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 <i>Uniforme Continua</i> (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 <i>Uniforme Continua</i> (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 <i>Uniforme Continuo</i> (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 <i>Logística</i> (a, b), con $b > 0$. Para hacer uso de esta función, ejecútese primero load(distrib) .	

cdf_logistic (<i>x,a,b</i>)	Función
Devuelve el valor correspondiente a <i>x</i> de la función de distribución de una variable aleatoria $\text{Logística}(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
quantile_logistic (<i>q,a,b</i>)	Función
Devuelve el <i>q</i> -cuantil de una variable aleatoria $\text{Logística}(a, b)$, con $b > 0$; en otras palabras, se trata de la inversa de cdf_logistic . El argumento <i>q</i> debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
mean_logistic (<i>a,b</i>)	Función
Devuelve la media de una variable aleatoria $\text{Logística}(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
var_logistic (<i>a,b</i>)	Función
Devuelve la varianza de una variable aleatoria $\text{Logística}(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
std_logistic (<i>a,b</i>)	Función
Devuelve la desviación típica de una variable aleatoria $\text{Logística}(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
skewness_logistic (<i>a,b</i>)	Función
Devuelve el coeficiente de asimetría de una variable aleatoria $\text{Logística}(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
kurtosis_logistic (<i>a,b</i>)	Función
Devuelve el coeficiente de curtosis una variable aleatoria $\text{Logística}(a, b)$, con $b > 0$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
random_logistic (<i>a,b</i>)	Función
random_logistic (<i>a,b,n</i>)	Función
Devuelve un valor aleatorio $\text{Logístico}(a, b)$, con $b > 0$. Llamando a random_logistic con un tercer argumento <i>n</i> , se simulará una muestra aleatoria de tamaño <i>n</i> . El algoritmo de simulación está basado en el método inverso. Para hacer uso de esta función, ejecútese primero load(distrib) .	
pdf_pareto (<i>x,a,b</i>)	Función
Devuelve el valor correspondiente a <i>x</i> de la función de densidad de una variable aleatoria de $\text{Pareto}(a, b)$, con $a, b > 0$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
cdf_pareto (<i>x,a,b</i>)	Función
Devuelve el valor correspondiente a <i>x</i> de la función de distribución de una variable aleatoria de $\text{Pareto}(a, b)$, 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 $Weibull(2, 1/b)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en la densidad de la Weibull.

```
(%i1) load (distrib)$
(%i2) pdf_rayleigh(x,b);
(%o2)                               pdf_weibull(x, 2, -)
                                         1
                                         b
(%i3) assume(x>0,b>0)$ pdf_rayleigh(x,b);
                                         2   2
                                         2      - b  x
(%o4)                               2 b  x %e
```

cdf_rayleigh (x,b) 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 $Weibull(2, 1/b)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en la distribución de la Weibull.

```
(%i1) load (distrib)$
(%i2) cdf_rayleigh(x,b);
(%o2)                               1
                  cdf_weibull(x, 2, -)
                                         b
(%i3) assume(x>0,b>0)$ cdf_rayleigh(x,b);
(%o3)                               2
                                         2
                                         - b   x
(%o4)                               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 $Weibull(2, 1/b)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en los cuantiles de la Weibull.

```
(%i1) load (distrib)$
(%i2) quantile_rayleigh(0.99,b);
(%o2)                               1
                  quantile_weibull(0.99, 2, -)
                                         b
(%i3) assume(x>0,b>0)$ quantile_rayleigh(0.99,b);
(%o3)                               2.145966026289347
(%o4)-----
```

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 $Weibull(2, 1/b)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en la media de la Weibull.

```
(%i1) load (distrib)$
(%i2) mean_rayleigh(b);
(%o2)                               1
                  mean_weibull(2, -)
                                         b
(%i3) assume(b>0)$ mean_rayleigh(b);
(%o3)-----
```

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 $Weibull(2, 1/b)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en la varianza de la Weibull.

```
(%i1) load (distrib)$
(%i2) var_rayleigh(b);
(%o2) var_weibull(2, b)
(%i3) assume(b>0)$ var_rayleigh(b);
(%o3) 
$$\frac{\pi}{b^2}$$

(%o4)
```

std_rayleigh (b)

Función

Devuelve la desviación típica de una variable aleatoria de $\text{Rayleigh}(b)$, con $b > 0$.

La variable aleatoria $Rayleigh(b)$ equivale a una $Weibull(2, 1/b)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en la desviación típica de la Weibull.

```
(%i1) load (distrib)$
(%i2) std_rayleigh(b);
(%o2)                               std_weibull(2, -)
(%i3) assume(b>0)$ std_rayleigh(b);
(%o3) %pi
(%o4) sqrt(1 - ---)
                  4
-----
```

skewness_rayleigh (b)

Función

Devuelve el coeficiente de asimetría de una variable aleatoria de $\text{Rayleigh}(b)$, con $b > 0$.

La variable aleatoria $\text{Rayleigh}(b)$ equivale a una $\text{Weibull}(2, 1/b)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en el coeficiente de asimetría de la Weibull.

```

(%i1) load (distrib)$
(%i2) skewness_rayleigh(b);
(%o2) skewness_weibull(2, -)
(%i3) assume(b>0)$ skewness_rayleigh(b);
(%o3) 3/2
      %pi      3 sqrt(%pi)

```

```

----- -----
        4      4
----- -----
(%o4)      %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 $Weibull(2, 1/b)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en el coeficiente de curtosis de la Weibull.

```

(%i1) load (distrib)$
(%i2) kurtosis_rayleigh(b);
(%o2)      kurtosis_weibull(2, -)
                  1
                  b
(%i3) assume(b>0)$ kurtosis_rayleigh(b);
(%o3)      2
                  3 %pi
                  16
----- - 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) assume(b>0)$ mean_gumbel(a,b);  
(%o3) %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) assume(b>0)$ skewness_gumbel(a,b);  
      12 sqrt(6) zeta(3)  
(%o3) -----  
            3  
            %pi  
(%i4) numer:true$ skewness_gumbel(a,b);  
(%o5) 1.139547099404649
```

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)`.

47.3 Funciones y variables para distribuciones discretas

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 $0 < p < 1$ 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 $0 < p < 1$ 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 $0 < p < 1$ 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 $0 < p < 1$ 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 $0 < p < 1$ 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 $0 < p < 1$ 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 $0 < p < 1$ 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 $0 < p < 1$ y n entero positivo. Para hacer uso de esta función, ejecútese primero <code>load(distrib)</code> .	
random_binomial (n,p)	Función
random_binomial (n,p,m)	Función
Devuelve un valor aleatorio $Binomial(n, p)$, con $0 < p < 1$ 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) <i>Binomial Random Variate Generation</i> . Communications of the ACM, 31, Feb., 216.	
Para hacer uso de esta función, ejecútese primero <code>load(distrib)</code> .	
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 <code>load(distrib)</code> .	
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) <code>load (distrib)\$</code> (%i2) <code>cdf_poisson(3,5);</code> (%o2) <code>gamma_incomplete_regularized(4, 5)</code> (%i3) <code>float(%);</code> (%o3) <code>.2650259152973623</code>	
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 <code>load(distrib)</code> .	
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 <code>load(distrib)</code> .	
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 <code>load(distrib)</code> .	
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 <code>load(distrib)</code> .	

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 $0 < p < 1$.

La variable aleatoria $Bernoulli(p)$ equivale a una $Binomial(1, p)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en la función de probabilidad de la binomial.

```
(%i1) load (distrib)$
(%i2) pdf_bernoulli(1,p);
(%o2)                      pdf_binomial(1, 1, p)
(%i3) assume(0<p,p<1)$ pdf_bernoulli(1,p);
(%o4)                                p
```

cdf_bernoulli (x,p) Función

Devuelve el valor correspondiente a x de la función de distribución de una variable aleatoria de $Bernoulli(p)$, con $0 < p < 1$. 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 $0 < p < 1$; 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 $0 < p < 1$.

La variable aleatoria $Bernoulli(p)$ equivale a una $Binomial(1, p)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en la media de la binomial.

```
(%i1) load (distrib)$
(%i2) mean_bernoulli(p);
(%o2)                      mean_binomial(1, p)
(%i3) assume(0<p,p<1)$ mean_bernoulli(p);
(%o4)                               p
```

var_bernoulli (p)

Función

Devuelve la varianza de una variable aleatoria de $Bernoulli(p)$, con $0 < p < 1$.

La variable aleatoria $Bernoulli(p)$ equivale a una $Binomial(1, p)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en la varianza de la binomial.

```
(%i1) load (distrib)$
(%i2) var_bernoulli(p);
(%o2)                      var_binomial(1, p)
(%i3) assume(0<p,p<1)$ var_bernoulli(p);
(%o4)           (1 - p) p
```

std_bernoulli (p)

Función

Devuelve la desviación típica de una variable aleatoria de $Bernoulli(p)$, con $0 < p < 1$.

La variable aleatoria $Bernoulli(p)$ equivale a una $Binomial(1, p)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en la desviación típica de la binomial.

```
(%i1) load (distrib)$
(%i2) std_bernoulli(p);
(%o2)                      std_binomial(1, p)
(%i3) assume(0<p,p<1)$ std_bernoulli(p);
(%o4)           sqrt(1 - p) sqrt(p)
```

skewness_bernoulli (p)

Función

Devuelve el coeficiente de asimetría de una variable aleatoria de $Bernoulli(p)$, con $0 < p < 1$.

La variable aleatoria $Bernoulli(p)$ equivale a una $Binomial(1, p)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en el coeficiente de asimetría de la binomial.

```
(%i1) load (distrib)$
(%i2) skewness_bernoulli(p);
(%o2)                      skewness_binomial(1, p)
(%i3) assume(0<p,p<1)$ skewness_bernoulli(p);
(%o4)           1 - 2 p
(%o4)           -----
                           sqrt(1 - p) sqrt(p)
```

kurtosis_bernoulli (p)

Función

Devuelve el coeficiente de curtosis una variable aleatoria de $Bernoulli(p)$, con $0 < p < 1$.

La variable aleatoria $Bernoulli(p)$ equivale a una $Binomial(1, p)$, por lo que si Maxima no tiene suficiente información para calcular el resultado, devolverá una forma nominal basada en el coeficiente de curtosis de la binomial.

```
(%i1) load (distrib)$
(%i2) kurtosis_bernoulli(p);
(%o2)                               kurtosis_binomial(1, p)
(%i3) assume(0 < p, p < 1)$ kurtosis_bernoulli(p);
(%o4)          
$$\frac{1 - 6(1 - p)p}{(1 - p)p}$$

```

random_bernoulli (p)
random_bernoulli (p, n)

Función
Función

Devuelve un valor aleatorio $Bernoulli(p)$, con $0 < p < 1$. 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 $0 < p < 1$. 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 $0 < p < 1$. 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 $0 < p < 1$; 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 $0 < p < 1$. Para hacer uso de esta función, ejecútese primero **load(distrib)**.

var_geometric (p)

Función

Devuelve la varianza de una variable aleatoria $Geométrica(p)$, con $0 < p < 1$. 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 $0 < p < 1$. 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 $0 < p < 1$. Para hacer uso de esta función, ejecútese primero <code>load(distrib)</code> .	
kurtosis_geometric (p)	Función
Devuelve el coeficiente de curtosis de una variable aleatoria $Geométrica(p)$, con $0 < p < 1$. Para hacer uso de esta función, ejecútese primero <code>load(distrib)</code> .	
random_geometric (p)	Función
random_geometric (p,n)	Función
Devuelve un valor aleatorio $Geométrico(p)$, con $0 < p < 1$. Llamando a <code>random_geometric</code> 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 <code>load(distrib)</code> .	
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 <code>load(distrib)</code> .	
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 <code>load(distrib)</code> .	
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 <code>cdf_discrete_uniform</code> . El argumento q debe ser un número de $[0, 1]$. Para hacer uso de esta función, ejecútese primero <code>load(distrib)</code> .	
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 <code>load(distrib)</code> .	
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 <code>load(distrib)</code> .	
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 <code>load(distrib)</code> .	

skewness_discrete_uniform (<i>n</i>)	Función
Devuelve el coeficiente de asimetría de una variable aleatoria <i>Uniforme Discreta(n)</i> , con <i>n</i> entero positivo. Para hacer uso de esta función, ejecútese primero load(distrib) .	
kurtosis_discrete_uniform (<i>n</i>)	Función
Devuelve el coeficiente de curtosis de una variable aleatoria <i>Uniforme Discreta(n)</i> , con <i>n</i> entero positivo. Para hacer uso de esta función, ejecútese primero load(distrib) .	
random_discrete_uniform (<i>n</i>)	Función
random_discrete_uniform (<i>n,m</i>)	Función
Devuelve un valor aleatorio <i>UniformeDiscreto(n)</i> , con <i>n</i> entero positivo. Llamando a random_discrete_uniform con un segundo argumento <i>m</i> , se simulará una muestra aleatoria de tamaño <i>m</i> .	
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 (<i>x,n1,n2,n</i>)	Función
Devuelve el valor correspondiente a <i>x</i> de la función de probabilidad de una variable aleatoria <i>Hipergeométrica(n1,n2,n)</i> , con <i>n1</i> , <i>n2</i> y <i>n</i> enteros positivos y <i>n</i> <= <i>n1+n2</i> . Para hacer uso de esta función, ejecútese primero load(distrib) .	
cdf_hypergeometric (<i>x,n1,n2,n</i>)	Función
Devuelve el valor correspondiente a <i>x</i> de la función de distribución de una variable aleatoria <i>Hipergeométrica(n1,n2,n)</i> , con <i>n1</i> , <i>n2</i> y <i>n</i> enteros positivos y <i>n</i> <= <i>n1+n2</i> . Para hacer uso de esta función, ejecútese primero load(distrib) .	
quantile_hypergeometric (<i>q,n1,n2,n</i>)	Función
Devuelve el <i>q</i> -cuantil de una variable aleatoria <i>Hipergeométrica(n1,n2,n)</i> , con <i>n1</i> , <i>n2</i> y <i>n</i> enteros positivos y <i>n</i> <= <i>n1+n2</i> ; en otras palabras, se trata de la inversa de cdf_hypergeometric . El argumento <i>q</i> debe ser un número de [0, 1]. Para hacer uso de esta función, ejecútese primero load(distrib) .	
mean_hypergeometric (<i>n1,n2,n</i>)	Función
Devuelve la media de una variable aleatoria uniforme discreta <i>Hyp(n1,n2,n)</i> , con <i>n1</i> , <i>n2</i> y <i>n</i> enteros positivos y <i>n</i> <= <i>n1+n2</i> . Para hacer uso de esta función, ejecútese primero load(distrib) .	
var_hypergeometric (<i>n1,n2,n</i>)	Función
Devuelve la varianza de una variable aleatoria <i>Hipergeométrica(n1,n2,n)</i> , con <i>n1</i> , <i>n2</i> y <i>n</i> enteros positivos y <i>n</i> <= <i>n1+n2</i> . Para hacer uso de esta función, ejecútese primero load(distrib) .	

std_hypergeometric (n_1, n_2, n)	Función
Devuelve la desviación típica de una variable aleatoria $Hipergeométrica(n_1, n_2, n)$, con n_1, n_2 y n enteros positivos y $n \leq n_1 + n_2$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
skewness_hypergeometric (n_1, n_2, n)	Función
Devuelve el coeficiente de asimetría de una variable aleatoria $Hipergeométrica(n_1, n_2, n)$, con n_1, n_2 y n enteros positivos y $n \leq n_1 + n_2$. Para hacer uso de esta función, ejecútese primero load(distrib) .	
random_hypergeometric (n_1, n_2, n)	Función
random_hypergeometric (n_1, n_2, n, m)	Función
Devuelve un valor aleatorio $Hipergeométrico(n_1, n_2, n)$, con n_1, n_2 y n enteros positivos y $n \leq n_1 + n_2$. 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) <i>Computer generation of hypergeometric random variates</i> . 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 $0 < p < 1$ y n entero 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 $0 < p < 1$ y n entero positivo.	
(%i1) load (distrib)\$ (%i2) cdf_negative_binomial(3, 4, 1/8); (%o2) $\frac{3271}{524288}$ (%i3) float(%); (%o3) .006238937377929687	
quantile_negative_binomial (q, n, p)	Función
Devuelve el q -cuantil de una variable aleatoria $Binomial\ Negativa(n, p)$, con $0 < p < 1$ y n entero 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 $0 < p < 1$ and n entero 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 $0 < p < 1$ and n entero positivo. Para hacer uso de esta función, ejecútese primero <code>load(distrib)</code> .	
std_negative_binomial (n,p)	Función
Devuelve la desviación típica de una variable aleatoria $Binomial\ Negativa(n, p)$, con $0 < p < 1$ and n entero positivo. Para hacer uso de esta función, ejecútese primero <code>load(distrib)</code> .	
skewness_negative_binomial (n,p)	Función
Devuelve el coeficiente de asimetría de una variable aleatoria $Binomial\ Negativa(n, p)$, con $0 < p < 1$ and n entero positivo. Para hacer uso de esta función, ejecútese primero <code>load(distrib)</code> .	
kurtosis_negative_binomial (n,p)	Función
Devuelve el coeficiente de curtosis una variable aleatoria binomial negativa $NB(n, p)$, con $0 < p < 1$ and n entero positivo. Para hacer uso de esta función, ejecútese primero <code>load(distrib)</code> .	
random_negative_binomial (n,p)	Función
random_negative_binomial (n,p,m)	Función
Devuelve un valor aleatorio $Binomial\ Negativo(n, p)$, con $0 < p < 1$ y n entero positivo. Llamando a <code>random_negative_binomial</code> con un tercer argumento m , se simulará una muestra aleatoria de tamaño m .	
Algoritmo descrito en Devroye, L. (1986) <i>Non-Uniform Random Variate Generation</i> . Springer Verlag, p. 480.	
Para hacer uso de esta función, ejecútese primero <code>load(distrib)</code> .	

48 draw

48.1 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`.

Sígase este enlace para ver ejemplos más elaborados de este paquete:

<http://www.telefonica.net/web2/biomates/maxima/gpdraw>

Se necesita tener instalado Gnuplot 4.2 para ejecutar este paquete.

48.2 Funciones y variables para draw

`proportional_axes`

Opción gráfica

Valor por defecto: `none`

Cuando `proportional_axes` es igual a `xy`, una escena 2D 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 para gráficos 2D. Una vez se haya liberado la versión 4.3 de Gnuplot, se extenderá `proportional_axes` a los gráficos 3D.

Ejemplos:

Gráfico sencillo.

```
(%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))
```

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`.

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`.

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`.

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`.

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` 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` 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` y `logy`.

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`.

terminal

Opción gráfica

Valor por defecto: `screen`

Selecciona el terminal a utilizar por Gnuplot; valores posibles son: `screen` (por defecto), `png`, `jpg`, `eps`, `eps_color`, `pdf`, `pdfcairo`, `gif`, `animated_gif`, `wxt` 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. Actualmente (Nov 2008) Gnuplot 4.3 es una versión en desarrollo. `pdf` necesita que Gnuplot 4.2/4.3 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,
              pic_width = 300,
              explicit(x^2,x,-1,1))$
(%i4) /* jpg file */
       draw2d(terminal = 'jpg,
              pic_width = 300,
              pic_height = 300,
              explicit(x^2,x,-1,1))$
(%i5) /* eps file */
       draw2d(file_name = "myfile",
              explicit(x^2,x,-1,1),
              terminal = 'eps)$
(%i6) /* pdf file */
       draw2d(file_name = "mypdf",
              pdf_width = 12.0,
              pdf_height = 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.

Véanse también `file_name`, `pic_width`, `pic_height` y `delay`.

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", "Helvetica-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*.
- *pdftcairo*: 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

Valor por defecto: 12

Opción gráfica

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`.

grid Opción gráfica

Valor por defecto: `false`

Cuando `grid` vale `true`, se dibujará una rejilla sobre el 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) draw2d(grid = true,
              explicit(exp(u),u,-2,2))$
```

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")$
```

xlabel Opción gráfica

Valor por defecto: `""` (cadena vacía)

La opción `xlabel` almacena una cadena con la etiqueta del eje `x`. 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(xlabel = "Time",
              explicit(exp(u),u,-2,2),
              ylabel = "Population")$
```

Véanse también `ylabel` y `zlabel`.

ylabel Opción gráfica

Valor por defecto: `""` (cadena vacía)

La opción `ylabel` almacena una cadena con la etiqueta del eje `y`. 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(xlabel = "Time",
              ylabel = "Population",
              explicit(exp(u),u,-2,2) )$
```

Véanse también xlabel y zlabel.

zlabel

Opción gráfica

Valor por defecto: "" (cadena vacía)

La opción zlabel almacena una cadena con la etiqueta del eje z. 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) 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.

xtics

Opción gráfica

Valor por defecto: auto

Esta opción gráfica controla la forma en la que se dibujarán las marcas del eje x.

- Cuando a xtics se le da el valor auto, las marcas se dibujarán de forma automática.
- Cuando a xtics se le da el valor none, 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 = 'none,
              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_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.

ytics

Opción gráfica

Valor por defecto: `auto`

Esta opción gráfica controla la forma en la que se dibujarán las marcas del eje y.

Véase `xtics` para una descripción completa.

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.

ztics

Opción gráfica

Valor por defecto: `auto`

Esta opción gráfica controla la forma en la que se dibujarán las marcas del eje z.

Véase `xtics` para una descripción completa.

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`.

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.

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.

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.

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_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.

`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_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.

`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.

`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_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_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`.

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_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**.

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_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**, **yticks_secondary**, **yticks_rotate_secondary** y **yticks_axis_secondary**.

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`.

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_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`.

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_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`.

`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_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`.

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))$
```

rot_vertical

Opción gráfica

Valor por defecto: 60

`rot_vertical` es el ángulo (en grados) de la rotación vertical (alrededor del eje x) para situar el punto del observador en las escenas 3d.

El ángulo debe pertenecer al intervalo [0, 180].

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(rot_vertical = 170,
              explicit(sin(x^2+y^2),x,-2,2,y,-2,2) )$
```

Véase también `rot_horizontal`.

rot_horizontal

Opción gráfica

Valor por defecto: 30

`rot_horizontal` es el ángulo (en grados) de la rotación horizontal (alrededor del eje z) para situar el punto del observador en las escenas 3d.

El ángulo debe pertenecer al intervalo [0, 360].

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(rot_vertical = 170,
              rot_horizontal = 360,
              explicit(sin(x^2+y^2),x,-2,2,y,-2,2) )$
```

Véase también `rot_vertical`.

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.	
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:	
El terminal <i>dumb</i> no está soportado por el paquete draw , pero es posible activarlo haciendo uso de la opción user_preamble ,	
<pre>(%i1) load(draw)\$ (%i2) draw2d(explicit(exp(x)-1,x,-1,1), parametric(cos(u),sin(u),u,0,2*pi), user_preamble="set terminal dumb")\$</pre>	
file_name	Opción gráfica
Valor por defecto: "maxima_out"	
file_name es el nombre del fichero en el que los terminales png , jpg , eps , pdf and pdfcairo y eps_color 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:	
<pre>(%i1) load(draw)\$ (%i2) draw2d(file_name = "myfile", explicit(x^2,x,-1,1), terminal = 'png')\$</pre>	
Véanse también terminal , pic_width y pic_height .	
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`.

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`.

file_bgcolor

Opción gráfica

Valor por defecto: "xffffffff"

Establece el color de fondo en los terminales png, jpg y gif. Los colores deben definirse en código hexadecimal *rgb*. Nombres de colores no están permitidos. El color de fondo por defecto es blanco.

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 opción `delay` sólo se activa en caso de gifs animados; se ignora en cualquier otro caso.

See also `terminal`, `pic_width`, and `pic_height`.

pic_width Opción gráfica

Valor por defecto: 640

pic_width es la anchura del fichero de imagen de bits generado por los terminales **png** y **jpg**.

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(terminal = 'png,
              pic_width = 300,
              pic_height = 300,
              explicit(x^2,x,-1,1))$
```

Véanse también **terminal**, **file_name** y **pic_height**.

pic_height Opción gráfica

Valor por defecto: 640

pic_height es la altura del fichero de imagen de bits generado por los terminales **png** y **jpg**.

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(terminal = 'png,
              pic_width = 300,
              pic_height = 300,
              explicit(x^2,x,-1,1))$
```

Véanse también **terminal**, **file_name** y **pic_width**.

eps_width Opción gráfica

Valor por defecto: 12

eps_width es el ancho (medido en cm) del archivo Postscript generado por los terminales **eps** y **eps_color**.

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(terminal = 'eps,
              eps_width = 3,
              eps_height = 3,
              explicit(x^2,x,-1,1))$
```

Véanse también **terminal**, **file_name** y **eps_height**.

eps_height

Opción gráfica

Valor por defecto: 8

`eps_height` es la altura (medida en cm) del archivo Postscript generado por los terminales `eps` y `eps_color`.

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(terminal = 'eps,
              eps_width = 3,
              eps_height = 3,
              explicit(x^2,x,-1,1))$
```

Véanse también `terminal`, `file_name` y `eps_width`.

pdf_width

Opción gráfica

Valor por defecto: 21.0 (ancho A4)

Es el ancho (medida en cm) del documento PDF generado por los terminales `pdf` y `pdfcairo`.

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(terminal = 'pdf,
              pdf_width = 3.0,
              pdf_height = 3.0,
              explicit(x^2,x,-1,1))$
```

Véanse también `terminal`, `file_name` y `pdf_height`.

pdf_height

Opción gráfica

Valor por defecto: 29.7 (alto A4)

Es el alto (medida en cm) del documento PDF generado por los terminales `pdf` y `pdfcairo`.

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(terminal = 'pdf,
              pdf_width = 3.0,
              pdf_height = 3.0,
              explicit(x^2,x,-1,1))$
```

Véanse también `terminal`, `file_name` y `pdf_width`.

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_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`.

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_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.**palette**

Opción gráfica

Valor por defecto: `color``palette` indica cómo transformar los valores reales de una matriz pasada al objeto `image` en componentes cromáticas.`palette` 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.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplos:

```
(%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`.

colorbox

Opción gráfica

Valor por defecto: `true`

Cuando `colorbox` vale `true`, se dibuja una escala de color al lado de los objetos `image`.

Puesto que ésta es una opción global, su posición dentro de la descripción de la escena no reviste importancia.

Ejemplo:

```
(%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))$
```

Véase también `palette`.

enhanced3d

Opción gráfica

Valor por defecto: `false`

Si `enhanced3d` vale `false`, no se colorearán las superficies de los gráficos tridimensionales. Si `enhanced3d` vale `true`, las superficies se colorearán activando el modo `pm3d` de Gnuplot. Si se da una expresión a `enhanced3d`, ésta se utilizará para asignar colores de acuerdo con el valor de la opción `palette`; las variables de esta expresión deben ser las mismas que luego se utilicen para la descripción de la superficie.

Ejemplos:

```
(%i1) load(draw)$
(%i2) draw3d(
          surface_hide = true,
          enhanced3d  = true,
          palette     = gray,
          explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3))$
(%i3) draw3d(
          surface_hide = true,
          /* mismas variables x e y que */
          /* en explicit mas abajo:      */
          enhanced3d  = sin(x*y),
          explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3))$
(%i4) draw3d(
          color = blue,
          nticks = 60,
          line_width = 3,
          enhanced3d = (u-1)^2,
          parametric(cos(5*u)^2,sin(7*u),u-2,u,0,2))$
```

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],
    xrange = [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]]),
               pointsJoined = true,
               point_type   = square,
               line_type    = dots,
               points([[1,2],[5,2],[9,2]]),
               point_type   = circle,
               color        = red,
               line_width   = 7,
               points([[1,3],[5,3],[9,3]]) )$
```

filled_func

Opción gráfica

Valor por defecto: `false`

La opción `filled_func` establece cómo se van a llenar 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 de la ventana gráfica se llena 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 llenan.

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`.

transparent

Opción gráfica

Valor por defecto: `false`

Cuando `transparent` vale `true`, 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]]))$
```

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]]))$
```

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_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_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 llenas), `empty` (flechas cerradas pero no llenas) 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`.

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]))$
```

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`.

color Opción gráfica

Valor por defecto: `"black"`

`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-corral", "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 "#rrggb".

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", /* double quotes if - is used */
              explicit(1.5 + x^2,x,-1,1),
              color = "#23ab0f",
              label(["This is a label",0,1.2]) )$
```

Véase también `fill_color`.

fill_color

Opción gráfica

Valor por defecto: "red"

`fill_color` especifica el color para llenar 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 (dados por `fill_color`) en los objetos `bars`.

Véase `bars` para ejemplos.

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.

line_type

Opción gráfica

Valor por defecto: solid

line_type indica cómo se van a dibujar las líneas; valores posibles son solid y dots.

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.

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) )$
```

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.	
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:	
<ul style="list-style-type: none"> • gr2d: <code>points</code>, <code>polygon</code>, <code>rectangle</code>, <code>ellipse</code>, <code>vector</code>, <code>explicit</code>, <code>implicit</code>, <code>parametric</code> y <code>polar</code>. • gr3d: <code>points</code>, <code>explicit</code>, <code>parametric</code>, y <code>parametric_surface</code>. 	
Ejemplo:	
	(%i1) load(draw)\$ (%i2) draw2d(key = "Sinus", explicit(sin(x),x,0,10), key = "Cosinus", color = red, explicit(cos(x),x,0,10))\$
xu_grid	Opción gráfica
Valor por defecto: 30	
xu_grid es el número de coordenadas de la primera variable (x en superficies explícitas y u en las paramétricas) para formar la rejilla de puntos muestrales.	
Esta opción afecta a los siguientes objetos gráficos:	
<ul style="list-style-type: none"> • gr3d: <code>explicit</code> y <code>parametric_surface</code>. 	
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 .	
yv_grid	Opción gráfica
Valor por defecto: 30	
yv_grid es el número de coordenadas de la segunda variable (y en superficies explícitas y v en las paramétricas) para formar la rejilla de puntos muestrales.	
Esta opción afecta a los siguientes objetos gráficos:	
<ul style="list-style-type: none"> • gr3d: <code>explicit</code> y <code>parametric_surface</code>. 	
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`.

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)) )$
```

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`.

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)$
```

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**.**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.**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.**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.**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)`.

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*. 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: `points`, `polygon`, `rectangle`, `ellipse`, `label`, `vector`, `explicit`, `implicit`, `polar`, `parametric`, `image` y `geomap`.

Para utilizar esta función, ejecútese primero `load(draw)`.

Véanse también las siguientes *opciones gráficas*: `xrange`, `yrange`, `logx`, `logy`, `terminal`, `grid`, `title`, `xlabel`, `ylabel`, `xtics`, `ytics`, `xtics_rotate`, `ytics_rotate`, `xtics_axis`, `ytics_axis`, `xaxis`, `yaxis`, `xaxis_width`, `yaxis_width`, `xaxis_type`, `yaxis_type`, `xaxis_color`, `yaxis_color`, `xy_file`, `file_name`, `pic_width`, `pic_height`, `eps_width`, `eps_height`, `user_preamble`, `axis_bottom`, `axis_left`, `axis_top` y `axis_right`.

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*. 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: `points`, `label`, `vector`, `explicit`, `implicit`, `parametric`, `parametric_surface` y `geomap`.

Véanse también las siguientes *opciones gráficas*: `xrange`, `yrange`, `zrange`, `logx`, `logy`, `logz`, `terminal`, `grid`, `title`, `xlabel`, `ylabel`, `zlabel`, `xtics`, `ytics`, `ztics`, `xtics_rotate`, `ytics_rotate`, `ztics_rotate`, `xtics_axis`, `ytics_axis`, `ztics_axis`, `xaxis`, `yaxis`, `zaxis`, `xaxis_width`, `yaxis_width`, `zaxis_width`, `xaxis_type`, `yaxis_type`, `zaxis_type`, `xaxis_color`, `yaxis_color`, `zaxis_color`, `xy_file`, `user_preamble`, `axis_bottom`, `axis_left`, `axis_top`, `file_name`, `pic_width`, `pic_height`, `eps_width`, `eps_height`, `axis_right`, `rot_vertical`, `rot_horizontal`, `axis_3d`, `xu_grid`, `yv_grid`, `surface_hide`, `contour`, `contour_levels`, `palette`, `colorbox` y `enhanced3d`.

Para utilizar esta función, ejecútese primero `load(draw)`.

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 $[x_1, y_1]$, $[x_2, y_2]$, etc. Si no se dan las abscisas, éstas se asignan automáticamente a enteros positivos consecutivos, de forma que `points([y1,y2,...])` dibuja los puntos $[1, y_1]$, $[2, y_2]$, 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)) $
```

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]]) )$
```

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] )$
```

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);
```

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])$
```

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]) )$
```

vector ([x,y], [dx,dy])
vector ([x,y,z], [dx,dy,dz])

Objeto gráfico
 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)) )$
```

explicit (fcn,var,minval,maxval)
explicit (fcn,var1,minval1,maxval1,var2,minval2,maxval2)

Objeto gráfico
 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`, `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*: *xu_grid*, *yv_grid*, *line_type*, *line_width*, *key*, *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.

mesh (*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*.

Este objeto se ve afectado por las siguientes *opciones gráficas*: *line_type*, *line_width*, *key*, *enhanced3d* y *color*.

Este objeto gráfico ignora valores de *enhanced3d* distintos de *true* o *false*.

Ejemplo:

```
(%i1) load(draw)$
(%i2) m: apply(
              matrix,
              makelist(makelist(random(10.0),k,1,30),i,1,20)) $
(%i3) draw3d(
              color = blue,
              mesh(m,0,0,3,2),
              xlabel = "x",
              ylabel = "y",
              surface_hide = true);
```

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` 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);
```

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 )$
```

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* y *color*.

Ejemplo:

```
(%i1) load(draw)$
(%i2) draw3d(spherical(1,a,0,2*%pi,z,0,%pi))$
```

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* y *color*.

Ejemplo:

```
(%i1) load(draw)$
(%i2) draw3d(cylindrical(1,z,-2,2,az,0,2*%pi))$
```

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")$
```

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.

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`.

geomap (<i>numlist</i>)	Objeto gráfico
geomap (<i>numlist,3Dprojection</i>)	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(draw)$
(%i2) /* Vertices of boundary #0: {(1,1),(2,5),(4,3)} */
      ( bnd0: make_array(flonum,6),
        bnd0[0]:1.0, bnd0[1]:1.0, bnd0[2]:2.0,
        bnd0[3]:5.0, bnd0[4]:4.0, bnd0[5]:3.0 )$
(%i3) /* Vertices of boundary #1: {(4,3),(5,4),(6,4),(5,1)} */
      ( bnd1: make_array(flonum,8),
        bnd1[0]:4.0, bnd1[1]:3.0, bnd1[2]:5.0, bnd1[3]:4.0,
        bnd1[4]:6.0, bnd1[5]:4.0, bnd1[6]:5.0, bnd1[7]:1.0 )$
(%i4) /* Vertices of boundary #2: {(5,1), (3,0), (1,1)} */
      ( bnd2: make_array(flonum,6),
        bnd2[0]:5.0, bnd2[1]:1.0, bnd2[2]:3.0,
        bnd2[3]:0.0, bnd2[4]:1.0, bnd2[5]:1.0 )$
(%i5) /* Vertices of boundary #3: {(1,1), (4,3)} */
      ( bnd3: make_array(flonum,4),
        bnd3[0]:1.0, bnd3[1]:1.0, bnd3[2]:4.0, bnd3[3]:3.0 )$
(%i6) /* Vertices of boundary #4: {(4,3), (5,1)} */
      ( bnd4: make_array(flonum,4),
        bnd4[0]:4.0, bnd4[1]:3.0, bnd4[2]:5.0, bnd4[3]:1.0 )$
(%i7) /* Pack all together in boundaries_array */
      ( boundaries_array: make_array(any,5),
        boundaries_array[0]: bnd0, boundaries_array[1]: bnd1,
        boundaries_array[2]: bnd2, boundaries_array[3]: bnd3,
        boundaries_array[4]: bnd4 )$
(%i8) draw2d(geomap([0,1,2,3,4]))$
```

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 `draw` no carga automáticamente `worldmap`.

```
(%i1) load(draw)$
(%i2) load(worldmap)$
(%i3) c1: gr2d(geomap(Canada,United_States,
                         Mexico,Cuba))$
(%i4) c2: gr2d(geomap(Africa))$
(%i5) c3: gr2d(geomap(Oceania,China,Japan))$
(%i6) c4: gr2d(geomap(France,Portugal,Spain,
                         Morocco,Western_Sahara))$
(%i7) draw(columns = 2,
            c1,c2,c3,c4)$
```

`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(draw)$
(%i2) load(worldmap)$
(%i3) mymap: append(
    [color      = white], /* borders are white */
    [fill_color = red],   make_poly_country(Bolivia),
    [fill_color = cyan],  make_poly_country(Paraguay),
    [fill_color = green], make_poly_country(Colombia),
    [fill_color = blue],  make_poly_country(Chile),
    [fill_color = "#23ab0f"], make_poly_country(Brazil),
    [fill_color = goldenrod], make_poly_country(Argentina),
    [fill_color = "midnight-blue"], make_poly_country(Uruguay))$
(%i4) apply(draw2d, mymap)$
```

3D

`geomap (numlist)` 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(draw)$
(%i2) load(worldmap)$
(%i3) draw3d(geomap(Australia), /* default projection */
             geomap(Australia,
                   [spherical_projection,2,2,2,3]))$
```

- `[cylindrical_projection,x,y,z,r,rc]`: re-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(draw)$
(%i2) load(worldmap)$
(%i3) draw3d(geomap([America_coastlines,Eurasia_coastlines],
[cylindrical_projection,2,2,2,3,4]))$
```

- `[conic_projection,x,y,z,r,alpha]`: re-proyecta mapas esféricos sobre los conos de ángulo α , 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(draw)$
(%i2) load(worldmap)$
(%i3) draw3d(geomap(World_coastlines,
[conic_projection,0,0,0,1,90]))$
```

En <http://www.telefonica.net/web2/biomates/maxima/gpdraw/geomap> hay ejemplos más elaborados.

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*: `xu_grid`, `yv_grid`, `line_type`, `line_width`, `key`, `enhanced3d` y `color`.

Ejemplo:

```
(%i1) load(draw)$
(%i2) draw3d(title      = "Sea shell",
              xu_grid    = 100,
              yv_grid    = 25,
              rot_vertical = 100,
              rot_horizontal = 20,
              surface_hide = true,
              parametric_surface(0.5*u*cos(u)*(cos(v)+1),
                                 0.5*u*sin(u)*(cos(v)+1),
                                 u*sin(v) - ((u+3)/8*pi)^2 - 20,
                                 u, 0, 13*pi, v, -pi, pi))$
```

draw (gr2d, ..., gr3d, ..., options, ...)

Función

Representa gráficamente una serie de escenas; sus argumentos son objetos `gr2d` y/o `gr3d`, junto con algunas opciones. Por defecto, las escenas se representan en una columna.

La función `draw` acepta las siguientes opciones globales: `terminal`, `columns`, `pic_width`, `pic_height`, `eps_width`, `eps_height`, `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.

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, **pic_width**, **pic_height**, **eps_width**, **eps_height**, **file_name** y **file_bgcolor**.

Ejemplo:

```
(%i1) load(draw)$
(%i2) /* dibujo en pantalla */
      draw(gr3d(explicit(x^2+y^2,x,-1,1,y,-1,1)))$
(%i3) /* mismo dibujo en formato eps */
      draw_file(terminal = eps,
                 eps_width = 5,
                 eps_height = 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 gráfico a la misma ventana, sin borrar los anteriores. Para desactivar el modo de gráficos múltiples escribase **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)$
```

48.3 Funciones y variables para picture

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 elementos:

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; las 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)})
```

picturep (x)

Función

Devuelve **true** si el argumento es una imagen bien formada, o **false** en caso contrario.

picture_equalp (x,y)

Función

Devuelve **true** si los dos argumentos son imágenes idénticas, o **false** en caso contrario.

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 elementos:

1. el símbolo **rgb**
2. anchura de la imagen
3. altura de la imagen
4. un array de enteros de $3 * \text{ancho} * \text{alto}$ 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)})
```

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)})
```

negative_picture (pic)

Función

Devuelve el negativo de la imagen, sea ésta de tipo nivel (*level*) o color (*rgb*).**rgb2level (pic)**

Función

Transforma una imagen en color *rgb* a otra de niveles *level* promediando los niveles.**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.**read_xpm (xpm_file)**

Función

Lee el fichero gráfico en formato xpm y devuelve un objeto `picture`.

48.4 Funciones y variables para worldmap

region_boundaries (x1,y1,x2,y2)

Función

Detecta los segmentos poligonales almacenados en la variable global `boundaries_array` 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(draw)$
(%i2) load(worldmap)$
(%i3) region_boundaries(10.4,41.5,20.7,35.4);
(%o3)          [1846, 1863, 1864, 1881, 1888, 1894]
(%i4) draw2d(geomap(%))$
```

numbered_boundaries (nlist)

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(draw)$
(%i2) load(worldmap)$
(%i3) european_borders:
        region_boundaries(-31.81,74.92,49.84,32.06)$
(%i4) numbered_boundaries(european_borders)$
```

make_polygon (nlist)

Función

Devuelve un objeto **polygon** a partir 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 poligonales 171, 173 y 1143 (en este orden) son compatibles y el polígono coloreado se podrá dibujar.

```
(%i1) load(draw)$
(%i2) load(worldmap)$
(%i3) Bhutan;
(%i3)                                [[171, 173, 1143]]
(%i4) boundaries_array[171];
(%o4) {Array:
      #(88.750549 27.14727 88.806351 27.25305 88.901367 27.282221
       88.917877 27.321039)}
(%i5) boundaries_array[173];
(%o5) {Array:
      #(91.659554 27.76511 91.6008 27.66666 91.598022 27.62499
       91.631348 27.536381 91.765533 27.45694 91.775253 27.4161
       92.007751 27.471939 92.11441 27.28583 92.015259 27.168051
       92.015533 27.08083 92.083313 27.02277 92.112183 26.920271
       92.069977 26.86194 91.997192 26.85194 91.915253 26.893881
       91.916924 26.85416 91.8358 26.863331 91.712479 26.799999
       91.542191 26.80444 91.492188 26.87472 91.418854 26.873329
       91.371353 26.800831 91.307457 26.778049 90.682457 26.77417
       90.392197 26.903601 90.344131 26.894159 90.143044 26.75333
       89.98996 26.73583 89.841919 26.70138 89.618301 26.72694
       89.636093 26.771111 89.360786 26.859989 89.22081 26.81472
       89.110237 26.829161 88.921631 26.98777 88.873016 26.95499
       88.867737 27.080549 88.843307 27.108601 88.750549
       27.14727)}
(%i6) boundaries_array[1143];
(%o6) {Array:
      #(91.659554 27.76511 91.666924 27.88888 91.65831 27.94805
```

```

91.338028 28.05249 91.314972 28.096661 91.108856 27.971109
91.015808 27.97777 90.896927 28.05055 90.382462 28.07972
90.396088 28.23555 90.366074 28.257771 89.996353 28.32333
89.83165 28.24888 89.58609 28.139999 89.35997 27.87166
89.225517 27.795 89.125793 27.56749 88.971077 27.47361
88.917877 27.321039) }
(%i7) Bhutan_polygon: make_polygon([171,173,1143])$
(%i8) draw2d(Bhutan_polygon)$

```

make_poly_country (*country_name*)

Función

Construye los polígonos necesarios para dibujar un país coloreado. En caso de tener islas, un país tendrá asociados varios polígonos.

Ejemplo:

```

(%i1) load(draw)$
(%i2) load(worldmap)$
(%i3) make_poly_country(India)$
(%i4) apply(draw2d, %)$

```

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(draw)$
(%i2) load(worldmap)$
(%i3) /* A continent */
      make_poly_continent(Africa)$
(%i4) apply(draw2d, %)$
(%i5) /* A list of countries */
      make_poly_continent([Germany,Denmark,Poland])$
(%i6) apply(draw2d, %)$

```


49 dynamics

49.1 Introducción a dynamics

El paquete adicional **dynamics** incluye varias funciones para crear diversas representaciones gráficas de sistemas dinámicos y fractales, y además una implementación del método numérico de Runge-Kutta de cuarto orden, para resolver sistemas de ecuaciones diferenciales.

Para usar las funciones en este paquete será necesario primero que todo cargarlo con `load("dynamics")`, y las funciones que crean gráficas necesitan que Xmaxima esté instalado.

49.2 Funciones y variables para dynamics

chaosgame ($[[x_1, y_1] \dots [x_m, y_m]], [x_0, y_0], b, n, \dots \text{opciones...}$); Función

Usa el método llamado juego del caos, para producir fractales: se dibuja un punto inicial (x_0, y_0) y luego se elige aleatoriamente uno de los m puntos $[x_1, y_1] \dots [x_m, y_m]$. 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.

evolution ($F, y_0, n, \dots \text{opciones...}$); Función

Dibuja $n+1$ puntos en una gráfica bidimensional (serie de tiempo), en que las coordenadas horizontales de los puntos son los números enteros $0, 1, 2, \dots, n$, 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 y_0 . F deberá ser una expresión que dependa únicamente de la variable y (y no de n), y_0 deberá ser un número real y n un número entero positivo.

evolution2d ($[F, G], [x_0, y_0], n, \dots \text{opciones...}$); 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 x_0 y y_0 . F y G deben ser dos expresiones que dependan únicamente de x y y .

ifs ($[r_1, \dots, r_m], [A_1, \dots, A_m], [[x_1, y_1] \dots [x_m, y_m]], [x_0, y_0], n, \dots \text{opciones...}$); 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 A_i 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 r_1, \dots, r_m . 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 r_1 , r_2 y r_3 podrían ser 2, 7 y 10, o cualquier otro grupo de números que tengan la misma proporción.

julia (x, y, \dots opciones...)

Función

Crea un fichero gráfico con la representación del conjunto de Julia del número complejo $(x + i y)$. Los parámetros x y y deben ser reales. El fichero se crea en el directorio actual o en el directorio del usuario, usando el formato gráfico XPM. El programa puede demorar varios segundos a ser ejecutado y cuando termina imprime un mensaje con el nombre del fichero creado.

Se asignan diferentes colores a los puntos que no pertenecen al conjunto de Julia, de acuerdo con el número de iteraciones que demore la secuencia, comenzando en ese punto, a salir fuera del círculo de convergencia con radio 2. El número máximo de iteraciones se define con la opción *levels*; después de ejecutadas ese número de iteraciones, si la secuencia aun está dentro del círculo de convergencia, el punto será coloreado con el color definido por la opción *color*.

Todos los colores usados para los puntos que no pertenecen al conjunto de Julia tendrán los mismos valores de saturación (*saturation*) y valor (*value*), pero con diferentes ángulos de tonalidad, distribuidos uniformemente en el intervalo entre *hue* y (*hue* + *huerange*).

Se puede dar a la función una secuencia de opciones. La lista de posibles opciones aparece en una sección más al frente.

mandelbrot (*options*)

Función

Crea un fichero gráfico con la representación del conjunto de Mandelbrot. El fichero se crea en el directorio actual o en el directorio del usuario, usando el formato gráfico XPM. El programa puede demorar varios segundos a ser ejecutado y cuando termina imprime un mensaje con el nombre del fichero creado.

Se asignan diferentes colores a los puntos que no pertenecen al conjunto de Mandelbrot, de acuerdo con el número de iteraciones que demore la secuencia generada por ese punto a salir fuera del círculo de convergencia con radio 2. El número máximo de iteraciones se define con la opción *levels*; después de ejecutadas ese número de iteraciones, si la secuencia aun está dentro del círculo de convergencia, el punto será coloreado con el color definido por la opción *color*.

Todos los colores usados para los puntos que no pertenecen al conjunto de Mandelbrot tendrán los mismos valores de saturación (*saturation*) y valor (*value*), pero con diferentes ángulos de tonalidad, distribuidos uniformemente en el intervalo entre *hue* y (*hue* + *huerange*).

Se puede dar a la función una secuencia de opciones. La lista de posibles opciones aparece en una sección más al frente.

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 $y(n1+1), \dots, y(n1+n2+1)$, obtenidos después de dejarla evolucionar durante $n1$ iteraciones iniciales.

rk (EDO , var, inicial, dominio) Función

rk ([$EDO1, \dots, EDOm$], [$v1, \dots, vm$], [$inic1, \dots, inicm$], 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 dependiente, 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 $v1, v2, \dots, vm$. Los valores iniciales para esas variables serán $inic1, inic2, \dots, inicm$. Continuará existiendo apenas una variable independiente definida por la lista domain, como en el caso anterior. $EDO1, \dots, EDOm$ 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 $EDO1, \dots, EDOm$ 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.

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) .

Opciones

Cada opción es una lista con dos o más elementos. El primer elemento en la lista es el nombre de la opción y el resto consiste en los argumentos para esa opción.

Las opciones aceptadas por las funciones **evolution**, **evolution2**, **staircase**, **orbits**, **ifs** y **chaosgame** son las siguientes:

- **domain** especifica los valores mínimo y máximo de la variable independiente para la gráfica de la función F representada por **staircase**.
- **pointsize** define el radio de cada punto dibujado, en unidades de puntos. El valor por omisión es 1.
- **xaxislabel** es la etiqueta que se escribirá en el eje horizontal.
- **xcenter** es la coordenada x del punto que deberá aparecer en el centro de la gráfica. Esta opción no es usada por la función **orbits**.
- **xradius** es mitad de la longitud del intervalo de valores de x que será representado. Esta opción no es usada por la función **orbits**.
- **yaxislabel** es la etiqueta que se escribirá en el eje vertical.
- **ycenter** es la coordenada y del punto que deberá aparecer en el centro de la gráfica.
- **yradius** es mitad de la longitud del intervalo de valores de y que será representado.

Las opciones aceptadas por los programas **julia** y **mandelbrot** son las siguientes:

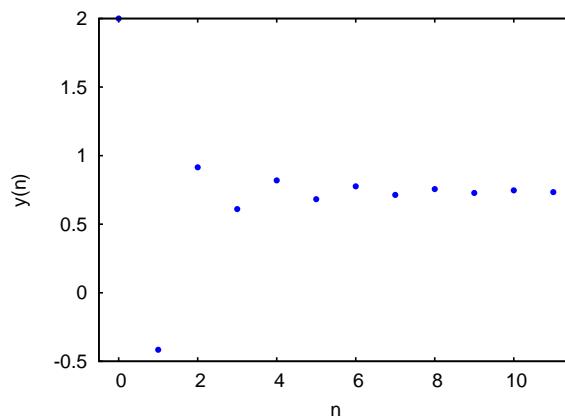
- **size** acepta uno o dos argumentos. Si se da solo un argumento, el ancho y la altura del fichero gráfico creado serán iguales a ese valor en pixels. Si se dan dos argumentos, esos dos valores serán usados para el ancho y la altura. El valor por omisión es 400 pixels tanto para el ancho como para la altura. Si los dos valores no son iguales, el conjunto aparecerá distorsionado.
- **levels** define el número máximo de iteraciones, que es también el número de colores usado para los puntos que no pertenecen al conjunto. El valor por omisión es 12; valores mayores implican tiempos de procesamiento más elevados.
- **huerange** define el intervalo de ángulos usados para la tonalidad de los puntos que no pertenecen al conjunto. El valor por omisión es 360, que hace que los colores usados abarcarán todo el rango de tonalidades. Valores mayores que 360 implican repetición de algunos valores de la tonalidad, y pueden usarse valores negativos para que el ángulo de tonalidad sea menor a medida que el número de iteraciones aumente.

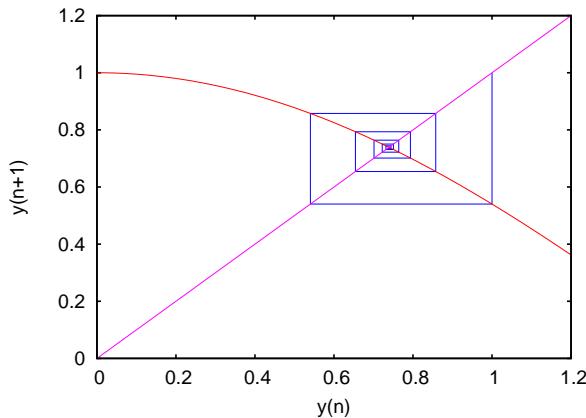
- *hue* define la tonalidad, en grados, del primer color usado para representar los puntos que no pertenecen al conjunto. Su valor por omisión es 300 grados, que corresponde al color magenta; el color correspondiente para otros valores estandar es rojo 0, naranja 45, amarillo 60, verde 120, cian 180 y azul 240. Consulte también la opción *huerange*.
- *saturation* define el nivel de saturación que será usado para los colores de los puntos que no pertenecen al conjunto. Debe ser un valor entre 0 y 1. El valor por omisión es 0.46.
- *value* define el valor de los colores usados para puntos que no pertenezcan al conjunto. Debe estar comprendido entre 0 y 1; cuanto mayor sea, mas brillantes serán los colores. Su valor por omisión es 0.96.
- *color* debe ir seguido de tres parámetros que definen la tonalidad, saturación y valor del color que será usado para los puntos del conjunto. El valor por omisión es 0 para los tres parámetros, que corresponde al negro. Consulte las explicación sobre el rango de valores aceptados en la explicación de las opciones *hue*, *saturation* y *value*.
- *center* deberá tener dos parámetros reales que dan las coordenadas, en el plano complejo, del punto en el centro de la región representada. El valor por omisión es 0 para las dos coordenadas (el origen).
- *radius* es el radio de el mayor círculo que cabe dentro de la región cuadrada que será representada. El valor por omisión es 2.
- *filename* da el nombre del fichero donde se guardará la gráfica producida. A ese nombre se le acrecentará la terminación .xpm. Si el fichero ya existe, será substituido por el fichero producido por la función. El valor por omisión es julia para el conjunto de Julia y mandelbrot para el conjunto de Mandelbrot.

Ejemplos

Representación gráfica y diagrama de escalera de la secuencia: $2, \cos(2), \cos(\cos(2)), \dots$

```
(%i1) load("dynamics")$  
(%i2) evolution(cos(y), 2, 11, [yaxislabel, "y"], [xaxislabel,"n"]);  
(%i3) staircase(cos(y), 1, 11, [domain, 0, 1.2]);
```



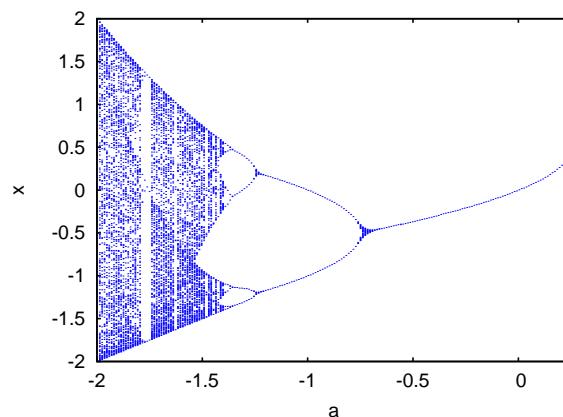


Si su procesador es lento, tendrá que reducir el número de iteraciones usado en los ejemplos siguientes. Y el valor de *pointsize* que da mejores resultados depende del monitor y de la resolución que use. Tendrá que experimentar con diferentes valores.

Diagrama de órbitas para el mapa cuadrático

$$y_{n+1} = x + y_n^2$$

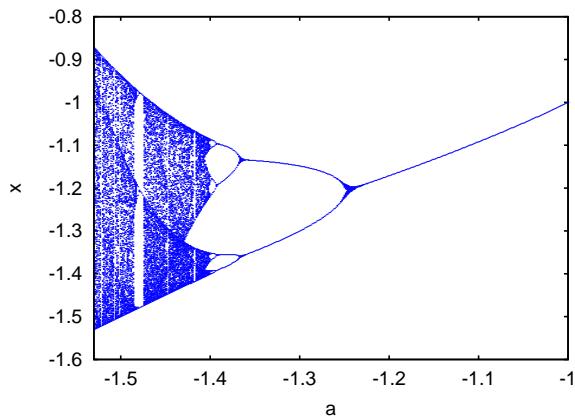
```
(%i4) orbits(y^2+x, 0, 50, 200, [x, -2, 0.25, 0.01], [pointsize, 0.9]);
```



Para ampliar la región alrededor de la bifurcación en la parte de abajo, cerca de $x = -1.25$, use el comando:

```
(%i5) orbits(x+y^2, 0, 100, 400, [x,-1,-1.53,-0.001], [pointsize,0.9],
```

```
[ycenter,-1.2], [yradius,0.4]);
```

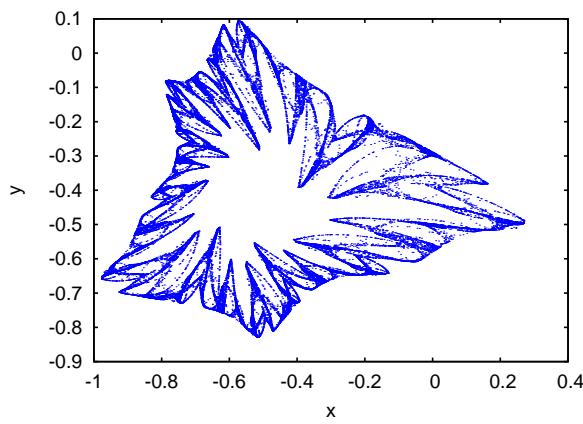


Evolución de un sistema en dos dimensiones, que conduce a un fractal:

```
(%i6) f: 0.6*x*(1+2*x)+0.8*y*(x-1)-y^2-0.9$  

(%i7) g: 0.1*x*(1-6*x+4*y)+0.1*y*(1+9*y)-0.4$  

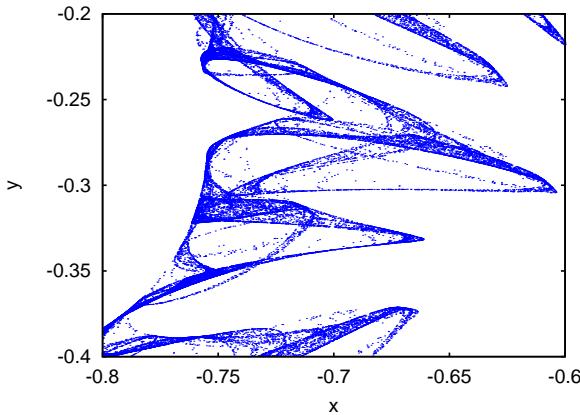
(%i8) evolution2d([f,g],[-0.5,0],50000,[pointsize,0.7]);
```



Y una ampliación de una pequeña región en el fractal:

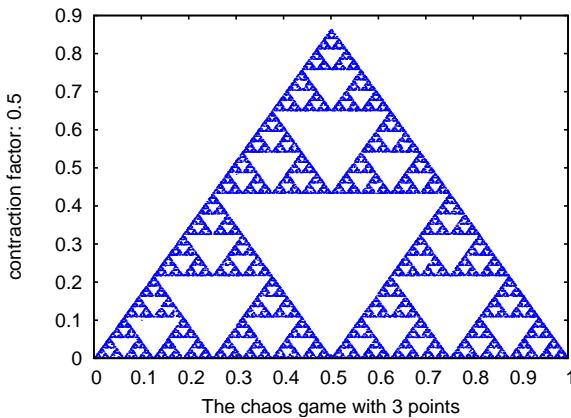
```
(%i9) evolution2d([f,g],[-0.5,0],300000,[pointsize,0.7],[xcenter,-0.7],
```

```
[ycenter,-0.3],[xradius,0.1],[yradius,0.1]);
```



Una gráfica del triangulo de Sierpinsky, obtenida con el juego del caos:

```
(%i9) chaosgame([[0, 0], [1, 0], [0.5, sqrt(3)/2]], [0.1, 0.1], 1/2,
30000, [pointsize,0.7]);
```



El helecho de Barnsley, obtenido con el Sistema de Funciones Iteradas:

```
(%i10) a1: matrix([0.85,0.04],[-0.04,0.85])$  

(%i11) a2: matrix([0.2,-0.26],[0.23,0.22])$  

(%i12) a3: matrix([-0.15,0.28],[0.26,0.24])$  

(%i13) a4: matrix([0,0],[0,0.16])$  

(%i14) p1: [0,1.6]$  

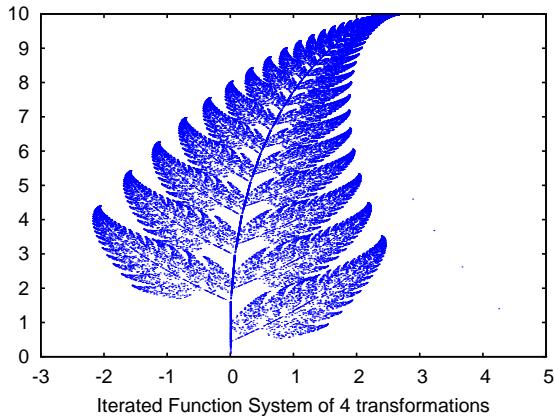
(%i15) p2: [0,1.6]$  

(%i16) p3: [0,0.44]$  

(%i17) p4: [0,0]$  

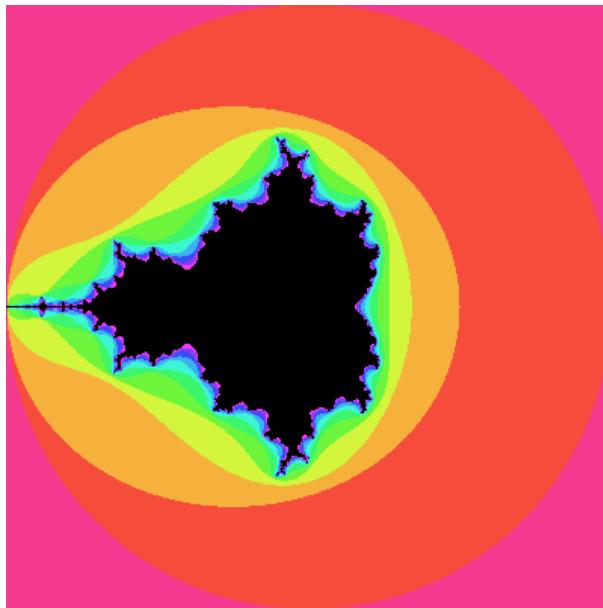
(%i18) w: [85,92,99,100]$
```

```
(%i19) ifs(w,[a1,a2,a3,a4],[p1,p2,p3,p4],[5,0],50000,[pointsize,0.9]);
```



Para crear un fichero llamado *dinamica9.xpm* con la representación gráfica del conjunto de Mandelbrot, con 12 colores, use el comando:

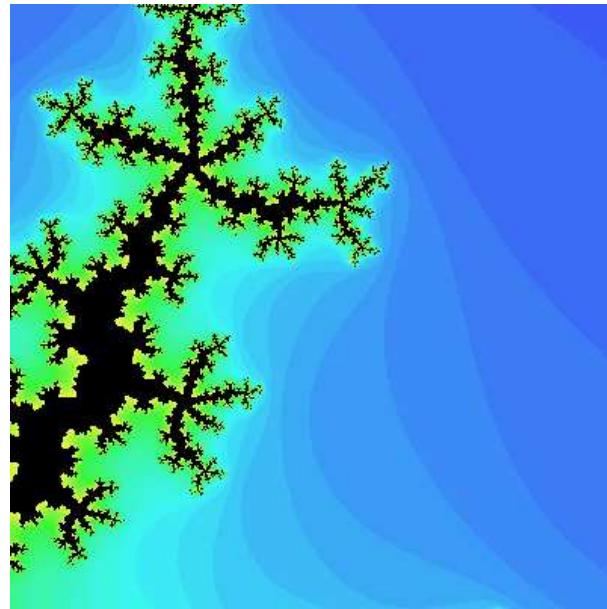
```
mandelbrot([filename,"dinamica9"]);$
```



y el conjunto de Julia del número $(-0.55 + i 0.6)$ puede ser obtenido con:

```
julia(-0.55, 0.6, [levels, 36], [center, 0, 0.6], [radius, 0.3],  
[hue, 240], [huerange, -180], [filename, "dinamica10"]);$
```

la gráfica se guardará en el fichero *dinamica10.xpm* y mostrará la región desde -0.3 hasta 0.3 en la dirección x, y desde 0.3 hasta 0.9 en la dirección y. Serán usados 36 colores, comenzando con azul e terminando con amarillo.



Para resolver numéricamente la ecuación diferencial

$$\frac{dx}{dt} = t - x^2$$

Con valor inicial $x(t=0) = 1$, en el intervalo de t desde 0 hasta 8, y con incrementos de 0.1, se usa:

(%i20) resultados: rk($t-x^2, x, 1, [t, 0, 8, 0.1]$)\$

los resultados quedarán guardados en la lista resultados.

Para resolver numéricamente el sistema:

$$\begin{cases} \frac{dx}{dt} = 4 - x^2 - 4y^2 \\ \frac{dy}{dt} = y^2 - x^2 + 1 \end{cases}$$

para t entre 0 y 4, con valores iniciales -1.25 y 0.75 para (x, y) en $t=0$:

(%i21) sol: rk([$4-x^2-4*y^2, y^2-x^2+1$], [x, y], [-1.25, 0.75], [t, 0, 4, 0.02])\$

50 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 `demos(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 $(x * y)^{a+b}$.

$(x^a) + (y^b)$ es igual a $(x + y)^{a+b}$.

$(x^a)^b$ es igual a $x^{a \cdot b}$ 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:

<code>%c</code>	velocidad de la luz en el vacío
<code>%mu_0</code>	constante magnética
<code>%e_0</code>	constante eléctrica
<code>%Z_0</code>	impedancia característica del vacío
<code>%G</code>	constante gravitatoria de Newton
<code>%h</code>	constante de Planck
<code>%h_bar</code>	constante de Planck
<code>%m_P</code>	masa de Planck
<code>%T_P</code>	temperatura de Planck
<code>%l_P</code>	longitud de Planck
<code>%t_P</code>	tiempo de Planck
<code>%%e</code>	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);
(%o4)          299792458 ' - m
                           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;
(%o2)          %c^2 m
(%i3) %, m = 1 ' lbm;
(%o3)          %c^2 ' lbm
(%i4) constvalue (%);
(%o4)          2
                           lbm m
                           ----- ' -----
                           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.

$(x \cdot a) * (y \cdot b)$ es igual a $(x * y) \cdot (a * b)$.

$(x \cdot a) + (y \cdot a)$ es igual a $(x + y) \cdot a$.

$(x \cdot a)^y$ es igual a $x^y \cdot a^y$ 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);
(%o3)                               length
(%i4) fundamental_units (bar);
(%o4)                               2
                               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;
(%o3)                                0 ' m
(%i4) (10 ' kg) * (17 ' m/s^2);
(%o4)                                kg m
                                         170 ' -----
                                         2
                                         s
(%i5) (x ' m) / (y ' s);
(%o5)                                - ' -
                                         x   m
                                         y   s
(%i6) (a ' m)^2;
(%o6)                                2      2
                                         a   ' m
```

“

Operador

Operador de conversión de unidades. Una expresión tal como $a \cdot b^c$ 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,galloon = 757*1/200,GHz = 1000000000*Hz,
       GOhm = 1000000000*Ohm,GPa = 1000000000*Pa,
       GWb = 1000000000*Wb,Gg = 1000000*kg,Gm = 1000000000*m,
       Gmol = 1000000*mol,Gs = 1000000000*s,ha = hectare,
       hectare = 100*m^2,hour = 3600*s,Hz = 1/s,inch = feet/12,
       km = 1000*m,kmol = 1000*mol,ks = 1000*s,l = liter,
```

```

lbf = pound_force,lbm = pound_mass,liter = m^3/1000,
metric_ton = Mg,mg = kg/1000000,MHz = 1000000*Hz,
microgram = kg/1000000000,micrometer = m/1000000,
micron = micrometer,microsecond = s/1000000,
mile = 5280*feet,minute = 60*s,mm = m/1000,
mmol = mol/1000,month = 2629800*s,MOhm = 1000000*Ohm,
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.
                                         381
(%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;
(%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;
(%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;

```

```
(%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

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.

Para utilizar estas funciones ejecútese primero **load(ezunits)**.

Ejemplos:

Valor de una constante física.

```
(%i1) load (physical_constants)$
(%i2) constvalue (%G);
(%o2) 6.67428 ' -----
3
m
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);
(%o2) 100 ' -----
1bm
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;
(%o3)                               x ` -m
                                         s
(%i4) units (foo);
(%o4)                               kg
(%i5) units (bar);
(%o5)                               -m
                                         s
(%i6) units (foo * bar);
(%o6)                               kg m
                                         -----
                                         s
(%i7) units (foo / bar);
(%o7)                               kg s
                                         -----
                                         m
(%i8) units (foo^2);
(%o8)                               kg^2
```

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);
(%o4)                               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;
(%o4) v '  $\frac{m}{s}$ 
(%i5) foo * bar;
(%o5) 100 v '  $\frac{kg \cdot m}{s}$ 
(%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);
(%o4) xx2
(%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 opción 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.
```

```
(%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 ($a_1, d_1, \dots, a_n, d_n$)
remove_dimensions (a_1, \dots, a_n)

Función
Función

declare_dimensions declara a_1, \dots, a_n con las dimensiones d_1, \dots, 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, \dots)
remove_fundamental_dimensions (d_1, d_2, d_3, \dots)
fundamental_dimensions

Función
Función
Variable global

declare_fundamental_dimensions declara dimensiones fundamentales. Los símbolos d_1, d_2, d_3, \dots 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, \dots, u_n de dimensiones d_1, \dots, 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, \dots, u_n , y `fundamental_units(d_k)` devuelve u_k para cada d_1, \dots, 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
(%o6) -----
                  dollar goat
                  -----
                  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
           -
           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
(%o2) [---]
           y z
           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
(%o3) [-----, -----]
           %%e           %m_e
           %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.

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 ggf

52.1 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")`.

53 graphs

53.1 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 múltiples 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 identificación (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)`.

53.2 Funciones y variables para graphs

53.2.1 Construyendo grafos

<code>create_graph (v_list, e_list)</code>	Función
<code>create_graph (n, e_list)</code>	Función
<code>create_graph (v_list, e_list, directed)</code>	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 (`[[v1,11], [v2,12], ..., [vn,1n]]`).

`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 <i>g</i> .	
complete_bipartite_graph (n, m)	Función
Devuelve el grafo bipartido completo de <i>n+m</i> vértices.	
complete_graph (n)	Función
Devuelve el grafo completo de <i>n</i> vértices.	
cycle_digraph (n)	Función
Devuelve el ciclo dirigido de <i>n</i> vértices.	
cycle_graph (n)	Función
Devuelve el ciclo de <i>n</i> vértices.	
cube_graph (n)	Función
Devuelve el cubo de <i>n</i> dimensiones.	
dodecahedron_graph ()	Función
Devuelve el grafo del dodecaedro.	
empty_graph (n)	Función
Devuelve el grafo vacío de <i>n</i> vértices.	
flower_snark (n)	Función
Devuelve el grafo de flor de <i>4n</i> 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 <i>A</i> .	
frucht_graph ()	Función
Devuelve el grafo de Frucht.	
graph_product (g1, g2)	Función
Devuelve el producto dirigido de los grafos <i>g1</i> y <i>g2</i> .	
Ejemplo:	
(%i1) load (graphs)\$ (%i2) grid : graph_product(path_graph(3), path_graph(4))\$ (%i3) draw_graph(grid)\$	

graph_union (<i>g1, g2</i>)	Función
Devuelve la unión (suma) de los grafos <i>g1</i> y <i>g2</i> .	
grid_graph (<i>n, m</i>)	Función
Devuelve la rejilla <i>n</i> x <i>m</i> .	
grotzsch_graph ()	Función
Devuelve el grafo de Grotzsch.	
heawood_graph ()	Función
Devuelve el grafo de Heawood.	
icosahedron_graph ()	Función
Devuelve el grafo del icosaedro.	
induced_subgraph (<i>V, g</i>)	Función
Devuelve el grafo inducido por el subconjunto <i>V</i> de vértices del grafo <i>g</i> .	
Ejemplo:	
<pre>(%i1) load(graphs)\$ (%i2) p : petersen_graph()\$</pre>	
<pre>(%i3) V : [0,1,2,3,4]\$</pre>	
<pre>(%i4) g : induced_subgraph(V, p)\$</pre>	
<pre>(%i5) print_graph(g)\$</pre>	
Graph on 5 vertices with 5 edges.	
Adjacencies:	
<pre>4 : 3 0 3 : 2 4 2 : 1 3 1 : 0 2 0 : 1 4</pre>	
line_graph (<i>g</i>)	Función
Devuelve el grafo de línea del grafo <i>g</i> .	
make_graph (<i>vrt, f</i>)	Función
make_graph (<i>vrt, f, oriented</i>)	Función
Crea un grafo por medio de la función de predicado <i>f</i> .	
<i>vrt</i> es una lista o conjunto de vértices o un simplemente un número entero. Si <i>vrt</i> es un número entero, entonces los vértices del grafo serán los enteros desde 1 hasta <i>vrt</i> .	
<i>f</i> es una función de predicado. Dos vértices <i>a</i> y <i>b</i> se conectarán si <i>f(a,b)=true</i> .	
Si <i>directed</i> no es <i>false</i> , entonces en grafo será dirigido.	
Ejemplo 1:	
<pre>(%i1) load(graphs)\$ (%i2) g : make_graph(powerset({1,2,3,4,5}, 2), disjointp)\$</pre>	
<pre>(%i3) is_isomorphic(g, petersen_graph());</pre>	

```
(%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.

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.

53.2.2 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);  

(%o3)
```

$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 & 0 & 1 \\ [] & [] & [] & [] \\ 1 & 0 & 1 & 0 \\ [] & [] & [] & [] \\ 0 & 1 & 0 & 1 \\ [] & [] & [] & [] \\ 1 & 0 & 1 & 0 \end{bmatrix}$
--	---

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
```

biconected_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 (<i>v, gr</i>)	Función
Devuelve la etiqueta del vértice <i>v</i> del grafo <i>gr</i> .	
Ejemplo:	
(%i1) load (graphs)\$ (%i2) g : create_graph([[0,"Zero"], [1, "One"]], [[0,1]])\$ (%i3) get_vertex_label(0, g); (%o3)	Zero
graph_charpoly (<i>gr, x</i>)	Función
Devuelve el polinomio característico (de variable <i>x</i>) del grafo <i>gr</i> .	
Ejemplo:	
(%i1) load (graphs)\$ (%i2) p : petersen_graph()\$ (%i3) graph_charpoly(p, x), factor;	
(%o3)	$(x - 3)^5 (x - 1)^4 (x + 2)$
graph_center (<i>gr</i>)	Función
Devuelve el centro del grafo <i>gr</i> .	
Ejemplo:	
(%i1) load (graphs)\$ (%i2) g : grid_graph(5,5)\$ (%i3) graph_center(g); (%o3)	[12]
graph_eigenvalues (<i>gr</i>)	Función
Devuelve los valores propios del grafo <i>gr</i> . 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 (<i>gr</i>)	Función
Devuelve la periferia del grafo <i>gr</i> .	
Ejemplo:	
(%i1) load (graphs)\$ (%i2) g : grid_graph(5,5)\$ (%i3) graph_periphery(g); (%o3)	[24, 20, 4, 0]
graph_size (<i>gr</i>)	Función
Devuelve el número de aristas del grafo <i>gr</i> .	
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 ]
          [                   ]
          [ 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 (<i>gr</i>)	Función
Devuelve el mínimo nodo <i>covering</i> del grafo <i>gr</i> .	
min_vertex_cut (<i>gr</i>)	Función
Devuelve el mínimo <i>vertex cut</i> del grafo <i>gr</i> .	
Véase también <i>vertex_connectivity</i> .	
minimum_spanning_tree (<i>gr</i>)	Función
Devuelve el grafo de expansión mínimo del grafo <i>gr</i> .	
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 (<i>v, gr</i>)	Función
Devuelve la lista de los vecinos del vértice <i>v</i> del grafo <i>gr</i> .	
Ejemplo:	
(%i1) load (graphs)\$ (%i2) p : petersen_graph()\$ (%i3) neighbors(3, p); (%o3) [4, 8, 2]	
odd_girth (<i>gr</i>)	Función
Devuelve la longitud del ciclo impar más corto del grafo <i>gr</i> .	
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 (<i>v, gr</i>)	Función
Devuelve la lista de los nodos padres del vértice <i>v</i> del grafo orientado <i>gr</i> .	
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_neighbours(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]
```

53.2.3 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]]]
```

53.2.4 Lectura y escritura de ficheros

dimacs_export (<i>gr, fl</i>)	Función
dimacs_export (<i>gr, fl, comment1, ..., commentn</i>)	Función
Exporta el grafo al fichero <i>fl</i> en formato DIMACS. Los comentarios adicionales se añadirán al comienzo del fichero.	
dimacs_import (<i>fl</i>)	Función
Lee el grafo almacenado en el fichero <i>fl</i> en formato DIMACS.	
graph6_decode (<i>str</i>)	Función
Devuelve el grafo codificado en formato graph6 en la cadena <i>str</i> .	
graph6_encode (<i>gr</i>)	Función
Devuelve una cadena codificando el grafo <i>gr</i> en formato graph6.	
graph6_export (<i>gr_list, fl</i>)	Función
Exporta los grafos de la lista <i>gr_list</i> al fichero <i>fl</i> en formato graph6.	
graph6_import (<i>fl</i>)	Función
Lee la lista de grafos almacenados en el fichero <i>fl</i> en formato graph6.	
sparse6_decode (<i>str</i>)	Función
Devuelve el grafo codificado en formato sparse6 en la cadena <i>str</i> .	
sparse6_encode (<i>gr</i>)	Función
Devuelve una cadena codificando el grafo <i>gr</i> en formato sparse6.	
sparse6_export (<i>gr_list, fl</i>)	Función
Exporta los grafos de la lista <i>gr_list</i> al fichero <i>fl</i> en formato sparse6.	
sparse6_import (<i>fl</i>)	Función
Lee la lista de grafos almacenados en el fichero <i>fl</i> en formato sparse6.	

53.2.5 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.

Los argumentos opcionales de la función `draw_graph` son:

- `show_id=show`: si `show` vale `true` entonces se muestran los números identificadores de los vértices.
- `show_label=show`: si `show` vale `true` entonces se muestran las etiquetas de los vértices.
- `label_alignment=pos`: indica cómo se deben alinear las etiquetas o números identificadores de los vértices. Puede ser: `left`, `center` or `right`. El valor por defecto es `left`.
- `show_weight=show`: si `show` vale `true` entonces se mostrarán los pesos de las aristas.
- `vertex_type=type`: establece cómo se mostrarán los vértices. Véase la opción `point_type` del paquete `draw`.
- `vertex_size=size`: tamaño de los vértices.
- `vertex_color=c`: color a utilizar en los vértices.
- `show_vertices=v_list`: dibuja los vértices de la lista `v_list` con colores diferentes.
- `show_vertex_type=type`: establece cómo se mostrarán los vértices de `show_vertices`. Véase la opción `point_type` del paquete `draw`.
- `show_vertex_size=size`: tamaños de los vértices de `show_vertices`.
- `show_vertex_color=c`: color a utilizar en los vértices de la lista `show_vertices`.
- `vertex_partition=part`: una partición $[[v_1, v_2, \dots], \dots, [v_k, \dots, v_n]]$ de los vértices del grafo. Los vértices de cada lista se dibujarán de diferente color.
- `vertex_coloring=col`: colores de los vértices. Los colores `col` deben especificarse en el mismo formato que el devuelto por `vertex_coloring`.
- `edge_color=c`: color a utilizar en las aristas.
- `edge_width=width`: ancho de las aristas.
- `edge_type=type`: establece cómo se dibujarán las aristas. Véase la opción `line_type` del paquete `draw`.
- `show_edges=e_list`: dibuja las aristas de la lista `e_list` con colores diferentes.
- `show_edge_color=c`: color a utilizar en las aristas de la lista `show_edges`.
- `show_edge_width=width`: anchos de las aristas de `show_edges`.
- `show_edge_type=type`: establece cómo se dibujarán las aristas de `show_edges`. Véase la opción `line_type` del paquete `draw`.
- `edge_partition=partition`: una partición $[[e_1, e_2, \dots], \dots, [e_k, \dots, e_m]]$ de las aristas del grafo. Las aristas de cada lista se dibujarán de diferente color.

- `edge_coloring=col`: colores de las aristas. Los colores `col` deben especificarse en el mismo formato que el devuelto por `edge_coloring`.
- `redraw=r`: 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=angle`: ángulo de las flechas de los arcos en los grafos orientados. Valor por defecto: 15.
- `head_length=len`: longitud de las flechas de los arcos en los grafos orientados. Valor por defecto: 0.1.
- `spring_embedding_depth=depth`: número de iteraciones del algoritmo de dibujo de grafos. Valor por defecto: 50.
- `terminal=term`: terminal utilizado para ver el gráfo. Véase la opción `terminal` del paquete `draw`.
- `file_name=file`: nombre del fichero cuando el terminal especificado no es la pantalla.
- `program=prg`: 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=[]`: especifica una lista de vértices con posiciones fijas en un polígono regular. Se puede utilizar cuando `program=spring_embedding`.

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,
show_id=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**.**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*.

54 grobner

54.1 Introducción a grobner

`grobner` es un paquete para operar con bases de Groebner en Maxima. Se puede encontrar un tutorial sobre *Bases de Groebner* en
<http://www.geocities.com/CapeCanaveral/Hall/3131/>
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.

54.1.1 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

54.1.2 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.

54.2 Funciones y variables para grobner

54.2.1 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 <i>maxima</i> . 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 <i>maxima</i> .	
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.

54.2.2 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.

54.2.3 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]);
(%o1)                               2      2
                           x - y
(%i2) poly_expand((y+x)^2,[x,y]);
(%o2)                               2      2
                           y + 2 x y + x
(%i3) poly_expand((y+x)^5,[x,y]);
(%o3)      5      4      2  3      3  2      4      5
           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]);
(%o4)
                           y      x
           - x %e + ----- - 1
                           sqrt(y)

(%i5) poly_expand(-1-sin(x)^2+sin(x),[sin(x)]);
(%o5)      2
           - 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.

54.2.4 Postprocesamiento estándar de bases de Groebner

El *k*-ésimo ideal de eliminación I_k de un ideal *I* sobre $K[x_1, \dots, x_n]$ es $I \cap K[x_{k+1}, \dots, x_n]$.

El ideal $I : J$ es el ideal $\{h \mid \forall w \in J : wh \in I\}$.

El ideal $I : p^\infty$ es el ideal $\{h \mid \exists n \in N : p^n h \in I\}$.

El ideal $I : J^\infty$ es el ideal $\{h \mid \exists n \in N, \exists p \in J : p^n h \in I\}$.

El ideal radical \sqrt{I} es el ideal $\{h \mid \exists n \in N : h^n \in I\}$.

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 sublistas 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 sublistas 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 (<i>polylist, varlist</i>)	Función
Devuelve la base de Groebner del ideal asociado a los polinomios de <i>polylist</i> . El resultado depende de las variables globales.	
poly_reduced_grobner (<i>polylist, varlist</i>)	Función
Devuelve la base de Groebner reducida del ideal asociado a los polinomios de <i>polylist</i> . El resultado depende de las variables globales.	
poly_depends_p (<i>poly, var, varlist</i>)	Función
<i>poly_depends</i> comprueba si el polinomio depende de la variable <i>var</i> .	
poly_elimination_ideal (<i>polylist, n, varlist</i>)	Función
<i>poly_elimination_ideal</i> devuelve la base de Groebner del <i>n</i> -ésimo ideal de eliminación de un ideal especificado como una lista de polinomios generadores (no necesariamente una base de Groebner).	
poly_colon_ideal (<i>polylist1, polylist2, varlist</i>)	Función
Devuelve la base de Groebner reducida del ideal $I(polylist1) : I(polylist2)$ siendo <i>polylist1</i> y <i>polylist2</i> dos listas de polinomios.	
poly_ideal_intersection (<i>polylist1, polylist2, varlist</i>)	Función
<i>poly_ideal_intersection</i> devuelve la intersección de dos ideales.	
poly_lcm (<i>poly1, poly2, varlist</i>)	Función
Devuelve el MCM de <i>poly1</i> y <i>poly2</i> .	
poly_gcd (<i>poly1, poly2, varlist</i>)	Función
Devuelve el MCD de <i>poly1</i> y <i>poly2</i> .	
poly_grobner_equal (<i>polylist1, polylist2, varlist</i>)	Función
<i>poly_grobner_equal</i> comprueba si dos bases de Groebner generan el mismo ideal. Devuelve true si dos listas de polinomios <i>polylist1</i> y <i>polylist2</i> , 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) <i>poly_grobner_equal</i> ([y+x,x-y],[x,y],[x,y]); (%o1) false	
poly_grobner_subsetp (<i>polylist1, polylist2, varlist</i>)	Función
<i>poly_grobner_subsetp</i> comprueba si el ideal generado por <i>polylist1</i> está contenido en el ideal generado por <i>polylist2</i> . Para que esta comprobación tenga éxito, <i>polylist2</i> 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

$$I(\text{polylist}) : \text{poly}^\infty$$

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

$$I(\text{polylist1}) : I(\text{polylist2})^\infty$$

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 polinomios [*poly1, ..., polyn*]. Devuelve la base de Groebner reducida del ideal

$$I(\text{polylist}) : \text{poly1}^\infty : \dots : \text{polyn}^\infty$$

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

$$I(\text{polylist}) : I(\text{polylist}_1)^\infty : \dots : I(\text{polylist}_n)^\infty$$

poly_saturation_extension (*poly, polylist, varlist1, varlist2*) Función

poly_saturation_extension ejecuta el truco de Rabinowitz.

poly_polysaturation_extension (*poly, polylist, varlist1, varlist2*) Función

55 impdiff

55.1 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")`.

56 interpol

56.1 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*'.

56.2 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, escríbese 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.

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):=%;
```

```

(%o4)   f(x) := -----
                           (x - 7) (x - 6) (x - 3) (x - 1)
                           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)
- -----

```

$$+ \frac{(w - 8)(w - 7)(w - 6)(w - 3)}{84}$$

charfun2 (*x, a, b*)

Función

Devuelve **true** si el número *x* pertenece al intervalo $[a, b]$, y **false** en caso contrario.**linearinterp** (*points*)

Función

linearinterp (*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, escribáse algo como *varname='z'*.

Ejemplos:

```
(%i1) load(interp)$
(%i2) p: matrix([7,2],[8,3],[1,5],[3,2],[6,7])$ 
(%i3) linearinterp(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]
```

```
(%i17) load(draw)$ /* load draw package */
(%i18) /* 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)))$ 
(%i19) /* Change variable name */
      linearinterp(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.

Ejemplos:

```
(%i1) load(interp)$
(%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, option2, ...*)

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 dos opciones para acomodarse a necesidades concretas:

- '**denterm**', por defecto 1, es el término independiente del polinomio en el denominador.
- '**varname**', por defecto '**x**', es el nombre de la variable independiente.

Ejemplos:

```

(%i1) load(interp)$
(%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])$
```

57 lapack

57.1 Introducción a lapack

lapack es una traducción automática a Common Lisp (con el programa `f2c`) de la librería LAPACK escrita en Fortran.

57.2 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)
(%i3) M : matrix ([9.5, 1.75], [3.25, 10.45]);
          [ 9.5  1.75 ]
          [             ]
(%o3)          [               ]
          [ 3.25  10.45 ]
(%i4) dgeev (M);
(%o4)           [[7.54331, 12.4067], false, false]
(%i5) [L, v, u] : dgeev (M, true, true);
          [ - .666642 - .515792 ]
```

```
(%o5) [[7.54331, 12.4067], [ .745378 - .856714 ]  
[ - .856714 - .745378 ]  
[ ]]  
[ .515792 - .666642 ]  
(%i6) D : apply (diag_matrix, L);  
[ 7.54331 0 ]  
(%o6) [ ]  
[ 0 12.4067 ]  
(%i7) M . v - v . D;  
[ 0.0 - 8.88178E-16 ]  
(%o7) [ ]  
[ - 8.88178E-16 0.0 ]  
(%i8) transpose (u) . M - D . transpose (u);  
[ 0.0 - 4.44089E-16 ]  
(%o8) [ ]  
[ 0.0 0.0 ]
```

dgesvd (*A*)

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^T*, tales que

$$A = U \cdot Sigma \cdot V^T$$

donde *U* es una matriz unitaria *m*-por-*m*, *Sigma* es una matriz diagonal *m*-por-*n* y *V^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 singulares por la izquierda cuando *left-p* = **true**, y los vectores singulares 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*)
zrange (*norm, A*)

Función
Función

Calcula una norma o seudonorma de la matriz *A*.

- | | |
|------------------|---|
| max | Calcula $\max(\text{abs}(A(i, j)))$, siendo <i>i</i> y <i>j</i> números de filas y columnas, respectivamente, de <i>A</i> . Nótese que esta función no es una norma matricial. |
| one_norm | Calcula la norma $L[1]$ de <i>A</i> , esto es, el máximo de la suma de los valores absolutos de los elementos de cada columna. |
| inf_norm | Calcula la norma $L[\infty]$ de <i>A</i> , esto es, el máximo de la suma de los valores absolutos de los elementos de cada fila. |
| frobenius | Calcula la norma de Frobenius de <i>A</i> , esto es, la raíz cuadrada de la suma de los cuadrados de los elementos de la matriz. |

58 lbfgs

58.1 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

58.2 Funciones y variables para lbfgs

<code>lbfgs</code> (<i>FOM</i> , <i>X</i> , <i>X0</i> , <i>epsilon</i> , <i>iprint</i>)	Función
<code>lbfgs</code> ([<i>FOM</i> , <i>grad</i>] <i>X</i> , <i>X0</i> , <i>epsilon</i> , <i>iprint</i>)	Function
Encuentra una solución aproximada para el problema de minimización sin restricciones de la función objetivo <i>FOM</i> para la lista de variables <i>X</i> , partiendo de los estimadores iniciales <i>X0</i> , de tal manera que $\text{norm}(\text{grad}(\text{FOM})) < \text{epsilon} * \max(1, \text{norm}(X))$.	
El argumento <i>grad</i> es el gradiente de <i>FOM</i> respecto de las variables <i>X</i> . <i>grad</i> es una lista con el mismo número de elementos que <i>X</i> . Si no se pasa este argumento, el gradiente se calcula automáticamente mediante derivación simbólica.	
El algoritmo utilizado es una técnica <i>cuasi-Newton con memoria limitada</i> (BFGS) [1]. El término <i>memoria limitada</i> 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 <i>iprint</i> controla los mensajes de progreso que envía la función <code>lbfgs</code> .	
<code>iprint[1]</code>	
<code>iprint[1]</code> controla la frecuencia con la que se emiten los mensajes.	
<code>iprint[1] < 0</code>	
No se envían mensajes.	
<code>iprint[1] = 0</code>	
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.
STEPLLENGTH	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 $k = 1, \dots, 8$.

```

(%i1) load (lbfgs);
(%o1)  /usr/share/maxima/5.10.0cvs/share/lbfgs/lbfgs.mac
(%i2) t1[j] := 1 - u[j];
(%o2)          t1 := 1 - u

```

```

(%i3) t2[j] := 10*(u[j + 1] - u[j]^2);
          j      j
          2
(%o3)           t2 := 10 (u      - u )
          j      j + 1   j
(%i4) n : 8;
(%o4)
(%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=  9.68000000000000D+01   GNORM=  4.657353755084532D+02
*****
I  NFN  FUNC                      GNORM                  STEPLENGTH
 1    3  1.651479526340304D+01  4.324359291335977D+00  7.926153934390631D-04
 2    4  1.650209316638371D+01  3.575788161060007D+00  1.000000000000000D+00
 3    5  1.645461701312851D+01  6.230869903601577D+00  1.000000000000000D+00
 4    6  1.636867301275588D+01  1.177589920974980D+01  1.000000000000000D+00
 5    7  1.612153014409201D+01  2.292797147151288D+01  1.000000000000000D+00
 6    8  1.569118407390628D+01  3.687447158775571D+01  1.000000000000000D+00
 7    9  1.510361958398942D+01  4.501931728123680D+01  1.000000000000000D+00
 8   10  1.391077875774294D+01  4.526061463810632D+01  1.000000000000000D+00
 9   11  1.165625686278198D+01  2.748348965356917D+01  1.000000000000000D+00
10   12  9.859422687859137D+00  2.111494974231644D+01  1.000000000000000D+00
11   13  7.815442521732281D+00  6.110762325766556D+00  1.000000000000000D+00
12   15  7.346380905773160D+00  2.165281166714631D+01  1.285316401779533D-01
13   16  6.330460634066370D+00  1.401220851762050D+01  1.000000000000000D+00
14   17  5.238763939851439D+00  1.702473787613255D+01  1.000000000000000D+00
15   18  3.754016790406701D+00  7.981845727704576D+00  1.000000000000000D+00
16   20  3.001238402309352D+00  3.925482944716691D+00  2.333129631296807D-01
17   22  2.794390709718290D+00  8.243329982546473D+00  2.503577283782332D-01
18   23  2.563783562918759D+00  1.035413426521790D+01  1.000000000000000D+00
19   24  2.019429976377856D+00  1.065187312346769D+01  1.000000000000000D+00
20   25  1.428003167670903D+00  2.475962450826961D+00  1.000000000000000D+00
21   27  1.197874264861340D+00  8.441707983493810D+00  4.303451060808756D-01
22   28  9.023848941942773D-01  1.113189216635162D+01  1.000000000000000D+00
23   29  5.508226405863770D-01  2.380830600326308D+00  1.000000000000000D+00
24   31  3.902893258815567D-01  5.625595816584421D+00  4.834988416524465D-01
25   32  3.207542206990315D-01  1.149444645416472D+01  1.000000000000000D+00

```

```

26 33 1.874468266362791D-01 3.632482152880997D+00 1.000000000000000D+00
27 34 9.575763380706598D-02 4.816497446154354D+00 1.000000000000000D+00
28 35 4.085145107543406D-02 2.087009350166495D+00 1.000000000000000D+00
29 36 1.931106001379290D-02 3.886818608498966D+00 1.000000000000000D+00
30 37 6.894000721499670D-03 3.198505796342214D+00 1.000000000000000D+00
31 38 1.443296033051864D-03 1.590265471025043D+00 1.000000000000000D+00
32 39 1.571766603154336D-04 3.098257063980634D-01 1.000000000000000D+00
33 40 1.288011776581970D-05 1.207784183577257D-02 1.000000000000000D+00
34 41 1.806140173752971D-06 4.587890233385193D-02 1.000000000000000D+00
35 42 1.769004645459358D-07 1.790537375052208D-02 1.000000000000000D+00
36 43 3.312164100763217D-10 6.782068426119681D-04 1.000000000000000D+00

```

THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.

```

IFLAG = 0
(%o6) [u  = 1.000005339815974, u  = 1.000009942839805,
      1          2
u  = 1.000005339815974, u  = 1.000009942839805,
      3          4
u  = 1.000005339815974, u  = 1.000009942839805,
      5          6
u  = 1.000005339815974, u  = 1.000009942839805]
      7          8

```

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);
(%o1)  /usr/share/maxima/5.10.0cvs/share/lbfgs/lbfgs.mac
(%i2) FOM : '((1/length(X))*sum((F(X[i])-Y[i])^2, i, 1,
                                         length(X)));
                                         2
                                         sum((F(X ) - Y ), i, 1, length(X))
                                         i     i
(%o2) -----
                                         length(X)
(%i3) X : [1, 2, 3, 4, 5];
(%o3)  [1, 2, 3, 4, 5]
(%i4) Y : [0, 0.5, 1, 1.25, 1.5];
(%o4)  [0, 0.5, 1, 1.25, 1.5]
(%i5) F(x) := A/(1 + exp(-B*(x - C)));
                                         A
(%o5)  F(x) := -----
                                         1 + exp((- B) (x - C))
(%i6) ''FOM;
                                         A          2          A          2
(%o6) ((----- - 1.5)  + (----- - 1.25)
             - B (5 - C)                  - B (4 - C)

```

```

%e          + 1          %e          + 1
      A          2          A          2
+ (----- - 1) + (----- - 0.5)
      - B (3 - C)      - B (2 - C)
%e          + 1          %e          + 1
      2
      A
+ -----)/5
      - B (1 - C)      2
(%e          + 1)
(%i7) estimates : lbfgs (FOM, '[A, B, C], [1, 1, 1], 1e-4, [1, 0]);
*****
N=      3      NUMBER OF CORRECTIONS=25
INITIAL VALUES
F=  1.348738534246918D-01    GNORM=  2.000215531936760D-01
*****
I  NFN  FUNC                      GNORM                  STEPLENGTH
1   3  1.177820636622582D-01  9.893138394953992D-02  8.554435968992371D-01
2   6  2.302653892214013D-02  1.180098521565904D-01  2.100000000000000D+01
3   8  1.496348495303005D-02  9.611201567691633D-02  5.257340567840707D-01
4   9  7.900460841091139D-03  1.325041647391314D-02  1.000000000000000D+00
5  10  7.314495451266917D-03  1.510670810312237D-02  1.000000000000000D+00
6  11  6.750147275936680D-03  1.914964958023047D-02  1.000000000000000D+00
7  12  5.850716021108205D-03  1.028089194579363D-02  1.000000000000000D+00
8  13  5.778664230657791D-03  3.676866074530332D-04  1.000000000000000D+00
9  14  5.777818823650782D-03  3.010740179797255D-04  1.000000000000000D+00

```

```

THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
IFLAG = 0
(%o7) [A = 1.461933911464101, B = 1.601593973254802,
      C = 2.528933072164854]
(%i8) plot2d ([F(x), [discrete, X, Y]], [x, -1, 6]), ''estimates;
(%o8)
```

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;
              2          4          6
(%o2)      F(a, b, c) := (a - 5)  + (b - 3)  + (c - 2)
(%i3) F_grad : map (lambda ([x], diff (F(a, b, c), x)), [a, b, c]);
              3          5
(%o3)      [2 (a - 5), 4 (b - 3) , 6 (c - 2) ]
(%i4) estimates : lbfgs ([F(a, b, c), F_grad], [a, b, c], [0, 0, 0], 1e-4, [1,
*****
N=      3      NUMBER OF CORRECTIONS=25
INITIAL VALUES
```

```

F= 1.70000000000000D+02   GNORM= 2.205175729958953D+02
*****
I  NFN  FUNC          GNORM          STEPLENGTH
1    2  6.632967565917638D+01  6.498411132518770D+01  4.534785987412505D-03
2    3  4.368890936228036D+01  3.784147651974131D+01  1.000000000000000D+00
3    4  2.685298972775190D+01  1.640262125898521D+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  1.215263596054294D+00  2.204727876126879D+00  1.000000000000000D+00
7    8  1.080252896385334D-02  1.431637116951849D-01  1.000000000000000D+00
8    9  8.407195124830908D-03  1.126344579730013D-01  1.000000000000000D+00
9   10  5.022091686198527D-03  7.750731829225274D-02  1.000000000000000D+00
10  11  2.277152808939775D-03  5.032810859286795D-02  1.000000000000000D+00
11  12  6.489384688303218D-04  1.932007150271008D-02  1.000000000000000D+00
12  13  2.075791943844548D-04  6.964319310814364D-03  1.000000000000000D+00
13  14  7.349472666162257D-05  4.017449067849554D-03  1.000000000000000D+00
14  15  2.293617477985237D-05  1.334590390856715D-03  1.000000000000000D+00
15  16  7.683645404048675D-06  6.011057038099201D-04  1.000000000000000D+00

```

THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.

IFLAG = 0

(%o4) [a = 5.000086823042934, b = 3.05239542970518,
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`.

59 lindstedt

59.1 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")`.

60 linearalgebra

60.1 Introducción a linearalgebra

El paquete `linearalgebra` contiene una batería de funciones para álgebra lineal.

Ejemplo:

```
(%i1) M : matrix ([1, 2], [1, 2]);
(%o1)
      [ 1   2 ]
      [         ]
      [ 1   2 ]

(%i2) nullspace (M);
(%o2)
      [ 1   ]
      [         ]
      span([ 1 ])
      [ - - ]
      [ 2 ]

(%i3) columnspace (M);
(%o3)
      [ 1   ]
      span([         ])
      [ 1   ]

(%i4) ptriangularize (M - z*ident(2), z);
(%o4)
      [ 1   2 - z   ]
      [         ]
      [             2 ]
      [ 0   3 z - z   ]

(%i5) M : matrix ([1, 2, 3], [4, 5, 6], [7, 8, 9]) - z*ident(3);
(%o5)
      [ 1 - z   2   3   ]
      [         ]
      [ 4   5 - z   6   ]
      [         ]
      [ 7   8   9 - z   ]

(%i6) MM : ptriangularize (M, z);
(%o6)
      [ 4   5 - z           6           ]
      [         ]
      [             2           ]
      [ 66           z   102 z   132   ]
      [ 0   --   - -- + ----- + ----- ]
      [ 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]);
(%o8)
      [z   - 15 z   - 18 z]
```

```
(%i9) MM : ratsimp (MM);
      [ 4 5 - z           6          ]
      [                           ]
      [                           2        ]
(%o9)   [ 66      7 z  - 102 z - 132 ]
      [ 0  --  - ----- ]
      [ 49           49          ]
      [                           ]
      [ 0 0           0          ]
(%i10) nullspace (MM);
      [           1          ]
      [                   ]
      [           2          ]
      [ z - 14 z - 16 ]
      [ ----- ]
(%o10)  span([           8          ])
      [                   ]
      [           2          ]
      [ z - 18 z - 12 ]
      [ ----- ]
      [           12         ]
(%i11) M : matrix ([1, 2, 3, 4], [5, 6, 7, 8],
                  [9, 10, 11, 12], [13, 14, 15, 16]);
      [ 1 2 3 4 ]
      [           ]
      [ 5 6 7 8 ]
(%o11)
      [           ]
      [ 9 10 11 12 ]
      [           ]
      [ 13 14 15 16 ]
(%i12) columnspace (M);
      [ 1 ]  [ 2 ]
      [     ]  [     ]
      [ 5 ]  [ 6 ]
(%o12)  span([     ], [     ])
      [ 9 ]  [ 10 ]
      [     ]  [     ]
      [ 13 ]  [ 14 ]
(%i13) apply ('orthogonal_complement, args (nullspace (transpose (M))));
      [ 0 ]  [ 1 ]
      [     ]  [     ]
      [ 1 ]  [ 0 ]
(%o13)  span([     ], [     ])
      [ 2 ]  [ - 1 ]
      [     ]  [     ]
      [ 3 ]  [ - 2 ]
```

60.2 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.

columnnop (*M, i, j, theta*)

Función

Si *M* es una matriz, devuelve la matriz que resulta de hacer la operación columna $C_i \leftarrow C_i - \theta * C_j$. Si *M* carece de cualquiera de las filas *i* o *j*, devuelve un mensaje de error.

columnsnap (*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 $\{v_1, \dots, v_n\}$ 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 ]
      [       ]
      [ 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 $x = \text{lu_factor}(A)$, 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 $\text{diff}(f, x[i], 1, x[j], 1)$.

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 $\text{diff}(f[i], x[j])$.

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 Kronecker de 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 satisface cierta propiedad. Hay tres posibilidades:

(1) $rel = \text{'bool}$ 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) $rel = \text{'max}$ 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 $M = \text{lu_factor}(A, \text{field})$, 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 entero. 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 ($M, 1$)
mat_cond (M, \inf)

Función
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 ($M, 1$)
mat_norm (M, \inf)
mat_norm ($M, \text{frobenius}$)

Función
Función
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)
matrixp (e)

Función
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 ]
```

nonnegintegerp (n)

Función

Devuelve `true` si y solo si $n \geq 0$, siendo n un entero.

nullspace (M)

Función

Si M es una matriz, devuelve `span (v_1, ..., v_n)`, siendo $\{v_1, \dots, v_n\}$ 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 $\{u_1, \dots, u_m\}$ la base del complemento ortogonal del conjunto (v_1, \dots, 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.

polynomialp (*p, L, coeffp*) equivale a **polynomialp** (*p, L, coeffp, 'nonnegintegerp*).

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 $p = (-1)^n \text{charpoly}(\text{polytocompanion}(p, x))$.

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. $M2 = E_{-n} \dots E_{-1} M$, donde E_{-1}, \dots, 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 realizar la transformación $R_i \leftarrow R_i - \theta * R_j$ 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 ($[x_1, \dots, x_n]$)

Función

Devuelve una matriz n por n , cuya i -ésima fila es $[1, x_i, x_i^2, \dots, x_i^{(n-1)}]$.

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**.

zeromatrixxp (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 **zeromatrixxp** devuelve a su vez **true** para cada elemento de e .

61 lsquares

61.1 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 $[a = \dots, b = \dots, c = \dots]$. 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           ]
          [           ]
          [ 9           ]
```

(%o2)

```

[ - 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 $[a = \dots, b = \dots, c = \dots]$. 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
      ====
      \
      >     ((D + M      )^2 - C - M      B - M      A)^2
      /
      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 *lbfsgs*. 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 *lbfsgs*.

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           ]
      [           ]
      [ 9           ]
      [ -   2   1 ]
      [ 4           ]
      [           ]
      [ 3   2   2 ]
      [           ]
      [ 2   2   1 ]
(%o2)
(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);
      5
      ====
      \
      >    ((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

$$\frac{1}{n} \sum_{i=1}^n [\text{lhs}(e_i) - \text{rhs}(e_i)]^2,$$

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
=====
\     ((D + M      ) - C - M      B - M      A)
/           i, 1           i, 3           i, 2
=====
i = 1
(%o3) -----
      5
(%i4) diff (mse, D);
      5
=====
\     ((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
((D + 3) - C - 2 B - 2 A) + ((D + -) - C - B - 2 A)
      4
+ ((D + 2) - C - B - 2 A) + ((D + -) - C - 2 B - A)
      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

$$\text{lhs}(e_i) - \text{rhs}(e_i),$$

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           ]
      [           ]
      [ 9           ]
      [ -   2   1 ]
      [ 4           ]
      [           ]
      [ 3   2   2 ]
      [           ]
      [ 2   2   1 ]
(%o2)
(%i3) a : lsquares_estimates (
      M, [z,x,y], (z+D)^2 = A*x+B*y+C, [A,B,C,D]);
      59      27      10921      107
      [[A = - --, B = - --, C = -----, D = - ---]]
      16      16      1024      32
(%o3)
(%i4) lsquares_residuals (
      M, [z,x,y], (z+D)^2 = A*x+B*y+C, first(a));
      13      13      13      13      13
      [--, - --, - --, --, --]
      64      64      32      64      64
(%o4)
```

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

$$\frac{1}{n} \sum_{i=1}^n [\text{lhs}(e_i) - \text{rhs}(e_i)]^2,$$

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 *maxdegree* = 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 depen-

dientes, `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 $n \times n$ 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],[0,1,0,1,1,1,0,0],  
                  [1,0,0,1,1,1,0,0],[1,1,1,1,0,0,0,1])$  
(%i8) plsquares(mtrx,[a,b,_And,_Or,_Xor,_Nand,_Nor,_Nxor],  
                  [_And,_Or,_Xor,_Nand,_Nor,_Nxor],1,0);  
      Determination Coefficient for
```

```
[_And, _Or, _Xor, _Nand, _Nor, _Nxor] =
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
(%o2) [_And = a b, _Or = - a b + b + a,
_Xor = - 2 a b + b + a, _Nand = 1 - a b,
_Nor = a b - b - a + 1, _Nxor = 2 a b - b - a + 1]
```

Antes de hacer uso de esta función ejecútese `load("plsquares")`.

62 makeOrders

62.1 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")`.

63 mnewton

63.1 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.

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]]
```

Antes de hacer uso de esta función ejecútese **load("mnewton")**. Véanse también **newtonepsilon** y **newtonmaxiter**.

64 numericalio

64.1 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.

64.1.1 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.

64.1.2 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.

64.1.3 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 átomofalse se escribe tal cual y una lista `[1234, false, Foo]` se escribe `1234, false, Foo`.

64.1.4 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.

64.2 Funciones y variables para entrada y salida en formato texto

<code>read_matrix (S)</code>	Función
<code>read_matrix (S, M)</code>	Función
<code>read_matrix (S, separator_flag)</code>	Función
<code>read_matrix (S, M, separator_flag)</code>	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.

<code>read_array (S, A)</code>	Función
<code>read_array (S, A, separator_flag)</code>	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.

<code>read_hashed_array (S, A)</code>	Función
<code>read_hashed_array (S, A, separator_flag)</code>	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.

<code>read_nested_list (S)</code>	Función
<code>read_nested_list (S, separator_flag)</code>	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.

<code>read_list (S)</code>	Función
<code>read_list (S, L)</code>	Función
<code>read_list (S, separator_flag)</code>	Función
<code>read_list (S, L, separator_flag)</code>	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.

<code>write_data (X, D)</code>	Función
<code>write_data (X, D, separator_flag)</code>	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 sublistas 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.

64.3 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`.

65 opsubst

65.1 Funciones y variables para opsubst

opsubst (<i>f,g,e</i>)	Función
opsubst (<i>g=f,e</i>)	Función
opsubst ([<i>g1=f1,g2=f2,..., gn=fn</i>], <i>e</i>)	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);
(%o10)                               a
                  -
                  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")`.

66 orthopoly

66.1 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).

66.1.1 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);
(%o1)                                2
                                         - 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 $\text{hermite}(2, m[i,j])$, que no es lo mismo que calcular $-2 + 4 m \cdot m$, 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 (%);
$$P_{\{n\}}^{\{\left(a,a-b\right)}\left(\frac{x}{2}\right)}$$
(%o4) false

```

66.1.2 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 expresión 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            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);
(%o1) Pn1(x)
(%i2) float (%);
(%o2) Pn1.0(x)
(%i3) ev (% , n=2, x=0.9);
(%o3) P21.0(0.9)
```

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)
(%o2) 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
```

66.1.3 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(- 0.01917222265624955, 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)
```

66.1.4 Gráficos y orthopoly

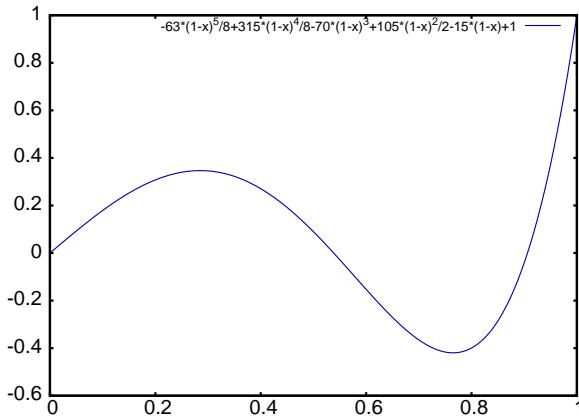
Para representar gráficamente expresiones que contengan polinomios ortogonales se deben hacer dos cosas:

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)`'.

66.1.5 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 $x = -1, -2, \dots, -n$. 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.

66.1.6 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.

66.2 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 $a \leq -1$ o $b \leq -1$.

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.

$$\begin{aligned} (\%i1) \quad & \text{orthopoly_recur (legendre_p, [n, x]);} \\ & (2 n - 1) P_{n-1}(x) x + (1 - n) P_{n-2}(x) \\ (\%o1) \quad & P_n(x) = \frac{(2 n - 1) P_{n-1}(x) x + (1 - n) P_{n-2}(x)}{n} \end{aligned}$$

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,

$$\begin{aligned} (\%i1) \quad & w : \text{orthopoly_weight (hermite, [n, x]);} \\ & 2 \\ & - x \\ (\%o1) \quad & [\%e^{\frac{-x^2}{2}}, -\infty, \infty] \\ (\%i2) \quad & \text{integrate (w[1] * hermite (3, x) * hermite (2, x), x, w[2], w[3])}; \\ (\%o2) \quad & 0 \end{aligned}$$

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* \leq **pochhammer_max_index**, la expresión **pochhammer** (*x, n*) se evalúa como el producto $x (x + 1) (x + 2) \dots (x + n - 1)$ si *n* > 0 y como 1 si *n* = 0. Para *n* negativo, **pochhammer** (*x, n*) se define como $(-1)^n / \text{pochhammer} (1 - x, -n)$. 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));
(%o1) -----
                                         gamma(x + n)
                                         -----
                                         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 $n \leq \text{pochhammer_max_index}$.

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 (<i>n, x</i>)	Función
Función esférica de Hankel de segunda especie.	
Referencia: Abramowitz y Stegun, ecuación 10.1.17, página 439.	
spherical_harmonic (<i>n, m, x, y</i>)	Función
Función armónica esférica.	
Referencia: Merzbacher 9.64.	
unit_step (<i>x</i>)	Función
Función de escalón unidad continua por la izquierda, definida de tal forma que unit_step (<i>x</i>) se anula para $x \leq 0$ 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 $(1 + \text{signum}(x))/2$.	
ultraspherical (<i>n, a, x</i>)	Función
Polinomio ultraesférico o de Gegenbauer.	
Referencia: Abramowitz y Stegun, ecuación 22.5.46, página 779.	

67 plotdf

67.1 Introducción a plotdf

La función `plotdf` crea un gráfico del campo de direcciones para una Ecuación Diferencial Ordinaria (EDO) de primer orden, o para un sistema de dos EDO's autónomas, de primer orden.

Como se trata de un paquete adicional, para poder usarlo debe cargarlo primero con el comando `load("plotdf")`. También es necesario que Xmaxima esté instalado, a pesar de que ejecute Maxima desde otra interface diferente.

Para dibujar el campo de direcciones de una única EDO, esa ecuación deberá escribirse en la forma siguiente:

$$\frac{dy}{dx} = F(x, y)$$

y la función F será dada como argumento para el comando `plotdf`. La variable independiente tiene que ser siempre x y la variable dependiente y . A esas dos variables no podrá estar asociado ningún valor numérico.

Para dibujar el campo de direcciones de un sistema autónomo de dos EDO's, Las dos ecuaciones deben ser escritas en la forma siguiente

$$\frac{dx}{dt} = G(x, y) \quad \frac{dy}{dt} = F(x, y)$$

y el argumento para el comando `plotdf` será una lista con dos expresiones para las funciones F y G .

Cuando se trabaja con una única ecuación, `plotdf` asume implícitamente que $x=t$ y $G(x,y)=1$, transformando la ecuación en un sistema autónomo con dos ecuaciones.

67.2 Funciones y variables para plotdf

plotdf ($dydx, \dots$ opciones...)

Function

plotdf ($[dxdt, dydt], \dots$ opciones...)

Function

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, seleccione 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.

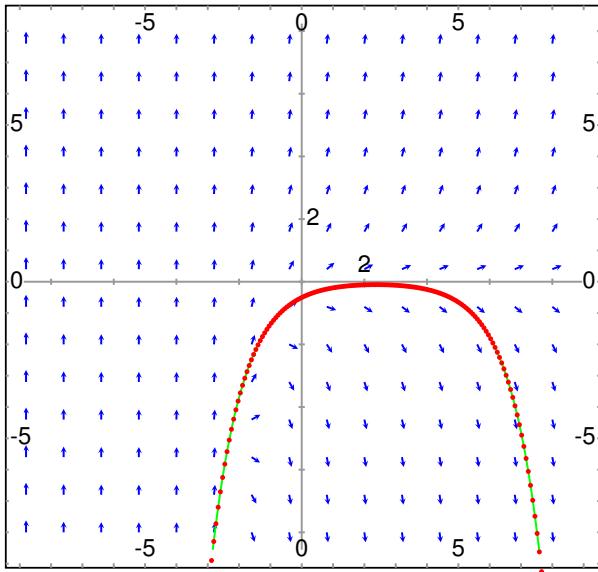
Ejemplos:

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")$
```

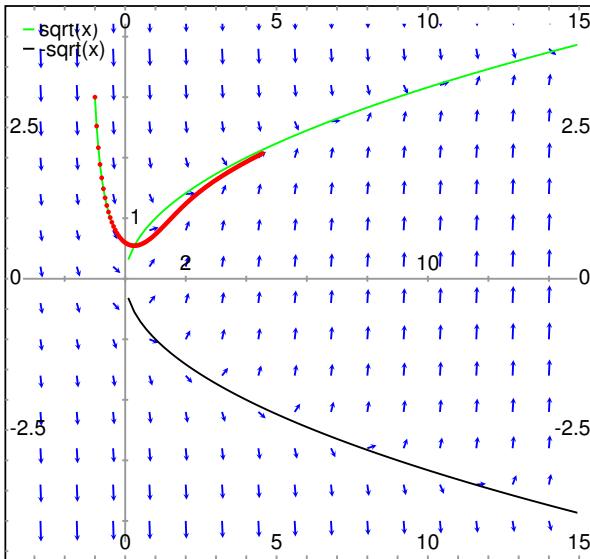
```
(%i2) plotdf(exp(-x)+y,[trajectory_at,2,-0.1]);
```



- 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]);
```

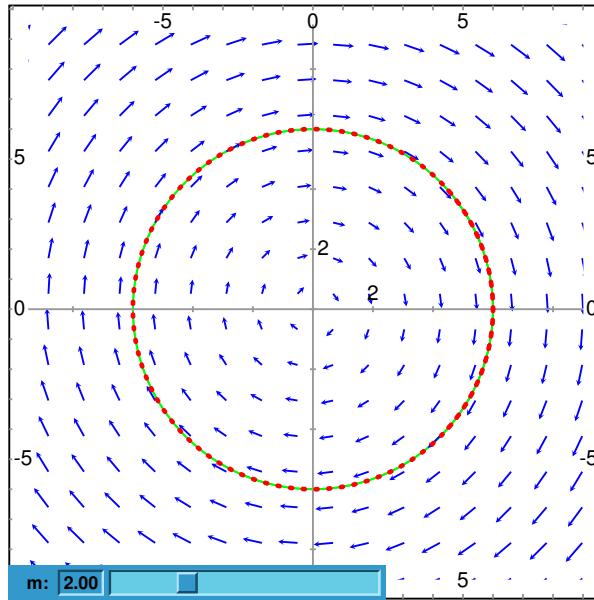
El gráfico también muestra la función $y = \sqrt{x}$.



- 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

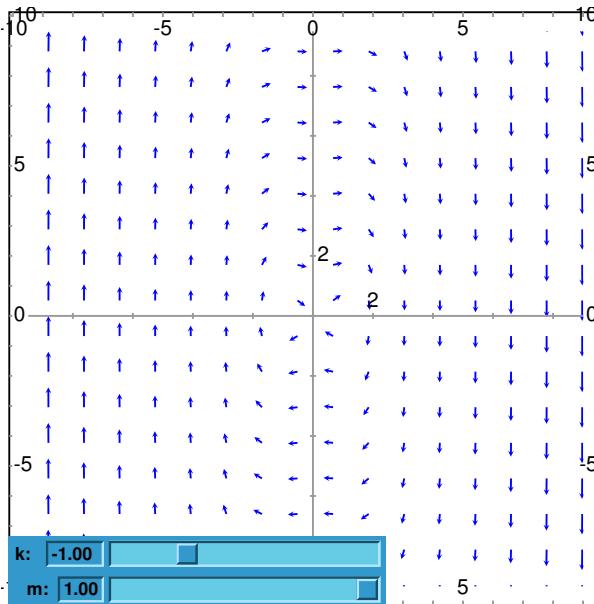
pasa por $(x, y) = (6, 0)$, con una barra de deslizamiento que permitirá cambiar el valor de m interactivamente (k permanece fijo a 2):

```
(%i4) plotdf([y,-k*x/m], [parameters,"m=2,k=2"],  
[sliders,"m=1:5"], [trajectory_at,6,0]);
```



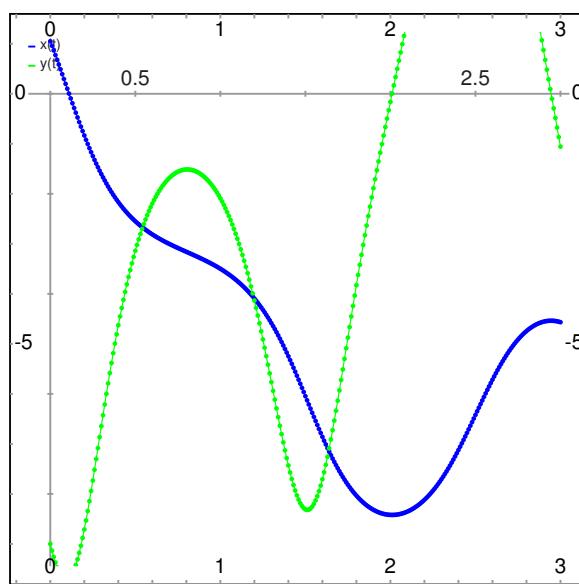
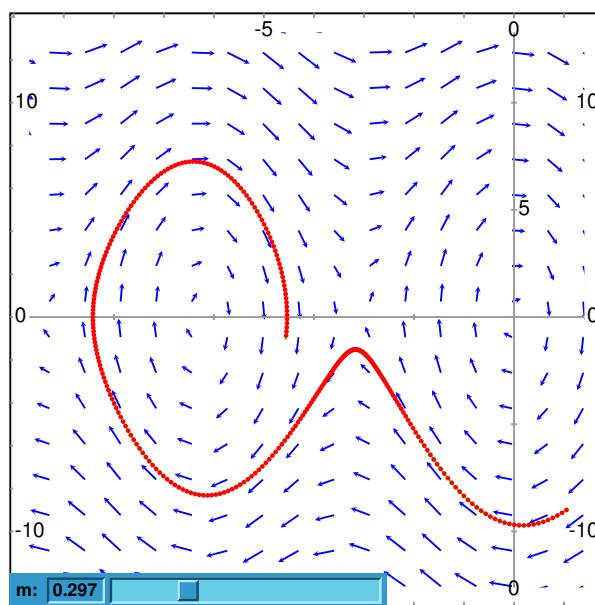
- Para representar el campo de direcciones de la ecuación de Duffing, $m * x'' + c * x' + k * x + b * x^3 = 0$, se introduce la variable $y = x'$ y se hace:

```
(%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]);
```



68 romberg

68.1 Funciones y variables para romberg

romberg (*expr, x, a, b*)
romberg (*F, a, b*)

Función 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 $2^{\text{rombergit}}$. 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) := ----- + ----- + -----
```

```


$$\frac{(x - 1)^2}{100} + \frac{1}{1000} \quad \frac{(x - 2)^2}{1000} + \frac{1}{10000} \quad \frac{(x - 3)^2}{200} + \frac{1}{2000}$$

(%i3) rombergtol : 1e-6;
(%o3) 9.999999999999995E-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);
(%o2) g(x, y) :=  $\frac{xy}{x+y}$ 
(%i3) rombergtol : 1e-6;
(%o3) 9.999999999999995E-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);
(%o6) 
$$-\frac{9 \log(-) + 9 \log(3) + \frac{27}{2}}{26 \log(3) - 26 \log(2) - 13}$$

(%i7) exact : radcan (%);
(%o7) 
$$-\frac{3}{\frac{27}{2}}$$

(%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 $2^{\text{rombergit}}$. 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`.

69 simplex

69.1 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  
          [--, [y = --, x = -]]  
          10     10      5
```

69.2 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  
          [[--, 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 \geq y $<$ por \leq . 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**.

70 simplification

70.1 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.

70.2 Funciones y variables para simplification

70.2.1 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
                               x
(%o2)
(%i3) diff (abs (x), x);
                               x
(%o3) -----
                           abs(x)
(%i4) cosh (abs (x));
(%o4)                  cosh(x)
```

70.2.2 Paquete `faceexp`

El paquete `faceexp` 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. Si no se especifican variables, como en `facsum (expr)`, el resultado devuelto es el mismo que el conseguido mediante `ratsimp (expr)`.

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 *sólo* 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, arg₁, ... arg_n*)

Función

Devuelve una expresión equivalente a *expr* obtenida aplicando **facsum** a los factores de *expr*, de argumentos *arg₁, ... 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, arg₁, ..., arg_n*)

Función

Si algunas expresiones fueron ya simplificadas con **facsum**, **factorfacsum**, **factenexpand**, **facextension** o **factorfacextension**, debiendo ser luego sumadas, puede ser conveniente combinarlas utilizando la función **collectterms**, 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.

70.2.3 Paquete functs**rempart** (*expr, n*)

Función

Elimina la parte *n* de la expresión *expr*.

Si *n* es una lista de la forma *[l, m]*, entonces las partes desde *l* a *m* serán eliminadas.

Para hacer uso de esta función ejecutar **load(functs)**.

wronskian (*[f₁, ..., f_n], x*)

Función

Devuelve la matriz wronskiana de las expresiones *f₁, ..., f_n* dependientes 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(funccts)`.

logand (x, y) Función

Devuelve el "y" lógico binario de los argumentos x e y .

Para hacer uso de esta función ejecutar `load(funccts)`.

logor (x, y) Función

Devuelve el "o" lógico binario de los argumentos x e y .

Para hacer uso de esta función ejecutar `load(funccts)`.

logxor (x, y) Función

Devuelve el "o-exclusivo" lógico binario de los argumentos x e y .

Para hacer uso de esta función ejecutar `load(funccts)`.

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(funccts)`.

linear (expr, x) Función

Si `expr` es una expresión lineal respecto de la variable x , `linear` devuelve $a*x + b$, siendo a no nula y, junto con b , no incluye a x . En otro caso, `linear` devuelve `expr`.

Para hacer uso de esta función ejecutar `load(funccts)`.

gcddivide (p, q) Función

Si `takegcd` vale `true`, `gcddivide` divide los polinomios p y q por su máximo común divisor y devuelve el cociente de los resultados.

Si `takegcd` vale `false`, `gcddivide` devuelve el cociente p/q .

Para hacer uso de esta función ejecutar `load(funccts)`.

arithmetic (a, d, n) Función

Devuelve el n -ésimo término de la progresión aritmética $a, a + d, a + 2*d, \dots, a + (n - 1)*d$.

Para hacer uso de esta función ejecutar `load(funccts)`.

geometric (a, r, n) Función

Devuelve el n -ésimo término de la progresión geométrica $a, a*r, a*r^2, \dots, a*r^{(n - 1)}$.

Para hacer uso de esta función ejecutar `load(funccts)`.

harmonic (a, b, c, n)	Función
Devuelve el n -ésimo término de la progresión armónica $a/b, a/(b + c), a/(b + 2*c), \dots, a/(b + (n - 1)*c)$.	
Para hacer uso de esta función ejecutar <code>load(functs)</code> .	
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 <code>load(functs)</code> .	
geosum (a, r, n)	Función
Devuelve la suma de la sucesión geométrica hasta el n -ésimo término. Si n es infinito (<code>inf</code>) la suma será finita sólo si el valor absoluto de r es menor que 1.	
Para hacer uso de esta función ejecutar <code>load(functs)</code> .	
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 <code>load(functs)</code> .	
gd (x)	Función
Devuelve la función de Gudermann, $2*\tan(\%e^x) - \pi/2$.	
Para hacer uso de esta función ejecutar <code>load(functs)</code> .	
agd (x)	Función
Devuelve la inversa de la función de Gudermann, $\log(\tan(\pi/4 + x/2)))$.	
Para hacer uso de esta función ejecutar <code>load(functs)</code> .	
vers (x)	Función
Devuelve $1 - \cos(x)$.	
Para hacer uso de esta función ejecutar <code>load(functs)</code> .	
covers (x)	Función
Devuelve $1 - \sin(x)$.	
Para hacer uso de esta función ejecutar <code>load(functs)</code> .	
exsec (x)	Función
Devuelve $\sec(x) - 1$.	
Para hacer uso de esta función ejecutar <code>load(functs)</code> .	
hav (x)	Función
Devuelve $(1 - \cos(x))/2$.	
Para hacer uso de esta función ejecutar <code>load(functs)</code> .	

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(funccts)**.**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(funccts)**.**70.2.4 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.
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 */
(%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 regla 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 regla 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.

70.2.5 Paquete rducon

reduce_consts (expr)

Funcióñ

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.

70.2.6 Paquete scifac

gcfac (*expr*)

Funcióñ

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.

70.2.7 Paquete sqdnst

sqrtdenest (*expr*)

Funcióñ

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  
-----  
      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.

71 solve_rec

71.1 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)
```

71.2 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 obtener 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]);
(%o6)          %u =  $\frac{1}{n(n+1)!}$ 
```

So the general solution is

$$\frac{\sum_{j=0}^{n-1} \frac{(-1)^j}{(j+1)!}}{2^n} + \frac{1}{n!}$$

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(binom(n+k,k)/2^k, k, 0, n) +  

      sum(binom(2*n, 2*k), k, 0, n);  

(%o2) >  $\frac{\sum_{k=0}^n \binom{n+k}{k}}{2^n} + \binom{2n}{2k}$   

(%i3) simplify_sum(%);  

(%o3)  $\frac{4^n}{2^{2n}}$ 
```

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 Riccati.

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 ejéctuese previamente `load(solve_rec);`.

Ejemplo de recurrencia lineal con coeficientes constantes:

$$\begin{aligned}
 (%i2) \quad & \text{solve_rec}(a[n]=a[n-1]+a[n-2]+n/2^n, a[n]); \\
 (%o2) \quad a[n] = & \frac{\frac{(sqrt(5)-1)^n}{1} - \frac{(-1)^n}{5^2}}{\frac{n^n}{2^n}}
 \end{aligned}$$

$$\begin{aligned}
 & + \frac{\frac{(sqrt(5)+1)^n}{2^2} - \frac{(-1)^n}{5^2}}{\frac{n^n}{2^n}}
 \end{aligned}$$

Ejemplo de recurrencia lineal con coeficientes polinomiales:

$$\begin{aligned}
 (%i7) \quad & 2*x*(x+1)*y[x] - (x^2+3*x-2)*y[x+1] + (x-1)*y[x+2]; \\
 (%o7) \quad & \frac{(x-1)y[x]}{x+2} - \frac{(x^2+3x-2)y[x+1]}{x+1} + \frac{2x(x+1)y[x+2]}{x} \\
 (%i8) \quad & \text{solve_rec}(% , y[x], y[1]=1, y[3]=3); \\
 (%o9) \quad y[x] = & \frac{x^3 2^x}{x^4 2^{x!}}
 \end{aligned}$$

Ejemplo de recurrencia de Riccati:

$$\begin{aligned}
 (%i2) \quad & x*y[x+1]*y[x] - y[x+1]/(x+2) + y[x]/(x-1) = 0; \\
 (%o2) \quad & \frac{y[x+1]}{x} - \frac{y[x]}{x+2} + \frac{y[x]}{x-1} = 0 \\
 (%i3) \quad & \text{solve_rec}(% , y[x], y[3]=5)$ \\
 (%i4) \quad & \text{ratsimp(minfactorial(factcomb(%)))}; \\
 (%o4) \quad y[x] = & \frac{30x^3 - 30x^2}{5x^6 - 3x^5 - 25x^4 + 15x^3 + 20x^2 - 12x - 1584}
 \end{aligned}$$

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) ax + 3 + (x + 3) ax + 2 - x ax + 1

$$+ (x^2 - 1) a^x$$


(%i2) solve_rec_rat(% = (x+2)/(x+1), a[x]);
(%o2)      a =  $\frac{1}{x(x - 1)(x + 1)}$ 
```

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  


$$n + 2$$
  


$$+ 12 (2 n + 3) (9 n^2 + 27 n + 22) sm$$
  


$$n + 1$$
  


$$- 81 (n + 1) (3 n + 2) (3 n + 4) sm$$

```

```
(%i21) sum(''summand, k, 0, n), n=0;
(%o21)
(%i22) sum(''summand, k, 0, n), n=1;
(%o22)
(%i23) product_use_gamma: false$  

(%i24) solve_rec(%o19, sm[n], sm[0]=1, sm[1]=4);
          n
          n - 1           n - 1
          /====\           /====\
          ! !           ! !
          ( ! !   (3 %j + 2)) ( ! !   (3 %j + 4)) 3
          ! !
          %j = 0           %j = 0
(%o24) sm = -----
          n           n - 1
          /====\
          ! !
          ( ! !   (2 %j + 3)) 2   n!
          ! !
          %j = 0
```


72 stats

72.1 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'.

72.2 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
|
|   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`.

72.3 Funciones y variables para stats

stats_numer

Valor por defecto: `true`

Variable opcional

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 χ^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 = Asymptotic test with Yates correction.  
|  
| hypotheses = H0: p = 0.5 , H1: p # 0.5
```

```

|           statistic = .43689
|
|           distribution = [normal, 0.5, .048872]
|
|           p_value = .19662

```

test_proportions_difference (*x1, n1, x2, n2*)

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 $H_0: p_1 = p_2$ contra la alternativa $H_1: p_1 > p_2$, 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));

$$\sqrt{\frac{1}{n_2} + \frac{1}{n_1} - \frac{(x_2 + x_1)}{n_2 + n_1}}$$

(%o4)

```

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 extraído la muestra es 6, frente a la alternativa $H_1 : \text{median} > 6$.

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 $H_0 : \text{median} = 15$ frente a la alternativa $H_1 : \text{median} > 15$. 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 $H_1 : \text{median1} < \text{median2}$.

```
(%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 cualesquier 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
```

`simple_linear_regression (x)`

Función
Función

`simple_linear_regression (x option)`

Regresión lineal simple, $y_i = a + bx_i + e_i$, donde las e_i son variables aleatorias independientes de distribución $N(0, \sigma)$. El argumento `x` debe ser una matriz de dos columnas o una lista de pares de números.

Opciones:

- `'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).
- `'regressor`, valor por defecto '`x`', nombre de la variable independiente.

El resultado devuelto por la función `simple_linear_regression` es un objeto `inference_result` con los siguientes apartados:

1. `'model`: la ecuación ajustada. Útil para hacer predicciones. Véanse los ejemplos más abajo.
2. `'means`: media bivariante.
3. `'variances`: varianzas de ambas variables.
4. `'correlation`: coeficiente de correlación.
5. `'adc`: coeficiente de determinación ajustado.
6. `'a_estimation`: estimador del parámetro a .
7. `'a_conf_int`: intervalo de confianza del parámetro a .
8. `'b_estimation`: estimador del parámetro b .
9. `'b_conf_int`: intervalo de confianza del parámetro b .
10. `'hypotheses`: hipótesis nula y alternativa sobre el parámetro b .
11. `'statistic`: valor del estadístico de contraste utilizado para probar la hipótesis.
12. `'distribution`: distribución del estadístico de contraste, junto con su parámetro.
13. `'p_value`: p -valor del test sobre b .

14. `'v_estimation`: estimador insesgado de la varianza, o varianza residual.
15. `'v_conf_int`: intervalo de confianza de la varianza.
16. `'cond_mean_conf_int`: intervalo de confianza para la media condicionada. Véanse los ejemplos más abajo.
17. `'new_pred_conf_int`: intervalo de confianza para una nueva predicción. Véanse los ejemplos más abajo.
18. `'residuals`: lista de pares (predicción, residuo), ordenado respecto de las predicciones. Útil para el análisis de la bondad de ajuste. Véanse los ejemplos más abajo.

Sólo los elementos 1, 4, 14, 9, 10, 11, 12 y 13 de arriba, y en este orden, son mostrados por defecto. El resto permanecen ocultos hasta que el usuario haga uso de las funciones `items_inference` y `take_inference`.

Ejemplo:

Ajuste de un modelo lineal a una muestra bivariante. La entrada `%i4` representa gráficamente la muestra junto con la recta de regresión; la entrada `%i5` calcula y dado `x=113`; también se calculan las medias y el intervalo de confianza para una nueva predicción cuando `x=113`.

```
(%i1) load("stats")$  
(%i2) s: [[125,140.7], [130,155.1], [135,160.3],  
           [140,167.2], [145,169.8]]$  
(%i3) z:simple_linear_regression(s,conflevel=0.99);  
          |           SIMPLE LINEAR REGRESSION  
          |  
          |   model = 1.405999999999985 x - 31.18999999999804  
          |  
          |   correlation = .9611685255255155  
          |  
          |   v_estimation = 13.579666666666665  
          |  
(%o3)    |   b_conf_int = [.04469633662525263, 2.767303663374718]  
          |  
          |   hypotheses = H0: b = 0 ,H1: b # 0  
          |  
          |   statistic = 6.032686683658114  
          |  
          |   distribution = [student_t, 3]  
          |  
          |   p_value = 0.0038059549413203  
(%i4) plot2d([[discrete, s], take_inference(model,z)],  
           [x,120,150],  
           [gnuplot_curve_styles, ["with points","with lines"]])$■  
(%i5) take_inference(model,z), x=133;  
(%o5)                      155.808  
(%i6) take_inference(means,z);  
(%o6)                      [135.0, 158.62]  
(%i7) take_inference(new_pred_conf_int,z), x=133;  
(%o7)                      [132.0728595995113, 179.5431404004887]
```

72.4 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`.

73 stirling

73.1 Funciones y variables para stirling

stirling (<i>z,n</i>)	Función
stirling (<i>z,n,pred</i>)	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")`.

74 stringproc

74.1 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 van.nek at arcor.de .

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 formatear 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.

74.2 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 el flujo de datos *stream*.

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** ajusta la posición en *stream*. El argumento *pos* debe ser un número positivo, ocupando el primer elemento en *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**.

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 (<i>dest, string</i>)	Función
printf (<i>dest, string, expr_1, ..., expr_n</i>)	Función
Pone al alcance de Maxima la función FORMAT de Common Lisp.	
Véanse las referencias de Lisp para más información.	
La siguiente descripción y los ejemplos pueden dar una idea de cómo usar printf.	
~%	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 (<i>bigfloat</i>)
~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 ~[está indexada a partir del cero. Téngase en cuenta que la directiva ~* no está soportada.

```
(%i1) printf( false, "~a ~a ~4f ~a ~@r",
              "String",sym,bound,sqrt(12),144), bound = 1.234;
(%o1)                               String sym 1.23 2*sqrt(3) CXLIV
(%i2) printf( false,"~{~a ~}",["one",2,"THREE"] );
(%o2)                               one 2 THREE
(%i3) printf( true,"~{~{~9,1f ~}~%~}",mat ),
      mat = args( matrix([1.1,2,3.33],[4,5,6],[7,8.88,9]) )$■
      1.1      2.0      3.3
      4.0      5.0      6.0
      7.0      8.9      9.0
(%i4) control: "~:(~r~) bird~p ~[is~;are~] singing."$
(%i5) printf( false,control, n,n,if n=1 then 0 else 1 ), n=2;
(%o5)                               Two birds are singing.
```

Si *dest* es un flujo de datos o **true**, entonces **printf** devuelve **false**. En otro caso, **printf** devuelve una cadena con la salida.

readline (<i>stream</i>)	Función
Devuelve una cadena con los caracteres desde la posición actual en el flujo de datos <i>stream</i> hasta el final de la línea, o false si está al final del fichero.	

sprint (*expr_1, ..., expr_n*) Función

Evaluá 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 22 do sprint( fib(n) )$  
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765■  
(%i2) for n:0 thru 22 do (  
    sprint(fib(n)), if mod(n,10)=9 then newline() )$  
0 1 1 2 3 5 8 13 21 34  
55 89 144 233 377 610 987 1597 2584 4181  
6765 10946 17711
```

74.3 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 $-1 < \text{int} < 256$.

```
(%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*.

lessp (<i>char_1, char_2</i>)	Función
Devuelve true si el número ASCII de <i>char_1</i> es menor que el de <i>char_2</i> .	
lesspignore (<i>char_1, char_2</i>)	Función
Como lessp , pero ignora si las letras están en mayúsculas o minúsculas.	
constituent (<i>char</i>)	Función
Devuelve true si <i>char</i> 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 (<i>lisp_char</i>)	Función
Convierte un carácter Lisp en uno de Maxima. El uso de esta función por parte del usuario no será necesario.	
digitcharp (<i>char</i>)	Función
Devuelve true si <i>char</i> es un dígito.	
lcharp (<i>obj</i>)	Función
Devuelve true si <i>obj</i> es un carácter de Lisp. El uso de esta función por parte del usuario no será necesario.	
lowercasep (<i>char</i>)	Función
Devuelve true si <i>char</i> 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 (<i>char</i>)	Función
Devuelve true si <i>char</i> es un carácter en mayúscula.	

74.4 Funciones y variables para cadenas

stringp (*obj*)

Función

Devuelve `true` si *obj* es una cadena. Véase un ejemplo en la introducción.

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`.

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)");
(%o2)          (foo : 42, bar : foo^2 + baz)
```

Véase también `eval_string`.

scopy (*string*)

Función

Devuelve una copia nueva de la cadena *string*.

sdowncase (<i>string</i>)	Función
sdowncase (<i>string, start</i>)	Función
sdowncase (<i>string, start, end</i>)	Función
Convierte caracteres en minúscula a mayúscula. Véase también upcase .	
sequal (<i>string_1, string_2</i>)	Función
Devuelve true si <i>string_1</i> y <i>string_2</i> son dos cadenas de caracteres iguales.	
seqalignore (<i>string_1, string_2</i>)	Función
Igual que sequal pero no diferencia entre minúsculas y mayúsculas..	
sexplode (<i>string</i>)	Función
El nombre sexplode es un seudónimo de la función charlist .	
implode (<i>list</i>)	Función
implode (<i>list, delim</i>)	Función
La función implode admite como entrada una lista de expresiones para luego convertirla en una cadena de caracteres. Si no se utiliza la opción <i>delim</i> para indicar el delimitador, entonces implode no hace uso de ninguno. El valor de <i>delim</i> puede ser cualquier cadena.	
<pre>(%i1) implode(["xx[",3,"]:"],expand((x+y)^3)); (%o1) xx[3]:y^3+3*x*y^2+3*x^2*y+x^3 (%i2) implode(sexplode("stars")," * "); (%o2) s * t * a * r * s (%i3) implode(["One","more","coffee."]," "); (%o3) One more coffee.</pre>	
sinsert (<i>seq, string, pos</i>)	Función
Devuelve la concatenación de las cadenas substring (<i>string, 1, pos - 1</i>), <i>seq</i> y substring (<i>string, pos</i>). Nótese que al primer carácter de <i>string</i> le corresponde la posición 1.	
<pre>(%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.</pre>	
sinvertcase (<i>string</i>)	Función
sinvertcase (<i>string, start</i>)	Función
sinvertcase (<i>string, start, end</i>)	Función
Devuelve la misma cadena <i>string</i> pero con todos sus caracteres desde la posición <i>start</i> hasta <i>end</i> invertidos, esto es, las mayúsculas se convierten en minúsculas y éstas en mayúsculas. Si no se incluye el argumento <i>end</i> , se invierten todos los caracteres desde <i>start</i> hasta el final de la cadena.	
<pre>(%i1) sinvertcase("sInvertCase"); (%o1) SiNVERTcASE</pre>	

slength (<i>string</i>)	Función
Devuelve el número de caracteres de <i>string</i> .	
smake (<i>num, char</i>)	Función
Construye una cadena de longitud <i>num</i> con todos sus caracteres iguales a <i>char</i> .	
(%i1) smake(3, "w");	
(%o1)	www
smismatch (<i>string_1, string_2</i>)	Función
smismatch (<i>string_1, string_2, test</i>)	Función
Devuelve la posición del primer carácter de <i>string_1</i> distinto del correspondiente a <i>string_2</i> . 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 <i>test</i> .	
(%i1) smismatch("seven", "seventh");	
(%o1)	6
split (<i>string</i>)	Función
split (<i>string, delim</i>)	Función
split (<i>string, delim, multiple</i>)	Función
Devuelve la lista de todos los lexemas (<i>tokens</i>) de <i>string</i> . La función split utiliza <i>delim</i> como delimitador, y en caso de no ser utilizado este argumento, será utilizado el espacio en blanco como delimitador por defecto. El argumento <i>multiple</i> 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 <i>multiple</i> se le asigna el valor false , se considerarán 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 (<i>char, string</i>)	Función
Devuelve la posición del primer carácter de <i>string</i> que coincide con <i>char</i> . Al primer carácter de <i>string</i> le corresponde la posición 1. Para cuando se quiera ignorar la diferencia entre mayúsculas y minúsculas, véase ssearch .	
sremove (<i>seq, string</i>)	Función
sremove (<i>seq, string, test</i>)	Función
sremove (<i>seq, string, test, start</i>)	Función
sremove (<i>seq, string, test, start, end</i>)	Función
Devuelve la cadena <i>string</i> pero sin las subcadenas que coinciden con <i>seq</i> . 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 <i>test</i> . Utilíicense <i>start</i> y <i>end</i> para acotar la búsqueda. Al primer carácter de <i>string</i> 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*. Utilíicense *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*. Utilíicense *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 (<i>new, old, string</i>)	Función
ssubstfirst (<i>new, old, string, test</i>)	Función
ssubstfirst (<i>new, old, string, test, start</i>)	Función
ssubstfirst (<i>new, old, string, test, start, end</i>)	Función

Actúa de forma similar a la función `subst`, pero sólo hace la sustitución en la primera coincidencia con *old*.

strim (<i>seq,string</i>)	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 (<i>seq, string</i>)	Función
--------------------------------------	---------

Actúa de forma similar a `strim`, pero sólo recorta en el extremo final de *string*.

trimr (<i>seq, string</i>)	Función
-------------------------------------	---------

Actúa de forma similar a `strim`, pero sólo recorta en el extremo inicial de *string*.

substring (<i>string, start</i>)	Función
substring (<i>string, start, end</i>)	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
```

upcase (<i>string</i>)	Función
---------------------------------	---------

upcase (<i>string, start</i>)	Función
--	---------

upcase (<i>string, start, end</i>)	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]
```

75 unit

75.1 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 TOLEVEL-MACSYMA-EVAL ...  
WARNING: DEFUN/DEFMACRO: redefining function MSETCHK ...  
WARNING: DEFUN/DEFMACRO: redefining function KILL1 ...  
WARNING: DEFUN/DEFMACRO: redefining function NFORMAT ...  
Initializing unit arrays...  
Done.
```

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.

75.2 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;
(%o2)

$$\frac{\text{kg m}}{\text{s}^2}$$


(%i3) dyn;
(%o3)

$$\frac{1}{100000} \frac{\text{kg m}}{\text{s}^2}$$


(%i4) g;
(%o4)

$$\frac{1}{1000} \frac{\text{kg}}{\text{m s}^2}$$


(%i5) centigram*inch/minutes^2;
(%o5)

$$\frac{127}{1800000000000} \frac{\text{kg m}}{\text{s}^2}$$

```

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)
(%i7) N;
(%o7)

$$\frac{1800000000000}{127} \frac{\text{in cg}}{\text{min}^2}$$


(%i8) dyn;
(%o8)

$$\frac{1800000}{127} \frac{\text{in cg}}{\text{min}^2}$$


(%i9) g;
(%o9)
(%i10) centigram*inch/minutes^2;
(%o10)
```

2

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:

Las unidades derivadas también se controlan con esta misma instrucción:

```

(%i17) setunits(N);
(%o17) done
(%i18) N;
(%o18) N
(%i19) dyn;
(%o19) 
$$\frac{1}{100000} \text{ (N)}$$

(%i20) kg*m/s^2;
(%o20) N
(%i21) centigram*inch/minutes^2;
(%o21) 
$$\frac{127}{180000000000000} \text{ (N)}$$


```

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;
(%o24) (-----) (dyn)
          127
          18000000
```

Para desactivar una unidad se utiliza la instrucción `uforget`:

```
(%i26) uforget(dyn);                                false
(%o26)
(%i27) kg*m/s^2;                                 kg m
(%o27)                                          -----
                                         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`:

La instrucción `uforget` opera sobre dimensiones, no sobre unidades, de modo que valdrá para cualquier unidad de una dimensión concreta. La propia dimensión es un 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
          -----
          2
          s
(%i4) convert(kg*m/s^2,[g,inch,minute]);
'rat' replaced 39.37007874015748 by 5000/127 = 39.37007874015748
          180000000000 %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 TOLEVEL-MACSYMA-EVAL ...■  
WARNING: DEFUN/DEFMACRO: redefining function MSETCHK ...  
WARNING: DEFUN/DEFMACRO: redefining function KILL1 ...  
WARNING: DEFUN/DEFMACRO: redefining function NFORMAT ...  
Initializing unit arrays...  
Done.  
User defaults found...  
User defaults initialized.  
(%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) (6241509596477042688000) (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;
(%o12) kg m
-----  
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:

(por defecto)	0 - none. Only base units 1 - kilo, centi, milli 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`.

76 zeilberger

76.1 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)`.

76.1.0.1 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 antiderivada hipergeométrica, esto es, el término hipergeométrico tal que

$$F_k = f_{k+1} - f_k.$$

76.1.0.2 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

$$a_0 F(n, k) + \dots + a_d F(n + d, k) = \text{Delta}_K(R(n, k) F(n, k))$$

donde Delta_k es el k -ésimo operador diferencia hacia adelante, esto es, $\text{Delta}_k(t_k) := t_k(k+1) - t_k$.

76.1.1 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`.

76.2 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)           -----
                           (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 recurrencia de orden d para $F_{\{n,k\}}$.

El algoritmo devuelve una secuencia $[s_1, s_2, \dots, s_m]$ de soluciones, cada una de las cuales tiene la forma

$[R(n, k), [a_0, a_1, \dots, a_d]]$.

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 $[s_1, s_2, \dots, s_m]$ de soluciones, cada una de las cuales tiene la forma

$[R(n, k), [a_0, a_1, \dots, a_d]]$.

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: <code>true</code>	
Si <code>gosper_in_zeilberger</code> vale <code>true</code> , la función <code>Zeilberger</code> llama a la función <code>Gosper</code> antes de llamar a <code>parGosper</code> . En caso contrario, <code>Zeilberger</code> invoca inmediatamente a <code>parGosper</code> .	
trivial_solutions	Variable opcional
Valor por defecto: <code>true</code>	
Si <code>trivial_solutions</code> vale <code>true</code> , la función <code>Zeilberger</code> devuelve soluciones triviales.	
mod_test	Variable opcional
Valor por defecto: <code>false</code>	
Si <code>mod_test</code> vale <code>true</code> , la función <code>parGosper</code> ejecuta una prueba modular para descartar sistemas sin soluciones.	
modular_linear_solver	Variable opcional
Valor por defecto: <code>linsolve</code>	
La variable <code>modular_linear_solver</code> guarda el nombre de la función que deberá ser llamada por la prueba modular de <code>parGosper</code> para resolver sistemas lineales.	
ev_point	Variable opcional
Valor por defecto: <code>big_primes[10]</code>	
La variable <code>ev_point</code> guarda el valor para el que debe evaluarse n durante la ejecución de la prueba modular de <code>parGosper</code> .	
mod_big_prime	Variable opcional
Valor por defecto: <code>big_primes[1]</code>	
La variable <code>mod_big_prime</code> guarda el módulo utilizado por la prueba modular de <code>parGosper</code> .	
mod_threshold	Variable opcional
Valor por defecto: 4	
La variable <code>mod_threshold</code> es el máximo orden que ensaya la prueba modular de <code>parGosper</code> .	

77 Índice de Funciones y Variables

Apéndice A Índice de Funciones y Variables

!	.	38
!	36	
!!	37	
#	/	32
#	/	
%	:	
%	:	38
%%	:	39
%c	::=	40
%d	:=	41
%e	189	
%e_to_numlog	193	
%edispflag	132	
%emode	79	
%enumer	79	
%f	221	
%gamma	414	
%i	189	
%iargs	198	
%k1	559	
%k2	559	
%m	221	
%phi	189	
%pi	190	
%piargs	197	
%rnum_list	265	
%s	207	
%th	132	
%unitexpand	890	
%w	221	
,	[
,	[326
,	^	
*	^	32
*	^~	36
*		
**		
+	-	130
+	--	129
+		
-	-	
-	'	700
-	''	702
,		

		antidiff	234
.....	362	AntiDifference	892
~		antisymmetric	46
~	361	append	467
A		appendfile	133
abasep	396	apply	508
abs	45	apply1	449
absboxchar	132	apply2	449
absint	294	applyb1	449
absolute_real_time	439	apropos	9
acos	198	args	441
acosh	198	arithmetic	836
acot	198	arithsum	837
acoth	198	array	297
acsc	198	arrayapply	297
acsch	198	arrayinfo	297
activate	161	arraymake	299
activecontexts	161	arrays	300
adapt_depth	661	ascii	873
add_edge	737	asec	199
add_edges	738	asech	199
add_vertex	738	asin	199
add_vertices	738	asinh	199
addcol	306	askexp	99
additive	45	askinteger	99
addmatrices	775	asksign	99
addrw	306	assoc	467
adim	395	assoc_legendre_p	816
adjacency_matrix	722	assoc_legendre_q	817
adjoin	479	assume	161
adjoint	306	assume_external_byte_order	804
af	396	assume_pos	162
aform	395	assume_pos_pred	163
agd	837	assumescalar	162
airy_ai	207	asymbol	395
airy_bi	207	at	69
airy_dai	207	atan	199
airy_db1	207	atan2	199
alg_type	395	atanh	199
algebraic	167	atensimp	395
algepsilon	159	atom	467
algexact	265	atomgrad	234
algsys	265	atrig1	199
alias	19	atvalue	234, 235
aliases	441	augcoefmatrix	306
all_dot simp_denoms	331	augmented_lagrangian_method	551
allbut	45	av	396
allroots	267	average_degree	723
allsym	346	axis_3d	651
alphabetic	441	axis_bottom	650
alphacharp	873	axis_left	650
alphanumericp	873	axis_right	650
and	44	axis_top	650
antid	233		

B

backsubst	268
backtrace	531
bars	669
barsplot	581
bashindices	300
batch	133
batchload	133
bc2	283
bdvac	382
belln	479
berlefact	168
bern	411
bernpoly	411
bessel_i	206
bessel_j	205
bessel_k	206
bessel_y	205
besselexpand	206
beta	210
beta_args_sum_to_integer	218
beta_expand	217
beta_incomplete	211
beta_incomplete_generalized	215
beta_incomplete_regularized	214
bezout	168
bfallroots	268
bffac	159
bfhzeta	411
bffloat	159
bfloatp	159
bfpsi	159
bfpsi0	159
bftorat	159
bftrunc	160
bfzeta	411
biconected_components	723
binomial	411
bipartition	723
block	509
blockmatrixp	775
bode_gain	553
bode_phase	554
border	655
bothcoef	168
boundaries_array	676
box	70
boxchar	71
boxplot	583
break	510
breakup	268
bug_report	6
build_info	6
buildq	505
burn	412

C

cabs	46
canform	347
canten	346
cardinality	480
carg	71
cartan	235
cartesian_product	480
catch	510
cauchysum	397
cbffac	160
cbrange	631
cbtics	638
cdf_bernoulli	622
cdf_beta	609
cdf_binomial	620
cdf_cauchy	618
cdf_chi2	600
cdf_continuous_uniform	611
cdf_discrete_uniform	625
cdf_exp	605
cdf_f	603
cdf_gamma	608
cdf_geometric	624
cdf_gumbel	619
cdf_hypergeometric	626
cdf_laplace	617
cdf_logistic	612
cdf_lognormal	607
cdf_negative_binomial	627
cdf_noncentral_chi2	602
cdf_noncentral_student_t	598
cdf_normal	595
cdf_pareto	612
cdf_poisson	621
cdf_rank_sum	866
cdf_rayleigh	614
cdf_signed_rank	866
cdf_student_t	596
cdf_weibull	613
cdisplay	383
ceiling	46
central_moment	570
cequal	873
cequalignore	873
cf	412
cfdisrep	413
cfexpand	413
cflength	414
cframe_flag	388
cgeodesic	382
cgreaterp	873
cgreaterpignore	873
changename	337
changevar	245
chaosgame	687
charat	875
charfun	47

charfun2	757	constvalue	705
charlist	875	cont2part	419
charp	873	content	168
charpoly	306	context	164
chebyshev_t	817	contexts	164
chebyshev_u	817	continuous_freq	565
check_overlaps	330	contortion	380
checkdiv	382	contour	662
cholesky	776	contour_levels	662
christof	372	contour_plot	107
chromatic_index	723	contract	340, 419
chromatic_number	723	contract_edge	738
cint	873	contragrad	381
circulant_graph	718	contrib_ode	557
clear_edge_weight	724	convert	884
clear_rules	464	coord	350
clear_vertex_label	724	copy	775
clebsch_graph	719	copy_graph	718
clessp	874	copylist	468
clesspignore	874	copymatrix	308
close	871	cor	577
closefile	134	cos	199
cmetric	369	cosh	199
cnonmet_flag	388	cosnpiflag	294
coeff	168	cot	199
coefmatrix	307	coth	199
cograd	381	cov	575
col	307	cov1	576
collectterms	835	covdiff	353
color	658	covect	307
colorbox	652	covers	837
columnnop	775	create_graph	717
columns	663	create_list	468
columnspace	775	csc	199
columnswap	775	csch	199
columnvector	307	csetup	369
combination	838	cspline	758
combine	168	ct_coords	391
commutative	47	ct_coordsys	369
comp2pui	419	ctaylor	374
compare	47	ctaypov	388
compfile	510	ctaypt	388
compile	511	ctayswitch	388
compile_file	529	ctayvar	388
complement_graph	719	ctorsion_flag	388
complete_bipartite_graph	719	ctransform	380
complete_graph	719	ctranspose	776
components	340	ctrssimp	387
concat	346	cube_graph	719
concat	134	cunlisp	874
conjugate	308	current_let_rule_package	450
conmetderiv	350	cv	570
connect_vertices	738	cycle_digraph	719
connected_components	724	cycle_graph	719
cons	467	cylindrical	674
constant	71		
constantp	71		
constituent	874		

D

data_file_name	647	dimension	270
dblint	246	dimensionless	712
deactivate	165	dimensions	711
debugmode	19	dimensions_as_list	711
declare	72	direct	420
declare_constvalue	705	discrete_freq	565
declare_dimensions	709	disjoin	481
declare_fundamental_dimensions	709	disjointp	481
declare_fundamental_units	710	disolate	77
declare_qty	707	disp	134
declare_translated	529	dispcon	135
declare_unit_conversion	708	dispflag	270
declare_units	706	dispform	77
declare_weights	329	dispfun	514
decsym	346	dispJordan	586
default_let_rule_package	450	display	135
defcon	339	display_format_internal	135
define	511	display2d	135
define_variable	513	disprule	452
defint	247	disptterms	135
defmatch	450	distrib	78
defrule	452	divide	169
deftaylor	397	divisors	481
degree_sequence	725	divsum	414
del	236	dkummer_m	559
delay	647	dkummer_u	559
delete	468	dlange	764
deleten	387	do	531
delta	236	doallmxops	309
demo	9	dodecahedron_graph	719
demoivre	99	domain	99
denom	169	domxexpt	309
dependencies	236	domxxmlops	310
depends	236	domxnctimes	310
derivabbrev	237	dontfactor	310
derivdegree	238	doscmxops	310
derivlist	238	doscmxplus	310
derivsubst	238	dot0nscsimp	310
describe	11	dotassoc	310
desolve	283	dotconstrules	310
determinant	308	dotdistrib	310
detout	308	dotexptsimp	311
dgauss_a	559	dotident	311
dgauss_b	559	dotproduct	776
dgeev	761	dotscrules	311
dgesvd	762	dotsimp	330
diag	585	dpart	78
diag_matrix	776	draw	679
diagmatrix	309	draw_file	680
diagmatrixp	383	draw_graph	741
diagmetric	387	draw_graph_program	744
diameter	724	draw2d	680
diff	238, 239, 347	draw3d	680
digitcharp	874	dscalar	382
dim	387		
dimacs_export	740		
dimacs_import	740		

E

echelon.....	311	every.....	483
edge_coloring.....	724	evflag.....	22
edge_connectivity.....	725	evfun.....	23
edges.....	725	evolution.....	687
eigens_by_jacobi.....	776	evolution2d.....	687
eigenvalues.....	311	evundiff.....	348
eigenvectors.....	312	example.....	12
eighth.....	469	exp.....	79
einstein.....	373	expand.....	100
eivals.....	311	expandwrt.....	100
eivects.....	312	expandwrt_denom.....	100
elapsed_real_time.....	440	expandwrt_factored.....	101
elapsed_run_time.....	440	expintegral_chi.....	219
ele2comp.....	421	expintegral_ci.....	219
ele2polynome.....	421	expintegral_e.....	219
ele2pui.....	422	expintegral_e1.....	219
elem.....	422	expintegral_ei.....	219
elementp.....	482	expintegral_li.....	219
eliminate.....	169	expintegral_shi.....	219
ellipse.....	670	expintegral_si.....	219
elliptic_e.....	228	expintexpand.....	220
elliptic_ec.....	229	expintrep.....	219
elliptic_eu.....	228	explicit.....	671
elliptic_f.....	228	explose.....	422
elliptic_kc.....	229	expon.....	101
elliptic_pi.....	229	exponentialize.....	101
ematrix.....	314	expop.....	101
empty_graph.....	719	express.....	239
emptyp.....	482	expt.....	137
endcons.....	469	exptdispflag.....	137
enhanced3d.....	652	exptsubst.....	79
entermatrix.....	314	exsec.....	837
entertensor.....	337	extdiff.....	362
entier.....	48	extract_linear_equations.....	330
eps_height.....	649	extremal_subset.....	484
eps_width.....	648	ezgcd.....	169
epsilon_lp.....	831		
equal.....	48		
equalp.....	293	F	
equiv_classes.....	482	f90.....	713
erf.....	220	faceexpand.....	170
erf_generalized.....	220	facsum.....	834
erf_representation.....	220	facsum_combine.....	834
erfc.....	220	factcomb.....	170
erfflag.....	247	factlim.....	101
erfi.....	220	factor.....	170
errcatch.....	534	factorfacsum.....	835
error.....	535	factorflag.....	172
error_size.....	136	factorial.....	415
error_syms.....	136	factorout.....	172
errormsg.....	535	factorsum.....	172
euler.....	414	facts.....	165
ev.....	19	false.....	189
ev_point.....	894	fast_central_elements.....	330
eval.....	51	fast_linsolve.....	329
eval_string.....	875	fasttimes.....	173
evenp.....	51	fb.....	390
		feature.....	437

featurep	438	freeof	79
features	165	freshline	871
fft	288	fresnel_c	220
fib	415	fresnel_s	220
fibtophi	415	from_adjacency_matrix	719
fifth	469	frucht_graph	719
file bgcolor	647	full_listify	485
file_name	646	fullmap	51
file_output_append	132	fullmapl	51
file_search	137	fullratsimp	173
file_search_demo	138	fullratsubst	174
file_search_lisp	138	fullsetify	485
file_search_maxima	138	funcsolve	270
file_type	138	functions	516
filename_merge	137	fundamental_dimensions	709
fill_color	659	fundamental_units	711
fill_density	659	fundef	516
fillarray	300	funmake	517
filled_func	654	funp	294
find_root	292		
find_root_abs	292		
find_root_error	292		
find_root_rel	292	gamma	208
findde	380	gamma_incomplete	209
first	469	gamma_incomplete_generalized	209
fix	51	gamma_incomplete_regularized	209
flatten	484	gammalim	209
flength	871	gauss_a	559
flipflag	339	gauss_b	559
float	160	gaussprob	837
float2bf	160	gcd	174
floatnump	160	gcdex	175
floor	49	gcddivide	836
flower_snark	719	gcfac	840
flush	349	gcfactor	175
flush1deriv	352	gd	837
flushd	349	gdet	388
flushnd	350	gen_laguerre	817
font	634	genfact	81
font_size	635	genindex	442
for	535	genmatrix	315
forget	165	gensumnum	442
fortindent	290	geomap	677
fortran	290	geometric	836
fortspaces	291	geometric_mean	573
fourcos	294	geosum	837
fourexpand	294	get	470
fourier	294	get_edge_weight	725
fourint	295	get_lu_factors	777
fourintcos	295	get_pixel	683
fourintsin	295	get_tex_environment	155
foursimp	294	get_tex_environment_default	155
foursin	295	get_vertex_label	726
fourth	470	gfactor	175
fposition	871	gfactorsum	175
fpprec	160	ggf	715
fpprintprec	160	GGFCFMAX	715
frame_bracket	377	GGFINFINITY	715

girth.....	727	hodge.....	363
global_variances.....	576	horner.....	291
globalsolve.....	271	hstruve.....	221
gnuplot_close.....	128	hypergeometric_representation.....	221
gnuplot_file_name.....	646		
gnuplot_replot.....	128		
gnuplot_reset.....	128		
gnuplot_restart.....	128	ibase.....	140
gnuplot_start.....	128	ic_convert.....	364
go.....	535	ic1.....	284
Gosper.....	892	ic2.....	284
gosper_in_zeilberger.....	894	icc1.....	356
GosperSum.....	892	icc2.....	357
gr2d.....	665	ichr1.....	352
gr3d.....	665	ichr2.....	353
gradef.....	240, 241	icosahedron_graph.....	720
gradefs.....	241	icounter.....	343
gramschmidt.....	316	icurvature.....	353
graph_center.....	726	ident.....	317
graph_charpoly.....	726	identfor.....	778
graph_eigenvalues.....	726	identity.....	485
graph_order.....	727	idiff.....	348
graph_periphery.....	726	idim.....	352
graph_product.....	719	idummy.....	342
graph_size.....	726	idummyx.....	343
graph_union.....	720	ieqn.....	272
graph6_decode.....	740	ieqnprint.....	272
graph6_encode.....	740	if.....	535
graph6_export.....	740	ifactors.....	415
graph6_import.....	740	ifb.....	356
grid.....	636	ifc1.....	357
grid_graph.....	720	ifc2.....	357
grind.....	138	ifg.....	358
grobner_basis.....	329	ifgi.....	358
grotzch_graph.....	720	ifr.....	357
		iframe_bracket_form.....	358
		ifri.....	358
		ifs.....	687
		igeodesic_coords.....	354
halfangles.....	200	igeowedge_flag.....	363
hamilton_cycle.....	727	ikt1.....	359
hamilton_path.....	727	ikt2.....	359
hankel.....	778	ilt.....	247
hankel_1.....	206	image.....	675
hankel_2.....	206	imagpart.....	81
harmonic.....	837	imetric.....	352
harmonic_mean.....	573	implicit.....	672
hav.....	837	implicit_derivative.....	753
head_angle.....	656	in_neighbors.....	728
head_both.....	655	in_netmath.....	107
head_length.....	656	inchar.....	141
head_type.....	657	ind.....	189
heawood_graph.....	720	indexed_tensor.....	340
hermite.....	817	indices.....	337
hessian.....	778	induced_subgraph.....	720
hgfred.....	223	inf.....	189, 442
hilbert_matrix.....	778	inference_result.....	849
hipow.....	176	inferencep.....	850
histogram.....	579		

H

halfangles	200
hamilton_cycle	727
hamilton_path	727
hankel	778
hankel_1	206
hankel_2	206
harmonic	837
harmonic_mean	573
hav	837
head_angle	656
head_both	655
head_length	656
head_type	657
heawood_graph	720
hermite	817
hessian	778
hgfred	223
hilbert_matrix	778
hipow	176
histogram	579

I

ibase.....	140
ic_convert.....	364
ic1.....	284
ic2.....	284
icc1.....	356
icc2.....	357
ichr1.....	352
ichr2.....	353
icosahedron_graph.....	720
icounter.....	343
icurvature.....	353
ident.....	317
identfor.....	778
identity.....	485
idiff.....	348
idim.....	352
idummy.....	342
idummyx.....	343
ieqn.....	272
ieqnprint.....	272
if.....	535
ifactors.....	415
ifb.....	356
ifc1.....	357
ifc2.....	357
ifg.....	358
ifgi.....	358
ifr.....	357
iframe_bracket_form.....	358
ifri.....	358
ifs.....	687
igeodesic_coords.....	354
igeowedge_flag.....	363
ikt1.....	359
ikt2.....	359
ilt.....	247
image.....	675
imagpart.....	81
imetric.....	352
implicit.....	672
implicit_derivative.....	753
in_neighbors.....	728
in_netmath.....	107
inchar.....	141
ind.....	189
indexed_tensor.....	340
indices.....	337
induced_subgraph.....	720
inf	189, 442
inference_result.....	849
inferencep	850

infeval.....	24	ishow.....	337
infinity.....	189, 442	isolate.....	83
infix.....	81	isolate_wrt_times.....	84
inflag.....	82	isomorphism.....	727
infolists.....	442	isqrt.....	53
init_atensor.....	394	items_inference.....	850
init_ctensor.....	371	itr.....	359
inm.....	358		
inmc1.....	358		
inmc2.....	358		
innerproduct.....	317	J	
inpart.....	83	jacobi.....	416
inprod.....	317	jacobi_cd.....	227
inrt.....	416	jacobi_cn.....	226
integer_partitions.....	486	jacobi_cs.....	227
integerp.....	443	jacobi_dc.....	227
integrate.....	248	jacobi_dn.....	226
integrate_use_rootsof.....	252	jacobi_ds.....	227
integration_constant.....	251	jacobi_nc.....	227
integration_constant_counter.....	251	jacobi_nd.....	227
intersect.....	486	jacobi_ns.....	227
intersection.....	486	jacobi_p.....	817
intervalp.....	817	jacobi_sc.....	227
intfaclim.....	176	jacobi_sd.....	227
inttopois.....	408	jacobi_sn.....	226
intosum.....	101	jacobian.....	779
inv_mod.....	416	JF.....	585
invariant1.....	382	join.....	470
inverse_fft.....	287	jordan.....	586
inverse_jacobi_cd.....	228	julia.....	688
inverse_jacobi_cn.....	227		
inverse_jacobi_cs.....	228		
inverse_jacobi_dc.....	228		
inverse_jacobi_dn.....	227	K	
inverse_jacobi_ds.....	228	kdeps.....	343
inverse_jacobi_nc.....	228	kdelta.....	343
inverse_jacobi_nd.....	228	keepfloat.....	176
inverse_jacobi_ns.....	227	key.....	661
inverse_jacobi_sc.....	227	kill.....	24, 25
inverse_jacobi_sd.....	227	killcontext.....	165
inverse_jacobi_sn.....	227	kinvariant.....	390
invert.....	317	kostka.....	423
invert_by_lu.....	779	kron_delta.....	487
ip_grid.....	664	kronecker_product.....	779
ip_grid_in.....	664	kt.....	390
is.....	51	kummer_m.....	559
is_biconnected.....	728	kummer_u.....	559
is_bipartite.....	728	kurtosis.....	574
is_connected.....	728	kurtosis_bernoulli.....	623
is_digraph.....	728	kurtosis_beta.....	610
is_edge_in_graph.....	729	kurtosis_binomial.....	621
is_graph.....	729	kurtosis_chi2.....	602
is_graph_or_digraph.....	729	kurtosis_continuous_uniform.....	611
is_isomorphic.....	729	kurtosis_discrete_uniform.....	626
is_planar.....	730	kurtosis_exp.....	607
is_sconnected.....	730	kurtosis_f.....	604
is_tree.....	730	kurtosis_gamma.....	609
is_vertex_in_graph.....	730	kurtosis_geometric.....	625
		kurtosis_gumbel.....	619
		kurtosis_laplace.....	618

kurtosis_logistic	612	line_graph	720
kurtosis_lognormal	608	line_type	660
kurtosis_negative_binomial	628	line_width	659
kurtosis_noncentral_chi2	603	linear	102, 836
kurtosis_noncentral_student_t	599	linear_program	831
kurtosis_normal	596	linear_solver	893
kurtosis_pareto	613	linearinterpol	757
kurtosis_poisson	622	linechar	143
kurtosis_rayleigh	617	linel	143
kurtosis_student_t	597	linenum	26
kurtosis_weibull	614	linsolve	273
		linsolve_params	274
		linsolvewarn	274
		lispdisp	143
label	670	list_correlations	578
label_alignment	658	list_nc_monomials	330
label_orientation	658	listarith	471
labels	26	listarray	301
lagrange	755	listconstvars	84
laguerre	817	listdummyvars	84
lambda	518	listify	489
lambert_w	224	listoftens	337
laplace	241	listofvars	85
laplacian_matrix	730	listp	471, 779
lassociative	101	lmax	53
last	471	lmin	53
lbfgs	765	lmxchar	317
lbfgs_ncorrections	770	load	143
lbfgs_nfeval_max	770	loadfile	144
lc_l	345	loadprint	144
lc_u	345	local	521
lc2kdt	344	locate_matrix_entry	779
lcharp	874	log	194
lcm	416	log_gamma	209
ldefint	253	logabs	194
ldisp	142	logand	836
ldisplay	142	logarc	195
legendre_p	817	logcb	632
legendre_q	817	logconcoeffp	195
leinstein	373	logcontract	195
length	471	logexpand	195
let	453	lognegint	195
let_rule_packages	455	lognumer	196
letrat	454	logor	836
letrules	454	logsimp	196
letsimp	455	logx	632
levi_civita	344	logxor	836
lfg	389	logy	632
lfreeof	85	logz	632
lg	389	lopow	85
lgtreillis	423	lorentz_gauge	354
lhospitallim	231	lowercasep	874
lhs	272	lpart	85
li	193	lratsubst	176
liediff	348	lreduce	489
limit	231	lriem	389
limsubst	231	lriemann	373
Lindstedt	771	lsquares_estimates	787

lsquares_estimates_approximate.....	789	max_clique	731
lsquares_estimates_exact	788	max_degree	731
lsquares_mse	790	max_flow	731
lsquares_residual_mse	792	max_independent_set	732
lsquares_residuals	791	max_matching	732
lstruve.....	221	max_ord	893
lsum.....	96	maxapplydepth	102
ltreillis	423	maxapplyheight	102
lu_backsub	780	maxi	571
lu_factor	780	maxima_tempdir	438
		maxima_userdir	438
		maximize_lp	831
		maxnegex	102
		maxposex	102
m1pbranch	443	maxpsifracdenom	218
macroexpand	506	maxpsifracnum	218
macroexpand1	507	maxpsinegint	218
macroexpansion	521	maxpsiposint	218
macros	507	maxtayorder	398
mainvar	102	maybe	52
make_array	303	mean	568
make_graph	720	mean_bernoulli	622
make_level_picture	681	mean_beta	610
make_poly_continent	685	mean_binomial	620
make_poly_country	685	mean_chi2	600
make_polygon	684	mean_continuous_uniform	611
make_random_state	54	mean_deviation	572
make_rgb_picture	682	mean_discrete_uniform	625
make_transform	127	mean_exp	605
makebox	350	mean_f	604
makefact	218	mean_gamma	609
makegamma	210	mean_geometric	624
makelist	471	mean_gumbel	619
makeOrders	797	mean_hypergeometric	626
makeset	489	mean_laplace	618
mandelbrot	688	mean_logistic	612
manual_demo	13	mean_lognormal	608
map	536	mean_negative_binomial	627
mapatom	537	mean_noncentral_chi2	603
maperror	537	mean_noncentral_student_t	599
maplist	537	mean_normal	596
mapprint	537	mean_pareto	613
mat_cond	782	mean_poisson	621
mat_fullunblocker	782	mean_rayleigh	615
mat_function	588	mean_student_t	597
mat_norm	782	mean_weibull	614
mat_trace	782	median	572
mat_unblocker	782	median_deviation	573
matchdeclare	455	member	471
matchfix	457	mesh	672
matrix	317	method	558
matrix_element_add	320	metricexpandall	889
matrix_element_mult	320	min	53
matrix_element_transpose	321	min_degree	732
matrix_size	782	min_edge_cut	732
matrixmap	320	min_vertex_cover	733
matrixxp	320, 782	min_vertex_cut	733
mattrace	322	minf	189
max	53		

mini.....	570	nmc.....	390
minimalPoly.....	587	noeval.....	103
minimize_lp.....	832	nolabels.....	26
minimum_spanning_tree.....	733	noncentral_moment.....	569
minor.....	322	nonegative_lp.....	832
mnewton.....	799	nonmetricity.....	380
mod.....	53	nonnegintegerp.....	783
mod_big_prime.....	894	nonscalar.....	323
mod_test.....	894	nonscalarp.....	323
mod_threshold.....	894	nonzeroandfreeof.....	836
mode_check_errorp.....	524	not.....	45
mode_check_warnp.....	524	notequal.....	50
mode_checkp.....	524	noun.....	103
mode_declare.....	524	noundisp.....	103
mode_identity.....	524	nounify.....	86
ModeMatrix.....	587	nouns.....	103
modular_linear_solver.....	894	np.....	390
modulus.....	177	npi.....	390
moebius.....	490	nptetrad.....	377
mon2schur.....	423	nroots.....	275
mono.....	330	nterms.....	86
monomial_dimensions.....	330	ntermst.....	383
multi_elem.....	424	nthroot.....	275
multi_orbit.....	424	nticks.....	660
multi_pui.....	424	ntrig.....	200
multinomial.....	424	nullity.....	783
multinomial_coeff.....	491	nullspace.....	783
multiplicative.....	102	num.....	177
multiplicities.....	274	num_distinct_partitions.....	491
multiplot_mode.....	681	num_partitions.....	492
multsym.....	424	numbered_boundaries.....	683
multthru.....	85	numberp.....	444
mycielski_graph.....	721	numer.....	103
myoptions	26	numerval	104
		numfactor	219
		nusum.....	399
		nzeta.....	224
		nzetai.....	224
		nzetar.....	224

N

natural_unit	712
nc_degree	329
ncexpt	322
ncharpoly	322
negative_picture	683
negdistrib	103
negsumdispflag	103
neighbors	733
new_graph	721
newcontext	166
newdet	323
newline	871, 874
newton	293
newtoneyepsilon	799
newtonmaxiter	799
next_prime	416
nextlayerfactor	834
niceindices	398
niceindicespref	399
ninth	472
nm	390

O

obase	144
odd_girth	733
oddp	54
ode_check	558
ode2	284
odelin	557
op	86
opena	871
opena_binary	804
openr	871
openr_binary	804
openw	871
openw_binary	804
operatorp	87
opproperties	104
ops subst	104, 807

optimize.....	87	pdf_normal	595
optimprefix.....	87	pdf_pareto	612
optionset.....	27	pdf_poisson	621
or.....	44	pdf_rank_sum	866
orbit.....	425	pdf_rayleigh	614
orbits.....	689	pdf_signed_rank	866
ordergreat.....	87	pdf_student_t	596
ordergreatp.....	88	pdf_weibull	613
orderless.....	87	pdf_width	649
orderlessp.....	88	pearson_skewness	575
orthogonal_complement	783	permanent	323
orthopoly_recur	818	permut	425
orthopoly_returns_intervals	818	permutation	838
orthopoly_weight	818	permutations	492
out_neighbors	733	petersen_graph	721
outative	104	petrov	378
outchar	145	pfeformat	145
outermap	538	pic_height	648
outofpois	408	pic_width	648
P			
packagefile	145	pickapart	90
pade	400	picture_equalp	682
palette	651	picturep	682
parametric	674	piece	92
parametric_surface	679	piechart	583
parGosper	893	planar_embedding	734
parse_string	875	playback	27
part	89	plog	196
part2cont	425	plot_options	115
partfrac	416	plot2d	107
partition	90	plot3d	123
partition_set	492	plotdf	821
partpol	425	plsquares	793
partswitch	90	pochhammer	818
path_digraph	721	pochhammer_max_index	819
path_graph	721	point_size	653
pdf_bernoulli	622	point_type	653
pdf_beta	609	points	665, 666
pdf_binomial	620	points_joined	654
pdf_cauchy	618	poisdiff	408
pdf_chi2	600	poisexpt	408
pdf_continuous_uniform	610	poisint	409
pdf_discrete_uniform	625	poislim	409
pdf_exp	604	poisson	409
pdf_f	603	poissubst	409
pdf_gamma	608	poistimes	409
pdf_geometric	624	polar	673
pdf_gumbel	619	polarform	92
pdf_height	649	polartorect	287
pdf_hypergeometric	626	poly_add	747
pdf_laplace	617	poly_buchberger	749
pdf_logistic	611	poly_buchberger_criterion	749
pdf_lognormal	607	poly_coefficient_ring	746
pdf_negative_binomial	627	poly_colon_ideal	750
pdf_noncentral_chi2	602	poly_content	748
pdf_noncentral_student_t	598	poly_depends_p	750

poly_elimination_ideal	750	printprops	28
poly_elimination_order	746	prodrac	426
poly_exact_divide	749	product	92
poly_expand	748	product_use_gamma	846
poly_expt	748	programmode	275
poly_gcd	750	prompt	28
poly_grobner	750	properties	444
poly_grobner_algorithm	747	proportional_axes	629
poly_grobner_debug	746	props	444
poly_grobner_equal	750	propvars	444
poly_grobner_member	751	psexpand	402
poly_grobner_subsetp	750	psi	218, 377
poly_ideal_intersection	750	ptriangularize	784
poly_ideal_polysaturation	751	pui	426
poly_ideal_polysaturation1	751	pui_direct	427
poly_ideal_saturation	751	pui2comp	426
poly_ideal_saturation1	751	pui2ele	427
poly_lcm	750	pui2polynome	427
poly_minimization	749	puireduc	428
poly_monomial_order	746	put	445
poly_multiply	747		
poly_normal_form	749		
poly_normalize	748		
poly_normalize_list	749	qput	445
poly_polysaturation_extension	751	qrangle	572
poly_primary_elimination_order	746	qty	707
poly_primitive_part	747	quad_qag	255
poly_pseudo_divide	748	quad_qagi	257
poly_reduced_grobner	750	quad_qags	256
poly_reduction	749	quad_qawc	259
poly_return_term_list	746	quad_qawf	260
poly_s_polynomial	747	quad_qawo	261
poly_saturation_extension	751	quad_qaws	262
poly_secondary_elimination_order	746	quantile	571
poly_subtract	747	quantile_bernoulli	622
poly_top_reduction_only	747	quantile_beta	610
polydecomp	177	quantile_binomial	620
Polygon	668	quantile_cauchy	618
polymod	53	quantile_chi2	600
polynome2ele	425	quantile_continuous_uniform	611
polynomialp	783	quantile_discrete_uniform	625
polytocompanion	784	quantile_exp	605
posfun	104	quantile_f	603
power_mod	417	quantile_gamma	609
powerdisp	401	quantile_geometric	624
powers	92	quantile_gumbel	619
powerseries	401	quantile_hypergeometric	626
powerset	493	quantile_laplace	617
pred	54	quantile_logistic	612
prederror	537	quantile_lognormal	607
prev_prime	417	quantile_negative_binomial	627
primep	417	quantile_noncentral_chi2	602
primep_number_of_tests	417	quantile_noncentral_student_t	598
print	146	quantile_normal	595
print_graph	734	quantile_pareto	613
printf	872	quantile_poisson	621
printfile	146	quantile_rayleigh	615
printpois	409	quantile_student_t	597

quantile_weibull.....	613	rateinstein.....	388
quartile_skewness.....	575	ratepsilon.....	181
quit.....	28	rateexpand.....	182
qunit.....	417	ratfac.....	182
quotient	178	ratinterpol.....	760
		rational.....	835
		rationalize.....	55
		ratmx.....	323
		ratnumer.....	183
		ratnump.....	183
		ratp.....	183
		ratprint.....	183
		ratriemann.....	389
		ratsimp.....	183
		ratsimpexpons.....	184
		ratsubst.....	184
		ratvars.....	185
		ratweight.....	185
		ratweights.....	186
		ratweyl.....	389
		ratwtlvl.....	186
		read.....	147
		read_array.....	802
		read_binary_array.....	805
		read_binary_list.....	805
		read_binary_matrix.....	804
		read_hashed_array.....	803
		read_list.....	803
		read_matrix.....	802
		read_nested_list.....	803
		read_xpm.....	683
		readline.....	872
		readonly.....	147
		readonly.....	275
		realpart.....	93
		realroots.....	276
		rearray.....	304
		rectangle.....	669
		rectform.....	93
		recttopolar.....	287
		rediff.....	348
		reduce_consts.....	839
		reduce_order.....	843
		refcheck.....	546
		region_boundaries.....	683
		rem.....	445
		remainder.....	186
		remarray.....	304
		rembox.....	93
		remcomps.....	342
		remcon.....	339, 340
		remcoord.....	350
		remfun.....	293, 294
		remfunction.....	28
		remlet.....	459
		remove.....	445
		remove_dimensions.....	709
		remove_edge.....	739

remove_fundamental_dimensions	709
remove_fundamental_units	710
remove_vertex	739
rempart	835
remrule	459
remsym	347
remvalue	446
rename	338
reset	28
residue	253
resolvante	428
resolvante_alternee1	432
resolvante_bipartite	432
resolvante_diedrale	432
resolvante_klein	432
resolvante_klein3	432
resolvante_produit_sym	433
resolvante_unitaire	433
resolvante_vierer	433
rest	472
resultant	186
return	537
reveal	147
reverse	472
revert	402
revert2	402
rgb2level	683
rhs	276
ric	389
ricci	372
riem	389
riemann	373
rinvariant	374
risch	253
rk	689
rmxchar	148
rncombine	446
romberg	827
rombergabs	828
rombergit	829
rombergmin	829
rombergtol	829
room	438
rootscommode	277
rootscontract	277
rootsepsilon	278
rot_horizontal	645
rot_vertical	645
round	57
row	323
rowop	784
rowswap	785
rreduce	493
run_testsuite	5

S

save	148, 149
savedef	149
savefactors	186
scalarmatrixp	323
scalarp	446
scaled_bessel_i	207
scaled_bessel_io	207
scaled_bessel_i1	207
scanmap	538
scatterplot	580
schur2comp	433
sconcat	134
scopy	875
scsimp	105
scurvature	372
sdowncase	876
sec	200
sech	200
second	473
sequal	876
sequalignore	876
set_draw_defaults	664
set_edge_weight	734
set_partitions	495
set_plot_option	127
set_random_state	54
set_tex_environment	155
set_tex_environment_default	155
set_up_dot_simplifications	329
set_vertex_label	735
setcheck	546
setcheckbreak	546
setdifference	494
setelmx	324
setequalp	494
setify	495
setp	495
setunits	882
setup_autoload	446
setval	546
seventh	473
sexplode	876
sf	395
shortest_path	735
shortest_weighted_path	735
show	150
showcomps	342
showratvars	150
showtime	29
sign	57
signum	57
similaritytransform	324
simple_linear_regression	864
simplified_output	893
simplify_products	844
simplify_sum	844
simpplode	876

simpmetderiv	351	spherical_bessel_y	819
simpsum	105	spherical_hankel1	819
simtran	324	spherical_hankel2	820
sin	200	spherical_harmonic	820
sinh	200	splice	507
sinnpiflag	294	split	877
sinsert	876	sposition	877
sinvertcase	876	sprint	873
sixth	473	sqrt	58
skewness	574	sqrtdenest	840
skewness_bernoulli	623	sqrtdispflag	58
skewness_beta	610	sremove	877
skewness_binomial	620	sremovefirst	878
skewness_chi2	601	sreverse	878
skewness_continuous_uniform	611	ssearch	878
skewness_discrete_uniform	626	ssort	878
skewness_exp	606	sstatus	29
skewness_f	604	ssubst	878
skewness_gamma	609	ssubstfirst	879
skewness_geometric	625	staircase	689
skewness_gumbel	619	stardisp	150
skewness_hypergeometric	627	stats_numer	851
skewness_laplace	618	status	439
skewness_logistic	612	std	569
skewness_lognormal	608	std_bernoulli	623
skewness_negative_binomial	628	std_beta	610
skewness_noncentral_chi2	603	std_binomial	620
skewness_noncentral_student_t	599	std_chi2	601
skewness_normal	596	std_continuous_uniform	611
skewness_pareto	613	std_discrete_uniform	625
skewness_poisson	622	std_exp	606
skewness_rayleigh	616	std_f	604
skewness_student_t	597	std_gamma	609
skewness_weibull	614	std_geometric	624
slength	877	std_gumbel	619
smake	877	std_hypergeometric	627
smismatch	877	std_laplace	618
solve	278	std_logistic	612
solve_rec	844	std_lognormal	608
solve_rec_rat	846	std_negative_binomial	628
solvedecomposes	281	std_noncentral_chi2	603
solveexplicit	281	std_noncentral_student_t	599
solvefactors	281	std_normal	596
solvenullwarn	282	std_pareto	613
solveradcan	282	std_poisson	621
solvetrigwarn	282	std_rayleigh	616
some	496	std_student_t	597
somrac	434	std_weibull	614
sort	57	std1	569
space	874	stirling	867
sparse	324	stirling1	497
sparse6_decode	740	stirling2	498
sparse6_encode	740	strim	879
sparse6_export	740	striml	879
sparse6_import	740	strimr	879
specint	222	string	150
spherical	674	stringdisp	150
spherical_bessel_j	819	stringout	150, 151

stringp.....	875	test_proportions_difference.....	859
strong_components.....	735	test_rank_sum.....	862
sublis.....	58	test_sign.....	860
sublis_apply_lambda.....	58	test_signed_rank.....	861
sublist.....	58	test_variance.....	855
sublist_indices.....	473	test_variance_ratio.....	856
submatrix.....	324	testsuite_files.....	5
subsample.....	565	tex.....	151
subset.....	499	tex1.....	152
subsetp.....	499	texput.....	152
subst.....	59	third.....	473
subst_inpart.....	59	throw.....	538
substpart.....	60	time.....	439
substring.....	879	timedate.....	439
subvar.....	304	timer.....	547
subvarp.....	61	timer_devalue.....	547
sum.....	94	timer_info.....	547
sumcontract.....	106	title.....	636
sumexpand.....	106	tldefint.....	254
summand_to_rec.....	846	tlimit.....	231
sumsplitfact.....	106	tlimswitch.....	232
supcase.....	879	to_lisp.....	29
supcontext.....	166	todd_coxeter.....	435
surface_hide.....	662	toeplitz.....	785
symbolp.....	61	tokens.....	880
symmdifference.....	500	topological_sort.....	736
symmetric.....	106	totaldisrep.....	187
symmetricip.....	383	totalfourier.....	295
system.....	156	totient.....	418
		tpartpol.....	434
		tr.....	390
		tr_array_as_ref.....	526
tab.....	874	tr_bound_function_applyp.....	526
take_channel.....	682	tr_file_tty_messagesp.....	527
take_inference.....	850	tr_float_can_branch_complex.....	527
tan.....	200	tr_function_call_default.....	527
tanh.....	200	tr_numer.....	527
taylor.....	403	tr_optimize_max_loop.....	527
taylor_logexpand.....	407	tr_semicompile.....	527
taylor_order_coefficients.....	407	tr_state_vars.....	528
taylor_simplifier.....	407	tr_warn_bad_function_calls.....	528
taylor_truncate_polynomials.....	407	tr_warn_fexpr.....	528
taylordepth.....	406	tr_warn_meval.....	528
taylorinfo.....	406	tr_warn_mode.....	528
taylorp.....	406	tr_warn_undeclared.....	528
taytorat.....	407	tr_warn_undefined_variable.....	528
tcontract.....	434	tr_warnings_get.....	528
tellrat.....	187	tr_windy.....	528
tellsimp.....	460	trace.....	548
tellsimpafter.....	461	trace_options.....	548
tensorkill.....	390	tracematrix.....	835
tentex.....	364	transcompile.....	524
tenth.....	473	translate.....	525
terminal.....	633	translate_file.....	525
test_mean.....	851	transparent.....	655
test_means_difference.....	853	transpose.....	324
test_normality.....	863	transrun.....	526
test_proportion.....	857	tree_reduce.....	500

trellis	434	var_beta	610
treinat	434	var_binomial	620
triangularize	325	var_chi2	601
trigexpand	200	var_continuous_uniform	611
trigexpandplus	201	var_discrete_uniform	625
trigexpandtimes	201	var_exp	606
triginverses	201	var_f	604
trigrat	202	var_gamma	609
trigreduce	201	var_geometric	624
trigsign	202	var_gumbel	619
trigsimp	202	var_hypergeometric	626
trivial_solutions	894	var_laplace	618
true	190	var_logistic	612
trunc	407	var_lognormal	608
ttyoff	156	var_negative_binomial	628
tutte_graph	722	var_noncentral_chi2	603
var_noncentral_student_t		var_normal	596
var_pareto		var_poisson	621
var_rayleigh		var_student_t	597
var_weibull		var_weibull	614
var1		vector	671
vector		vectorpotential	61
vectorsimp		verbify	97
verbose		verbose	408
vers		vertex_coloring	739
vertex_connectivity		vertex_degree	736
vertex_distance		vertex_eccentricity	736
vertex_in_degree		vertex_out_degree	737
vertices		vertices_to_cycle	744
vertices_to_path		vertices_to_path	744
W		warnings	893
weyl		wheel_graph	722
while		while	538
with_stdout		with_stdout	156
write_binary_data		write_data	805
write_file		writefile	803
wronskian		wronskian	157
V		values	29
values	29	vandermonde_matrix	785
vandermonde_matrix	785	var	568
var	568	var_bernoulli	623

X

x_voxel.....	664
xaxis.....	640
xaxis_color.....	642
xaxis_secondary.....	641
xaxis_type.....	641
xaxis_width.....	641
xgraph_curves.....	114
xlabel.....	636
xrange.....	630
xrange_secondary.....	630
xreduce.....	501
xthru.....	61
xtics.....	637
xtics_axis.....	639
xtics_rotate.....	639
xtics_rotate_secondary.....	639
xtics_secondary.....	638
xtics_secondary_axis.....	640
xu_grid.....	661
xy_file.....	646
xyplane.....	645

ylabel.....	636
yrange.....	630
yrange_secondary.....	630
ytics.....	638
ytics_axis.....	640
ytics_rotate.....	639
ytics_rotate_secondary.....	639
ytics_secondary.....	638
ytics_secondary_axis.....	640
yv_grid.....	661

Z

z_voxel.....	664
zaxis.....	643
zaxis_color.....	644
zaxis_type.....	644
zaxis_width.....	644
Zeilberger.....	893
zerobern.....	418
zeroequiv.....	62
zerofor.....	785
zeromatrix.....	326
zeromatrixxp.....	785
zeta.....	418
zeta%pi.....	418
zlabel.....	637
zlange.....	764
zrange.....	631
ztics.....	638
ztics_axis.....	640
ztics_rotate.....	639

Y

y_voxel.....	664
yaxis.....	642
yaxis_color.....	643
yaxis_secondary.....	642
yaxis_type.....	643
yaxis_width.....	642