Prácticas de ordenador

con wxMaxima

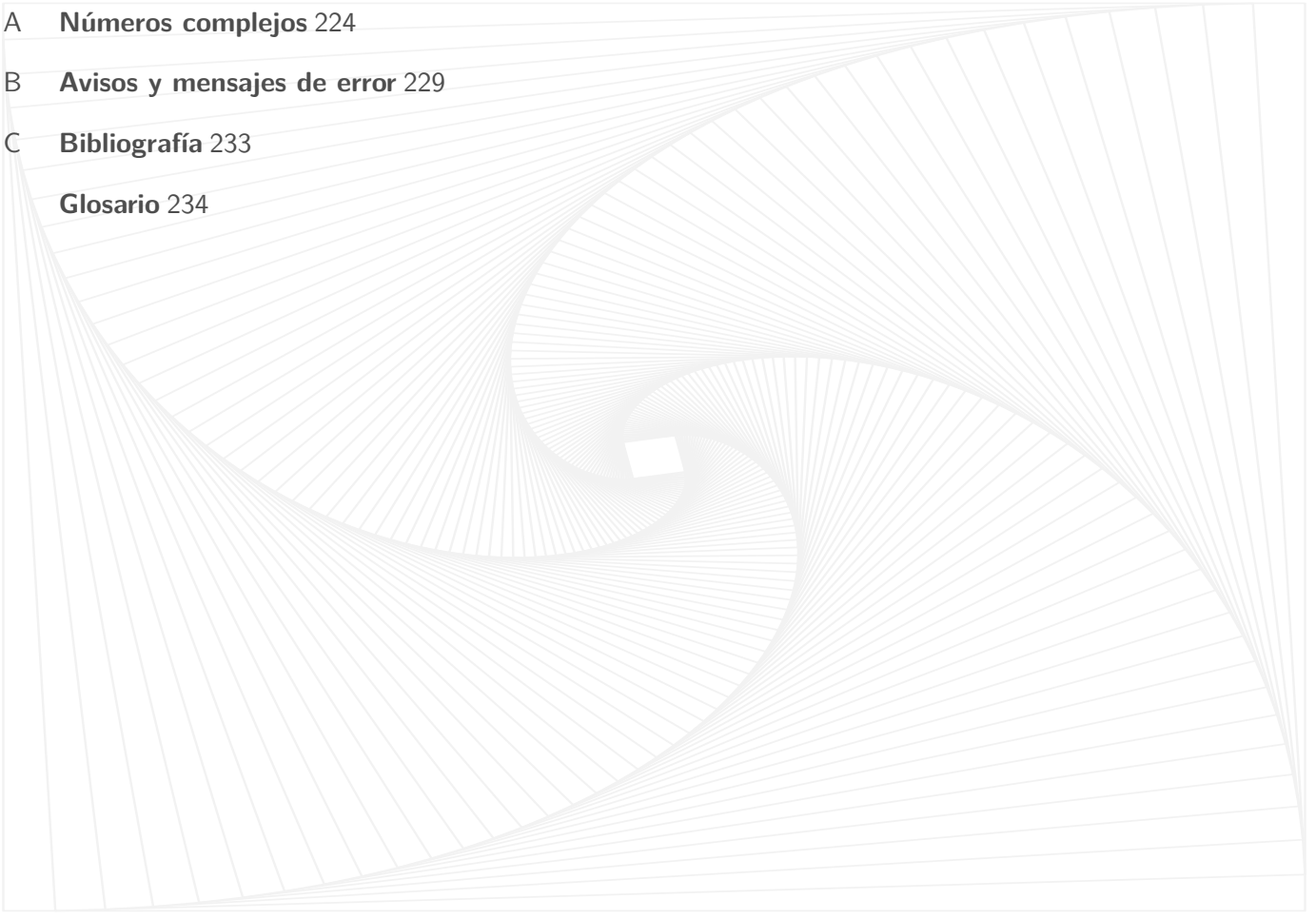
con wxMaxima

ugr Universidad de Granada

Jerónimo Alaminos Prats
Camilo Aparicio del Prado
José Extremera Lizana
Pilar Muñoz Rivas
Armando R. Villena Muñoz

16 septiembre 2011

Índice



Introducción

Maxima es un programa que realiza cálculos matemáticos de forma tanto numérica como simbólica, esto es, sabe tanto manipular números como calcular la derivada de una función. Sus capacidades cubren sobradamente las necesidades de un alumno de un curso de Cálculo en unos estudios de Ingeniería. Se encuentra disponible bajo licencia GNU GPL tanto el programa como los manuales del programa.

Lo que presentamos aquí son unas notas sobre el uso de *Maxima* para impartir la parte correspondiente a unas prácticas de ordenador en una asignatura de Cálculo que incluya derivadas e integrales en una y varias variables y una breve introducción a ecuaciones diferenciales ordinarias. Además de eso, hemos añadido unos capítulos iniciales donde se explican con algo de detalle algunos conceptos más o menos generales que se utilizan en la resolución de problemas con *Maxima.* Hemos pensado que es mejor introducir, por ejemplo, la gestión de gráficos en un capítulo separado que ir comentando cada orden en el momento que se use por primera vez. Esto no quiere decir que todo lo que se cuenta en los cuatro primeros capítulos sea necesario para el desarrollo del resto de estas notas. De hecho, posiblemente es demasiado. En cualquier caso pensamos que puede ser útil en algún momento.

Por qué

Hay muchos programas que cumplen en mayor o menor medida los requisitos que se necesitan para enseñar y aprender Cálculo. Sólo por mencionar algunos, y sin ningún orden particular, casi todos conocemos *Mathematica* (©Wolfram Research) o *Maple* (©Maplesoft). También hay una larga lista de programas englobados en el mundo del software libre que se pueden adaptar a este trabajo.

Siempre hay que intentar escoger la herramienta que mejor se adapte al problema que se presenta y, en nuestro caso, *Maxima* cumple con creces las necesidades de un curso de Cálculo. Es evidente que *Mathematica* o *Maple*

también pero creemos que el uso de programas de software libre permite al alumno y al profesor estudiar cómo está hecho, ayudar en su mejora y, si fuera necesario y posible, adaptarlo a sus propias necesidades.

Además pensamos que el programa tiene la suficiente capacidad como para que el alumno le pueda seguir sacando provecho durante largo tiempo. Estamos todos de acuerdo en que esto sólo es un primer paso y que *Maxima* se puede utilizar para problemas más complejos que los que aparecen en estas notas. Esto no es un callejón sin salida sino el comienzo de un camino.

Dónde y cómo

No es nuestra intención hacer una historia de *Maxima*, ni explicar cómo se puede conseguir o instalar, tampoco aquí encontrarás ayuda ni preguntas frecuentes ni nada parecido. Cualquiera de estas informaciones se encuentra respondida de manera detallada en la página web del programa:

http://maxima.sourceforge.net/es/

En esta página puedes descargarte el programa y encontrar abundante documentación sobre cómo instalarlo. Al momento de escribir estas notas, en dicha página puedes encontrar versiones listas para funcionar disponibles para entornos Windows y Linux e instrucciones detalladas para ponerlo en marcha en Mac OS X.

wxMaxima

En estas notas no estamos usando *Maxima* directamente sino un entorno gráfico que utiliza *Maxima* como motor para realizar los cálculos. Este entorno (programa) es *wxMaxima*. Nos va a permitir que la curva de aprendizaje sea mucho más suave. Desde el primer día el alumno será capaz de realizar la mayoría de las operaciones básicas.

A pesar de ello, en todos los ejemplos seguimos utilizando la notación de *Maxima* que es la que le aparece al lector en pantalla y que nunca está de más conocer. *wxMaxima* se puede descargar de su página web

http://wxmaxima.sourceforge.net/

Suele venir incluido con *Maxima* en su versión para entorno Windows y en Mac OS X. Sobre la instalación en alguna distribución Linux es mejor consultar la ayuda sobre su correspondiente programa para gestionar software.

Granada a 10 de septiembre de 2008

1 Primeros pasos

1.1 Introducción

Vamos a comenzar familiarizándonos con *Maxima* y con el entorno de trabajo *wxMaxima*. Cuando iniciamos el programa se nos presenta una ventana como la de la Figura 1.1. En la parte superior tienes el menú con las opciones usuales (abrir, cerrar, guardar) y otras relacionadas con las posibilidades más "matemáticas" de *Maxima*. En segundo lugar aparecen algunos iconos que sirven de atajo a algunas operaciones y la ventana de trabajo. En ésta última, podemos leer un recordatorio de las versiones que estamos utilizando de los programas *Maxima* y *wxMaxima* así como el entorno Lisp sobre el que está funcionando y la licencia (GNU Public License):[1]

```
wxMaxima 0.8.3a http://wxmaxima.sourceforge.net
Maxima 5.19.2 http://maxima.sourceforge.net
Using Lisp SBCL 1.0.30
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.
```

[1] Por defecto, la ventana de *wxMaxima* aparece en blanco. En las preferencias del programa, se puede elegir que aparezca la versión instalada al inicio del mismo como se ve en la figura.

Ya iremos comentando con mayor profundidad los distintos menús y opciones que se nos presentan pero antes de ir más lejos, ¿podemos escribir algo? Sí, situa el cursor dentro de la ventana, pulsa y escribe 2+3. Luego pulsa las teclas Shift + Return. Obtendrás algo similar a esto:

```
(%i1)   2+3;
(%o1)   5
```

Como puedes ver *Maxima* da la respuesta correcta: 5. Bueno, no parece mucho. Seguro que tienes una calculadora que hace eso. De acuerdo. Es sólo el principio.

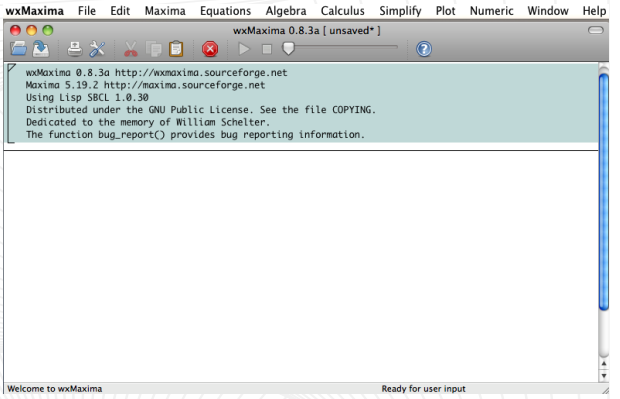


Figura 1.1 Ventana inicial de *wxMaxima*

Observación 1.1. Conviene hacer algunos comentarios sobre lo que acabamos de hacer:

a) No intentes escribir los símbolos "(%i1)" y "(%o1)", ya que éstos los escribe el programa para llevar un control sobre las operaciones que va efectuando. "(%i1)" se refiere a la primera entrada (input) y "(%o1)" a la primera respuesta (output).

b) La entradas terminan en punto y coma. *wxMaxima* lo añade si tú te has olvidado de escribirlo. Justamente lo que nos había pasado.

Operaciones básicas

+	suma
*	producto
/	división
^ o **	potencia
sqrt( )	raíz cuadrada

El producto se indica con "*":

```
(%i2)   3*5;
(%o2)   15
```

Para multiplicar números es necesario escribir el símbolo de la multiplicación. Si sólo dejamos un espacio entre los factores el resultado es un error:

```
(%i3)   5 4;
        Incorrect syntax: 4 is not an infix operator
(%o3)   5Space4;
          ^
```

También podemos dividir

```
(%i4)   5+(2*4+6)/7;
(%o4)   7
(%i5)   5+2*4+6/7;
(%o5)   97/7
```

eso sí, teniendo cuidado con la precedencia de las operaciones. En estos casos el uso de paréntesis es obligado.

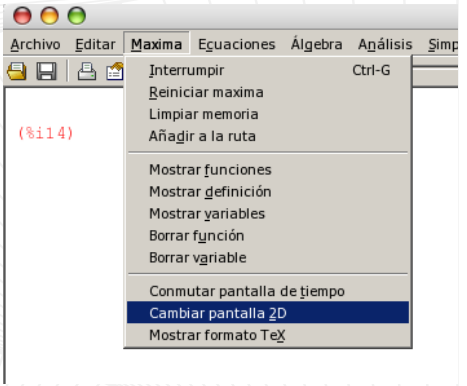
Podemos escribir potencias

```
(%i6)   3^57;
(%o6)   157004289908208161164053456 3
```

Fíjate en el número de dígitos que tiene el resultado. Es un primer ejemplo de que la potencia de cálculo de *Maxima* es mayor que la de una calculadora que no suele tener más allá de 10 o 12. Ya sé lo que estarás pensando en este momento: en lugar de elevar a 57, elevemos a un número más grande. De acuerdo.

```
(%i7)   3^1000;

        132207081948080663689045525975
(%o7)   5[418 digits]6143661321731027
        68902855220001
```



Como puedes ver, *Maxima* realiza la operación pero no muestra el resultado completo. Nos dice que, en este caso, hay 418 dígitos que no está mostrando. ¿Se puede saber cuáles son? Sí. Nos vamos al menú **Maxima→Cambiar pantalla 2D** y escogemos **ascii**. Por último, repetimos la operación.

```
(%i8)   set_display('ascii)$
(%i9)   3^1000;

        132207081948080663689045525975214436596542203275214816766
        492036822682859734670489954077831385060806196390977769687258
        235559509545821006189118653427252579536740276202251983208038
        78780147742289648412743904001175886180411289478156230944380
(%o9)   06156617305408667449050617812548034440554705439703889581746
        65368254916136220830268563778582290228416398307887896918556
        64040848989376093732421718463599386955167650189405881090604
        26089671438864102814350385648747165832010614366132173102768
        902855220001
```

La salida en formato ascii es la que tiene por defecto *Maxima*. La salida con formato xml es una mejora de *wxMaxima*. Siempre puedes cambiar entre una y otra vía el menú o volviendo a escribir

```
(%i10)  set_display('xml)$
(%i11)  3^1000;
(%o11)  1322070819480806636890455259752[418 digits]61436613217310
        2768902855220001
```

Observación 1.2. Antes de seguir, ¿por qué sale $ y no punto y coma al final de la salida anterior? El punto y coma sirve para terminar un comando o separar varios de ellos. El dólar, $, también termina un comando o separa varios de ellos pero, a diferencia del punto y coma, *no* muestra el resultado en pantalla.

Si trabajamos con fracciones, *Maxima* dará por defecto el resultado en forma de fracción

```
(%i12)  2+5/11;
(%o12)  27/11
```

simplificando cuando sea posible

```
(%i13)  128/234;
(%o13)  64/117
```

Cálculo simbólico

Cuando hablamos de que *Maxima* es un programa de cálculo simbólico, nos referimos a que no necesitamos trabajar con valores concretos. Fíjate en el siguiente ejemplo:

```
(%i14)  a/2+3*a/5
(%o14)  11a/10
```

Bueno, hasta ahora sabemos sumar, restar, multiplicar, dividir y poco más. *Maxima* tiene predefinidas la mayoría de las funciones usuales. Por ejemplo, para obtener la raíz de un número se usa el comando `sqrt`

```
(%i15)  sqrt(5);
(%o15)  sqrt(5)
```

lo cuál no parece muy buena respuesta. En realidad es la mejor posible: *Maxima* es un programa de cálculo simbólico y siempre intentará dar el resultado en la forma más exacta.

Obviamente, también puedes hacer la raíz cuadrada de un número, elevando dicho número al exponente $\frac{1}{2}$

```
(%i16)  5^(1/2);
(%o16)  sqrt(5)
```

Si queremos obtener la expresión decimal, utilizamos la orden `float`.

```
(%i17)   float(√5);
(%o17)   2.23606797749979
```

Constantes

Además de las funciones usuales (ya iremos viendo más), *Maxima* también conoce el valor de algunas de las constantes típicas.

%pi	el número π
%e	el número e
%i	la unidad imaginaria
%phi	la razón aúrea, $\frac{1+\sqrt{5}}{2}$

Podemos operar con ellas como con cualquier otro número.

```
(%i18)   (2+3*%i)*(5+3*%i);
(%o18)   (3*%i+2)*(3*%i+5)
```

Evidentemente necesitamos alguna manera de indicar a *Maxima* que debe desarrollar los productos, pero eso lo dejaremos para más tarde.

¿Cuál era el resultado anterior?

%	último resultado
%i*número*	entrada *número*
%o*número*	resultado *número*

Con *Maxima* podemos usar el resultado de una operación anterior sin necesidad de teclearlo. Esto se consigue con la orden %. No sólo podemos referirnos a la última respuesta sino a cualquier entrada o salida anterior. Para ello

```
(%i19)  %o15
(%o19)  √5
```

además podemos usar esa información como cualquier otro dato.

```
(%i20)  %o4+%o5;
(%o20)  146/7
```

1.2 Resultados exactos y aproximación decimal

Hay una diferencia básica entre el concepto abstracto de número real y cómo trabajamos con ellos mediante un ordenador: la memoria y la capacidad de proceso de un ordenador son finitos. La precisión de un ordenador es el número de dígitos con los que hace los cálculos. En un hipotético ordenador que únicamente tuviera capacidad para almacenar el primer decimal, el número π sería representado como 3.1. Esto puede dar lugar a errores si, por ejemplo, restamos números similares. *Maxima* realiza los cálculos de forma simbólica o numérica. En principio, la primera forma es mejor, pero hay ocasiones en las que no es posible.

Maxima tiene dos tipos de "números": exactos y aproximados. La diferencia entre ambos es la esperable. $\frac{1}{3}$ es un número exacto y 0.333 es una aproximación del anterior. En una calculadora normal todos los números son aproximados y la precisión (el número de dígitos con el que trabaja la calculadora) es limitada, usualmente 10 o 12 dígitos. *Maxima* puede manejar los números de forma exacta, por ejemplo

```
(%i21)   1/2+1/3;
(%o21)   5/6
```

Mientras estemos utilizando únicamente números exactos, *Maxima* intenta dar la respuesta de la misma forma. Ahora bien, en cuanto algún término sea aproximado el resultado final será siempre aproximado. Por ejemplo

```
(%i22)   1.0/2+1/3;
(%o22)   0.83333333333333
```

Este comportamiento de *Maxima* viene determinado por la variable `numer` que tiene el valor `false` por defecto. En caso de que cambiemos su valor a `true`, la respuesta de *Maxima* será aproximada.

```
(%i23)   numer;
(%o23)   false
(%i24)   numer:true$
(%i25)   1/2+1/3
(%o25)   0.83333333333333
(%i26)   numer:false$
```

Recuerda cambiar el valor de la variable `numer` a `false` para volver al comportamiento original de *Maxima*. En *wxMaxima*, podemos utilizar el menú **Numérico→conmutar salida numérica** para cambiar el valor de la variable `numer`.

`float(`*número*`)`	expresión decimal de *número*
número`,numer`	expresión decimal de *número*
`bfloat(`*número*`)`	expresión decimal larga de *número*

Si sólo queremos conocer una aproximación decimal de un resultado exacto, tenemos a nuestra disposición las órdenes `float` y `bfloat`.

```
(%i27)   float(sqrt(2));
(%o27)   1.414213562373095
```

En la ayuda de *Maxima* podemos leer

```
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.
```

Maxima puede trabajar con cualquier precisión. Dicha precisión la podemos fijar asignando el valor que queramos a la variable `fpprec`. Por ejemplo, podemos calcular cuánto valen los 100 primeros decimales de π:

```
(%i28)  fpprec:100;
(%o28)  100
(%i29)  float(%pi);
(%o29)  3.141592653589793
```

No parece que tengamos 100 dígitos...de acuerdo, justo eso nos decía la ayuda de máxima: “La variable `fpprec` no afecta a los cálculos con números decimales de punto flotante ordinarios”. Necesitamos la orden `bfloat` para que *Maxima* nos muestre todos los decimales pedidos (y cambiar la pantalla a `ascii`):

```
(%i30)  bfloat(%pi);
(%o30)  3.14159265358979323846264338327[43 digits]628620899862803482
        53421170680b0
(%i31)  set_display('ascii)$
(%i32)  bfloat(%pi);
```

```
(%o32)  3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706 8b0
```

Si te has fijado, en la salida anterior la expresión decimal del número π termina con "b0". Los números en coma flotante grandes siempre terminan con "b" seguido de un número n para indicar que debemos multiplicar por 10^n. En el caso anterior, la expresión decimal de π deberíamos multiplicarla por $10^0 = 1$.

Por último, observa que, como se puede ver en la Figura 1.2, también se puede utilizar el menú **Numérico→A real** o **Numérico→A real grande(bigfloat)** para obtener la expresión decimal buscada.

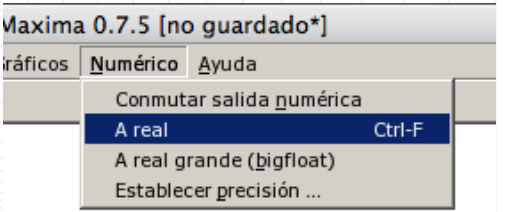


Figura 1.2 En el menú, la pestaña **Numérico** permite obtener la expresión decimal con la precisión que se desee

1.3 Funciones usuales

Además de las operaciones elementales que hemos visto, *Maxima* tiene definidas la mayor parte de las funciones elementales. Los nombres de estas funciones suelen ser su abreviatura en inglés, que algunas veces difiere bastante de su nombre en castellano. Por ejemplo, ya hemos visto raíces cuadradas

```
(%i33)   sqrt(4);
(%o33)   2
```

`sqrt(x)`	raíz cuadrada de x
`exp(x)`	exponencial de x
`log(x)`	logaritmo neperiano de x
`sin(x), cos(x), tan(x)`	seno, coseno y tangente *en radianes*
`csc(x), sec(x), cot(x)`	cosecante, secante y cotangente *en radianes*
`asin(x), acos(x), atan(x)`	arcoseno, arcocoseno y arcotangente
`sinh(x), cosh(x), tanh(x)`	seno, coseno y tangente hiperbólicos
`asinh(x), acosh(x), atanh(x)`	arcoseno, arcocoseno y arcotangente hiperbólicos

Potencias, raíces y exponenciales

Hemos visto que podemos escribir potencias utlizando `^` o `**`. No importa que el exponente sea racional. En otras palabras: podemos calcular raíces de la misma forma

```
(%i34)  625^(1/4);
(%o34)  5
(%i35)  625^(1/3)*2^(1/3):
(%o35)  2^(1/3) 5^(4/3)
```

En el caso particular de que la base sea el número e, podemos escribir

```
(%i36)  %e^2;
(%o36)  %e^2
```

o, lo que es más cómodo especialmente si el exponente es una expresión más larga, utilizar la función exponencial `exp`

```
(%i37)  exp(2);
(%o37)  %e^2
(%i38)  exp(2),numer;
(%o38)  7.38905609893065
```

Logaritmos

Maxima sólo tiene la definición del logaritmo neperiano o natural que se consigue con la orden `log`:

```
(%i39)   log(20);
(%o39)   log(20)
```

y si lo que nos interesa es su expresión decimal

```
(%i40)   log(20),numer;
(%o40)   2.995732273553991
```

Observación 1.3. Mucho cuidado con utilizar `ln` para calcular logaritmos neperianos:

```
(%i41)   ln(20);
(%o41)   ln(20)
```

puede parecer que funciona igual que antes pero en realidad *Maxima* no tiene la más remota idea de lo que vale, sólo está repitiendo lo que le habéis escrito. Si no te lo crees, pídele que te diga el valor:

```
(%i42)   ln(20),numer;
(%o42)   ln(20)
```

¿Cómo podemos calcular $\log_2(64)$? Para calcular logaritmos en cualquier base podemos utilizar que

$$\log_b(x) = \frac{\log(x)}{\log(b)}\,.$$

Se puede definir una función que calcule los logaritmos en base 2 de la siguiente manera

```
(%i43)  log2(x):=log(x)/log(2)$
(%i44)  log2(64);
(%o44)  log(64)/log(2)
```

Te habrás dado cuenta de que *Maxima* no desarrolla ni simplifica la mayoría de las expresiones. En segundo lugar, la posibilidad de definir funciones a partir de funciones conocidas nos abre una amplia gama de posibilidades. En el segundo capítulo veremos con más detalle cómo trabajar con funciones.

Funciones trigonométricas e hiperbólicas

Maxima tiene predefinidas las funciones trigonométricas usuales seno, `sin`, coseno, `cos`, y tangente, `tan`, que devuelven, si es posible, el resultado exacto.

```
(%i45)  sin(%pi/4)
(%o45)  1/sqrt(2)
```

Por defecto, las funciones trigonométricas están expresadas en radianes.

También están predefinidas sus inversas, esto es, arcoseno, arcocoseno y arcotangente, que se escriben respectivamente `asin(x)`, `acos(x)` y `atan(x)`, así como las funciones recíprocas secante, `sec(x)`, cosecante, `csc(x)`, y cotangente, `cot(x)`[2].

```
(%i46)  atan(1);
(%o46)  π/4
(%i47)  sec(0);
(%o47)  1
```

De forma análoga, puedes utilizar las correspondientes funciones hiperbólicas.

Otras funciones

Además de las anteriores, hay muchas más funciones de las que *Maxima* conoce la definición. Podemos, por ejemplo, calcular factoriales

```
(%i48)  32!
(%o48)  263130836933693530167218012160000000
```

o números binómicos

```
(%i49)  binomial(10,4);
(%o49)  210
```

[2] El número π no aparece como tal por defecto en *wxMaxima*. Para que aparezca así, puedes marcar **Usar fuente griega** dentro de **Preferencias**→**Estilo**.

¿Recuerdas cuál es la definición de $\binom{m}{n}$?

$$\binom{m}{n} = \frac{m(m-1)(m-2)\cdots(m-(n-1))}{n!}$$

En el desarrollo de Taylor de una función veremos que estos números nos simplifican bastante la notación.

`n!`	factorial de n
`entier(x)`	parte entera de x
`abs(x)`	valor absoluto o módulo de x
`random(x)`	devuelve un número aleatorio
`signum(x)`	signo de x
`max(`x_1`,`x_2`,...)`	máximo de x_1,x_2,...
`min(`x_1`,`x_2`,...)`	mínimo de x_1,x_2,...

Una de las funciones que usaremos más adelante es `random`. Conviene comentar que su comportamiento es distinto dependiendo de si se aplica a un número entero o a un número decimal, siempre positivo, eso sí. Si el número x es natural, `random(x)` devuelve un natural menor o igual que $x-1$.

```
(%i50)   random(100);
(%o50)   7
```

Obviamente no creo que tú también obtengas un 7, aunque hay un caso en que sí puedes saber cuál es el número "aleatorio" que vas a obtener:

```
(%i51)   random(1);
(%o51)   0
```

efectivamente, el único entero no negativo menor o igual que $1-1$ es el cero. En el caso de que utilicemos números decimales `random(x)` nos devuelve un número decimal menor que x. Por ejemplo,

```
(%i52)   random(1.0);
(%o52)   0.9138095996129
```

nos da un número (decimal) entre 0 y 1.

La lista de funciones es mucho mayor de lo que aquí hemos comentado y es fácil que cualquier función que necesites esté predefinida en *Maxima*. En la ayuda del programa puedes encontrar la lista completa.

1.4 Operadores lógicos y relacionales

Maxima puede comprobar si se da una igualdad (o desigualdad). Sólo tenemos que escribirla y nos dirá qué le parece:

```
(%i53)   is(3<5);
(%o53)   true
```

is(*expresión*)	decide si la expresión es cierta o falsa
assume(*expresión*)	supone que la expresión es cierta
forget(*expresión*)	olvida la expresión
and	y
or	o

No se pueden encadenar varias condiciones. No se admiten expresiones del tipo $3 < 4 < 5$. Las desigualdades sólo se aplican a parejas de expresiones. Lo que sí podemos hacer es combinar varias cuestiones como, por ejemplo,

```
(%i54)   is(3<2 or 3<4);
(%o54)   true
```

En cualquier caso tampoco esperes de *Maxima* la respuesta al sentido de la vida:

```
(%i55)   is((x+1)^2=x^2+2*x+1);
```

```
(%o55)   false
```

=	igual
notequal	distinto
x>y	mayor
x<y	menor
x>=y	mayor o igual
x<=y	menor o igual

Pues no parecía tan difícil de responder. Lo cierto es que *Maxima* no ha desarrollado la expresión. Vamos con otra pregunta fácil:

```
(%i56)   is((x+1)^2>0);
(%o56)   unknown
```

Pero, ¿no era positivo un número al cuadrado? Hay que tener en cuenta que x podría valer -1 . ¿Te parece tan mala la respuesta ahora? Si nosotros disponemos de información adicional, siempre podemos "ayudar". Por ejemplo, si sabemos que x es distinto de -1 la situación cambia:

```
(%i57)   assume(notequal(x,-1));
(%o57)   [notequal(x,-1)]
(%i58)   is((x+1)^2>0);
```

```
(%o58)    true
```

Eso sí, en este caso *Maxima* presupone que x es distino de -1 en lo que resta de sesión. Esto puede dar lugar a errores si volvemos a utilizar la variable x en un ambiente distinto más adelante. El comando `forget` nos permite "hacer olvidar" a *Maxima*.

```
(%i59)    forget(notequal(x,-1));
(%o59)    [notequal(x,-1)]
(%i60)    is(notequal(x,-1));
(%o60)    unknown
```

1.5 Variables

El uso de variables es muy fácil y cómodo en *Maxima*. Uno de los motivos de esto es que no hay que declarar tipos previamente. Para asignar un valor a una variable utilizamos los dos puntos

```
(%i61)   a:2
(%o61)   2
(%i62)   a^2
(%o62)   4
```

var`:`*expr*	la variable *var* vale *expr*
`kill(`*a1*`,`*a2*`,...)`	elimina los valores
`remvalue(`*var1*`,`*var2*`,...)`	borra los valores de las variables
`values`	muestra las variables con valor asignado

Cuando una variable tenga asignado un valor concreto, a veces diremos que es una constante, para distinguir del caso en que no tiene ningún valor asignado.

Observación 1.4. El nombre de una variable puede ser cualquier cosa que no empiece por un número. Puede ser una palabra, una letra o una mezcla de ambas cosas.

```
(%i63)   largo:10;
(%o63)   10
```

```
(%i64)   ancho:7;
(%o64)   7
(%i65)   largo*ancho;
(%o65)   70
```

Podemos asociar una variable con prácticamente cualquier cosa que se nos ocurra: un valor numérico, una cadena de texto, las soluciones de una ecuación, etc.

```
(%i66)   solucion:solve(x^2-1=0,x);
(%o66)   [x=-1,x=1]
```

para luego poder usarlas.

Los valores que asignamos a una variable no se borran por sí solos. Siguen en activo mientras no los cambiemos o comencemos una nueva sesión de *Maxima*. Quizá por costumbre, todos tendemos a usar como nombre de variables x, y, z, t, igual que los primeros nombres que se nos vienen a la cabeza de funciones son f o g. Después de trabajar un rato con *Maxima* es fácil que usemos una variable que ya hemos definido antes. Es posible que dar un valor a una variable haga que una operación posterior nos de un resultado inesperado o un error. Por ejemplo, damos un valor a x

```
(%i67)   x:3;
(%o67)   3
```

y después intentamos derivar una función de x, olvidando que le hemos asignado un valor. ¿Cuál es el resultado?

```
(%i68)    diff(sin(x),x);
          Non-variable 2nd argument to diff:
          3
          - an error. To debug this try debugmode(true);
```

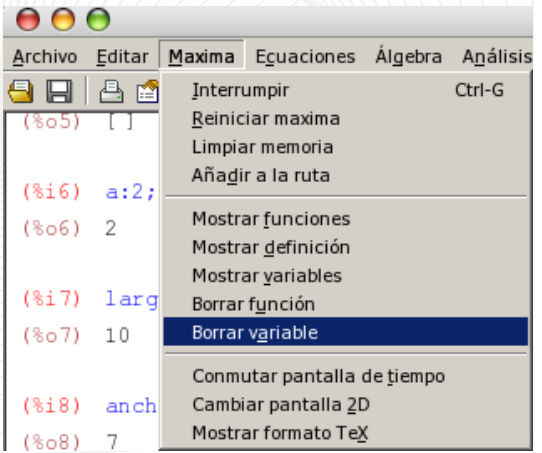
Efectivamente, un error. Hay dos maneras de evitar esto. La primera es utilizar el operador comilla, ', que evita que se evalúe la variable:

```
(%i69)    diff(sin('x),'x);
(%o69)    cos(x)
```

La segunda es borrar el valor de x. Esto lo podemos hacer con la orden `kill` o con la orden `remvalue`. También puedes ir al menú **Maxima**→**borrar variable** y escribir las variables que quieres borrar. Por defecto se borrarán todas.

Si te fijas, dentro del menú **Maxima** también hay varios ítems interesantes: se pueden borrar funciones y se pueden mostrar aquellas variables (y funciones) que tengamos definidas. Esto se consigue con la orden `values`.

```
(%i70)    values;
(%o70)    [a,largo,ancho,x,solucion]
```



Una vez que sabemos cuáles son, podemos borrar algunas de ellas

```
(%i71)   remvalue(a,x);
(%o71)   [a,x]
```

o todas.

```
(%i72)   remvalue(all);
(%o72)   [largo,ancho,solucion]
```

La orden `remvalue` sólo permite borrar valores de variables. Existen versiones similares para borrar funciones, reglas, etc. En cambio, la orden `kill` es la versión genérica de borrar valores de cualquier cosa.

```
(%i73)   ancho:10$
(%i74)   kill(ancho);
(%o74)   done
(%i75)   remvalue(ancho);
(%o75)   [false]
```

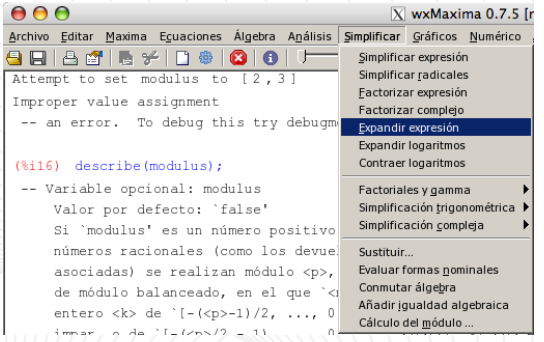
Una de las pequeñas diferencias entre `kill` y `remvalue` es que la primera no comprueba si la variable, o lo que sea, estaba previamente definida y siempre responde *done*. Existe también la posibilidad de borrar *todo*:

```
(%i76)   kill(all);
(%o0)    done
```

y, si te fijas, *Maxima* se reinicia: es como si empezáramos de nuevo. Hemos borrado cualquier valor que tuviésemos previamente definido.

1.6 Expresiones simbólicas

Hasta ahora sólo hemos usado el *Maxima* como una calculadora muy potente, pero prácticamente todo lo que hemos aprendido puede hacerse sin dificultad con una calculadora convencional. Entonces, ¿qué puede hacer *Maxima* que sea imposible con una calculadora? Bueno, entre otras muchas cosas que veremos posteriormente, la principal utilidad de *Maxima* es el cálculo simbólico, es decir, el trabajar con expresiones algebraicas (expresiones donde intervienen variables, constantes... y no tienen por qué tener un valor numérico concreto) en vez de con números. Por ejemplo, el programa sabe que la función logaritmo y la función exponencial son inversas una de otra, con lo que si ponemos




```
(%i1)   exp(log(x));
(%o1)   x
```

es decir, sin saber el valor de la variable x el programa es capaz de trabajar simbólicamente con ella. Más ejemplos

```
(%i2)   exp(x)*exp(y);
(%o2)   %e^(y+x)
```

Aunque parece que no siempre obtenemos el resultado esperado

```

```

```
(%i3)   log(x*y);
(%o3)   log(x y)
(%i4)   log(x)+log(y);
(%o4)   log(y)+log(x)
```

Vamos a practicar con comandos de *Maxima* para manejar expresiones algebraicas: polinomios, funciones racionales, trigonométricas, etc.

Casi todas las órdenes de esta sección, ya sea expandir o simplificar expresiones, se encuentran en el menú **Simplificar** y, opcionalmente, en los paneles de *wxMaxima*.

1.6.1 Desarrollo de expresiones simbólicas

La capacidad de *Maxima* para trabajar con expresiones es notable. Comencemos con funciones sencillas. Consideremos el polinomio

```
(%i5)   p:(x+2)*(x-1);
(%o5)   (x-1)(x+2)
```

lo único qué hace *Maxima* es reescribirlo. ¿Y las potencias?

```
(%i6)   q:(x-3)^2
(%o6)   (x-3)^2
```

Vale, tampoco desarrolla el cuadrado. Probemos ahora a restar las dos expresiones:

```
(%i7)   p-q;
(%o7)   (x-1)(x+2)-(x-3)^2
```

Si no había desarrollado las expresiones anteriores, no era lógico esperar que desarrollara ahora la diferencia. *Maxima* no factoriza ni desarrolla automáticamente: debemos decirle que lo haga. ¿Cómo lo hacemos?

`expand(`*expr*`)`	realiza productos y potencias
`partfrac(`*frac*`,`*var*`)`	descompone en fracciones simples
`num(`*frac*`)`	numerador
`denom(`*frac*`)`	denominador

La orden `expand` desarrollo productos, potencias,

```
(%i8)   expand(p);
(%o8)   x^2+x-2
```

y cocientes.

```
(%i9)   expand(p/q);
(%o9)   x^2/(x^2-6 x+9)+x/(x^2-6 x+9)-2/(x^2-6 x+9)
```

Como puedes ver, expand sólo divide la fracción teniendo en cuenta el numerador. Si queremos dividir en fracciones simples tenemos que usar partfrac.

```
(%i10)  partfrac(p/q,x);
(%o10)  7/(x-3) + 10/(x-3^2)+1
```

Por cierto, también podemos recuperar el numerador y el denominador de una fracción con las órdenes num y denom:

```
(%i11)  denom(p/q);
(%o11)  (x-3)^2
(%i12)  num(p/q);
(%o12)  (x-1)(x+2)
```

Comportamiento de expand

El comportamiento de la orden expand viene determinado por el valor de algunas variables. No vamos a comentar todas, ni mucho menos, pero mencionar algunas de ellas nos puede dar una idea del grado de control al que tenemos acceso.

expand(*expr*,n,m)	desarrolla potencias con grado entre $-m$ y n
logexpand	variable que controla el desarrollo de logaritmos
radexpand	variable que controla el desarrollo de radicales

Si quisiéramos desarrollar la función

$$(x+1)^{100}+(x-3)^{32}+(x+2)^2+x-1-\frac{1}{x}+\frac{2}{(x-1)^2}+\frac{1}{(x-7)^{15}}$$

posiblemente no estemos interesados en que *Maxima* escriba los desarrollos completos de los dos primeros sumandos o del último. Quedaría demasiado largo en pantalla. La orden `expand` permite acotar qué potencias desarrollamos. Por ejemplo, `expand(`*`expr`*`,3,5)` sólo desarrolla aquellas potencias que estén entre 3 y -5.

```
(%i13)  expand((x+1)^100+(x-3)^32+(x+2)^2+x-1-1/x+2/((x-1)^2)
          +1/((x-7)^15),3,4);
```

(%o13) $\frac{2}{x^2-2x+1}+(x+1)^{100}+x^2+5x-\frac{1}{x}+(x-3)^{32}+\frac{1}{(x-7)^{15}}+3$

Las variables `logexpand` y `radexpand` controlan si se simplifican logaritmos de productos o radicales con productos. Por defecto su valor es `true` y esto se traduce en que `expand` no desarrolla estos productos:

```
(%i14)  log(a*b);
(%o14)  log(a b)
(%i15)  sqrt(x*y)
(%o15)  sqrt(x y)
```

Cuando cambiamos su valor a `all`,

```
(%i16)  radexpand:all$ logexpand:all$
```

```
(%i17)  log(a*b);
(%o17)  log(a)+log(b)
(%i18)  sqrt(x*y)
(%o18)  sqrt(x)*sqrt(y)
```

Dependiendo del valor de `logexpand`, la respuesta de *Maxima* varía cuando calculamos $\log(a^b)$ o $\log(a/b)$.

Compara tú cuál es el resultado de $\sqrt{x^2}$ cuando `radexpand` toma los valores `true` y `all`.

Factorización

`factor(`*expr*`)`	escribe la expresión como producto de factores más sencillos

La orden `factor` realiza la operación inversa a `expand`. La podemos utilizar tanto en números

```
(%i19)  factor(100);
(%o19)  2^2 5^2
```

como con expresiones polinómicas como las anteriores

```
(%i20)  factor(x^2-1);
(%o20)  (x-1)(x+1)
```

El número de variables que aparecen tampoco es un problema:

```
(%i21)   (x-y)*(x*y-3*x^2);
(%o21)   (x-y)(xy-3x^2)
(%i22)   expand(%);
(%o22)   -xy^2+4x^2y-3x^3
(%i23)   factor(%);
(%o23)   -x(y-3x)(y-x)
```

Evaluación de valores en expresiones

ev(*expr*,*arg1*,*arg2*,...)	evalua la expresión según los argumentos

Ahora que hemos estado trabajando con expresiones polinómicas, para evaluar en un punto podemos utilizar la orden ev. En su versión más simple, esta orden nos permite dar un valor en una expresión:

```
(%i24)   ev(p,x=7);
(%o24)   54
```

que puede escribirse también de la forma

```
(%i25)   p,x=7;
```

```
(%o25)    54
```

También se puede aplicar ev a una parte de la expresión:

```
(%i26)    x^2+ev(2*x,x=3);
(%o26)    x^2+6
```

Este tipo de sustituciones se pueden hacer de forma un poco más general y sustituir expresiones enteras

```
(%i27)    ev(x+(x+y)^2-3*(x+y)^3,x+y=t);
(%o27)    x-3*t^3+t^2
```

En la ayuda de *Maxima* puedes ver con más detalle todos los argumentos que admite la orden ev, que son muchos.

1.6.2 Simplificación de expresiones

Es discutible qué queremos decir cuando afirmamos que una expresión es más simple o más sencilla que otra. Por ejemplo, ¿cuál de las dos siguientes expresiones te parece más sencilla?

```
(%i28)    radcan(p/q);
(%o28)    (x^2+x-2)/(x^2-6*x+9)
```

```
(%i29)  partfrac(p/q,x);
(%o29)  7/(x-3) + 10/(x-3^2)+1
```

`radcan(`*expr*`)`	simplifica expresiones con radicales
`ratsimp(`*expr*`)`	simplifica expresiones racionales
`fullratsimp(`*expr*`)`	simplifica expresiones racionales

Maxima tiene algunas órdenes que permiten simplificar expresiones pero muchas veces no hay nada como un poco de ayuda y hay que indicarle si queremos desarrollar radicales o no, logaritmos, etc como hemos visto antes.

Para simplificar expresiones racionales, `ratsimp` funciona bastante bien aunque hay veces que es necesario aplicarlo más de una vez. La orden `fullratsimp` simplifica algo mejor a costa de algo más de tiempo y proceso.

```
(%i30)  fullratsimp((x+a)*(x-b)^2*(x^2-a^2)/(x-a));
(%o30)  x^4+(2a-2b)x^3+(b^2-4ab+a^2)x^2+(2ab^2-2a^2b)x+a^2b^2
```

Para simplificar expresiones que contienen radicales, exponenciales o logaritmos es más útil la orden `radcan`

```
(%i31)  radcan((%e^(2*x)-1)/(%e^x+1));
(%o31)  %e^x-1
```

1.6.3 Expresiones trigonométricas

Maxima conoce las identidades trigonométricas y puede usarlas para simplificar expresiones en las que aparezcan dichas funciones. En lugar de `expand` y `factor`, utilizaremos las órdenes `trigexpand`, `trigsimp` y `trigreduce`.

`trigexpand(`*expresion*`)`	desarrolla funciones trigonométricas e hipérbolicas
`trigsimp(`*expresion*`)`	simplifica funciones trigonométricas e hiperbólicas
`trigreduce(`*expresion*`)`	simplifica funciones trigonométricas e hiperbólicas

Por ejemplo,

```
(%i32)  trigexpand(cos(a+b));
(%o32)  cos(a)cos(b)-sin(a)sin(b);
(%i33)  trigexpand(sin(2*atan(x)));
(%o33)  2x/(x^2+1)
(%i34)  trigexpand(sin(x+3*y)+cos(2*z)*sin(x-y));
(%o34)  -(cos(x)sin(y)-sin(x)cos(y))(cos(z)^2-sin(z)^2)+cos(x)sin(3y)+
        sin(x)cos(3y)
(%i35)  trigexpand(8*sin(2*x)^2*cos(x)^3);
```

```
(%o35)    32 cos(x)^5 sin(x)^2
```

Compara el resultado del comando `trigexpand` con el comando `trigreduce` en la última expresión:

```
(%i36)    trigreduce(8*sin(2*x)^2*cos(x)^3);
```

$$(\%o36)\quad 8\left(-\frac{\frac{\cos(7\,x)}{2}+\frac{\cos(x)}{2}}{8}-\frac{3\left(\frac{\cos(5\,x)}{2}+\frac{\cos(3\,x)}{2}\right)}{8}+\frac{\cos\left(3\,x\right)}{8}+\frac{3\,\cos\left(x\right)}{8}\right)$$

Quizás es complicado ver qué está ocurriendo con estas expresiones tan largas. Vamos a ver cómo se comportan en una un poco más sencilla:

```
(%i37)    eq:cos(2*x)+cos(x)^2$
(%i38)    trigexpand(eq);
(%o38)    2cos(x)^2-sin(x)^2
(%i39)    trigreduce(eq);
(%o39)    cos(2x)+1
          ---------+cos(2x)
              2
(%i40)    trigsimp(eq);
(%o40)    cos(2x)+cos(x)^2
```

Como puedes ver, `trigsimp` intenta escribir la expresión de manera simple, `trigexpand` y `trigreduce` desarrollan y agrupan en términos similares pero mientras una prefiere usar potencias, la otra utiliza múltiplos de la variable. Estos es muy a grosso modo.

Cualquiera de estas órdenes opera de manera similar con funciones hiperbólicas:

```
(%i41)  trigexpand(sinh(2*x)^3);
(%o41)  8 cosh(x)^3 sinh(x)^3
(%i42)  trigreduce(cosh(x+y)+sinh(x)^2);
(%o42)  cosh(y+x)+(cosh(2x)-1)/2
```

Observación 1.5. Al igual que con `expand` o `ratsimp`, se puede ajustar el comportamiento de estas órdenes mediante el valor de algunas variables como `trigexpand`, `trigexpandplus` o `trigexpandtimes`. Consulta la ayuda de *Maxima* si estás interesado.

1.7 La ayuda de *Maxima*

El entorno *wxMaxima* permite acceder a la amplia ayuda incluida con *Maxima* de una manera gráfica. En el mismo menú tenemos algunos comandos que nos pueden ser útiles.

`describe(`*expr*`)`	ayuda sobre *expr*
`example(`*expr*`)`	ejemplo de *expr*
`apropos("`*expr*`")`	comandos relacionados con *expr*
`??`*expr*	comandos que contienen *expr*

En el caso de que conozcamos el nombre del comando sobre el que estamos buscando ayuda, la orden `describe`[3] nos da una breve, a veces no tan breve, explicación sobre la variable, comando o lo que sea que hayamos preguntado.

```
(%i43)  describe(dependencies);  - Variable del sistema: dependencies
        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'.
(%o43)  true
```

[3] El comando `describe(`*expr*`)` produce el mismo resultado que `?`*expr*. Es obligatorio el espacio en blanco entre la interrogación y la expresión.

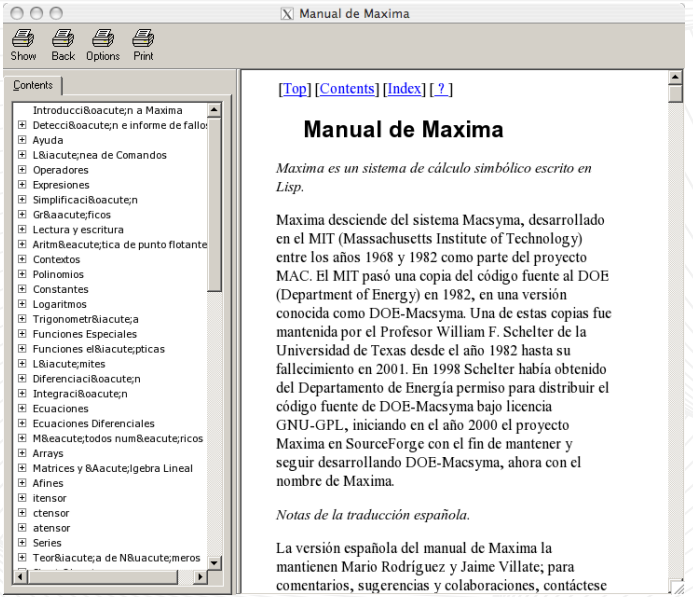


Figura 1.3 Ayuda de *wxMaxima*

Claro que a veces nos equivocamos y no nos acordamos exactamente del nombre del comando

```
(%i44)   describe(plot);  No exact match found for topic ‘plot’.
         Try ‘?? plot’ (inexact match) instead.
(%o44)   false
```

La solución la tenemos escrita justo en la salida anterior: ?? busca en la ayuda comandos, variables, etc. que contengan la cadena "plot".

```
(%i45)   ??plot  0: Funciones y variables para plotdf
         1: Introducción a plotdf
         2: barsplot (Funciones y variables para gráficos estadísticos)
         3: boxplot (Funciones y variables para gráficos estadísticos)
         4: contour_plot (Funciones y variables para gráficos)
         5: gnuplot_close (Funciones y variables para gráficos)
         6: gnuplot_replot (Funciones y variables para gráficos)
         7: gnuplot_reset (Funciones y variables para gráficos)
         8: gnuplot_restart (Funciones y variables para gráficos)
         9: gnuplot_start (Funciones y variables para gráficos)
         10: plot2d (Funciones y variables para gráficos)
         11: plot3d (Funciones y variables para gráficos)
         12: plotdf (Funciones y variables para plotdf)
         13: plot_options (Funciones y variables para gráficos)
         14: scatterplot (Funciones y variables para gráficos estadísticos)
         15: set_plot_option (Funciones y variables para gráficos)
         Enter space-separated numbers, 'all' or 'none':none;
(%o45)   true
```

Si, como en este caso, hay varias posibles elecciones, *Maxima* se queda esperando hasta que escribimos el número que corresponde al ítem en que estamos interesados, o `all` o `none` si estamos interesados en todos o en ninguno respectivamente. Mientras no respondamos a esto no podemos realizar ninguna otra operación.

Si has mirado en el menú de *wxMaxima*, seguramente habrás visto **Ayuda→A propósito**. Su propósito es similar a las dos interrogaciones, ??, que acabamos de ver pero el resultado es levemente distinto:

```
(%i46)  apropos("plot");
(%o46)  [plot,plot2d,plot3d,plotheight,plotmode,plotting,plot_format,
          plot_options,plot_realpart]
```

nos da la lista de comandos en los que aparece la cadena `plot` sin incluir nada más. Si ya tenemos una idea de lo que estamos buscando, muchas veces será suficiente con esto.

Muchas veces es mejor un ejemplo sobre cómo se utiliza una orden que una explicación "teórica". Esto lo podemos conseguir con la orden `example`.

```
(%i47)  example(limit);
(%i48)  limit(x*log(x),x,0,plus)
(%o48)  0
(%i49)  limit((x+1)^(1/x),x,0)
(%o49)  %e
(%i50)  limit(%e^x/x,x,inf)
(%o50)  ∞
(%i51)  limit(sin(1/x),x,0)
(%o51)  ind
(%o51)  done
```

Por último, la ayuda completa de *Maxima* está disponible en la página web de *Maxima*

http://maxima.sourceforge.net/es/

en formato PDF y como página web. Son más de 800 páginas que explican prácticamente cualquier detalle que se te pueda ocurrir.

1.8 Ejercicios

Ejercicio 1.1. Calcula

a) Los 100 primeros decimales del número e,
b) el logaritmo en base 3 de 164232032682606581462314678007092552 89.
c) el arcocoseno hiperbólico de 1,
d) el seno y el coseno de i, y
e) el logaritmo de -2.

Ejercicio 1.2.

a) ¿Qué número es mayor 1000^{999} o 999^{1000}?
b) Ordena de mayor a menor los números π, $\frac{73231844868435875}{37631844868435563}$ y $\cosh(3)/3$.

Ejercicio 1.3. Descompón la fracción $\frac{x^2-4}{x^5+x^4-2\,x^3-2\,x^2+x+1}$ en fracciones simples.

Ejercicio 1.4. Escribe $\operatorname{sen}(5x)\cos(3x)$ en función de $\operatorname{sen}(x)$ y $\cos(x)$.

Ejercicio 1.5. Comprueba si las funciones hiperbólicas y las correspondientes "arco"-versiones son inversas.

2 Gráficos

El objetivo de este capítulo es aprender a representar gráficos en dos y tres dimensiones. Lo haremos, tanto para gráficas en coordenadas cartesianas como para gráficas en coordenadas paramétricas y polares. *wxMaxima* permite hacer esto fácilmente aunque también veremos cómo utilizar el módulo `draw` que nos da algunas posibilidades más sin complicar excesivamente la escritura.

2.1 Funciones

funcion(*var1*,*var2*,..):=(*expr1*,*expr2*,...)	definición de función
define (*func*,*expr*)	la función vale *expr*
fundef(*func*)	devuelve la definición de la función
functions	lista de funciones definidas por el usuario
remfunction(*func1*,*func2*,...)	borra las funciones

Para definir una función en *Maxima* se utiliza el operador :=. Se pueden definir funciones de una o varias variables, con valores escalares o vectoriales,

```
(%i1)   f(x):=sin(x);
(%o1)   f(x):=sin(x)
```

que se pueden utilizar como cualquier otra función.

```
(%i2)   f(%pi/4);
(%o2)   1/sqrt(2)
```

Si la función tiene valores vectoriales o varias variables tampoco hay problema:

```
(%i3)   g(x,y,z):=[2*x,3*cos(x+y)];
(%o3)   g(x,y,z):=[2x,3cos(x+y)]
(%i4)   g(1,%pi,0);
(%o4)   [2,-3cos(1)]
```

También se puede utilizar el comando `define` para definir una función. Por ejemplo, podemos utilizar la función g para definir una nueva función y, de hecho veremos que ésta es la manera correcta de hacerlo cuando la definición involucra funciones previamente definidas, derivadas de funciones, etc. El motivo es que la orden `define` evalúa los comandos que pongamos en la definición.

```
(%i5)   define(h(x,y,z),g(x,y,z)^2);
```

```
(%o5)   h(x,y,z):=[4x^2,9cos(y+x)^2]
```

Eso sí, aunque hemos definido las funciones f, g y h, para utilizarlas debemos añadirles variables:

```
(%i6)   g;
(%o6)   g
```

Si queremos saber cuál es la definición de la función g, tenemos que preguntar

```
(%i7)   g(x,y);
        Too few arguments supplied to g(x,y,z):
        [x,y]
        - an error. To debug this try debugmode(true);
```

pero teniendo cuidado de escribir el número correcto de variables

```
(%i8)    g(x,y,z);
(%o8)    [2x,3cos(y+x)]
```

Esto plantea varias cuestiones muy relacionadas entre sí: cuando llevamos un rato trabajando y hemos definido varias funciones, ¿cómo sabemos cuales eran? y ¿cuál era su definición?. La lista de funciones que hemos definido se guarda en la variable `functions` a la que también puedes acceder desde el menú **Maxima→Mostrar funciones** de manera similar a como accedemos a la lista de variables. En el mismo menú, **Maxima→Borrar función** tenemos la solución a cómo borrar una función (o todas). También podemos hacer esto con la orden `remfunction`.

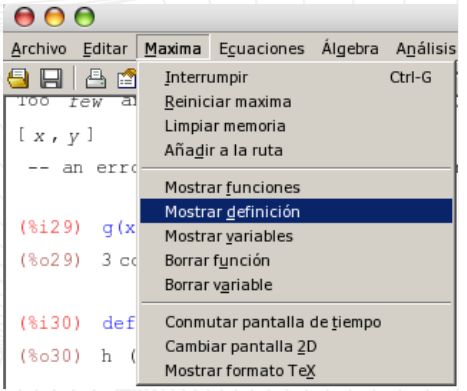


Figura 2.1 Desde el menú podemos consultar las funciones que tenemos definidas, cuál es su definición y borrar algunas o todas ellas

```
(%i9)    functions;
(%o9)    [f(x),g(x,y,z),h(x,y,z)]
```

Ya sabemos preguntar cuál es la definición de cada una de ellas. Más cómodo es, quizás, utilizar la orden `fundef` que nos evita escribir las variables

```
(%i10)   fundef(f);
(%o10)   f(x):=sin(x)
```

que, si nos interesa, podemos borrar

```
(%i11)   remfunction(f);
(%o11)   [f]
```

o, simplemente, borrar todas las que tengamos definidas

```
(%i12)   remfunction(all);
(%o12)   [g,h]
```

Funciones definidas a trozos

Las funciones definidas a trozos plantean algunos problemas de difícil solución para *Maxima*. Esencialmente hay dos formas de definir y trabajar con funciones a trozos:

a) definir una función para cada trozo con lo que tendremos que ocuparnos nosotros de ir escogiendo de elegir la función adecuada, o
b) utilizar una estructura `if-then-else` para definirla.[4]

Cada uno de los métodos tiene sus ventajas e inconvenientes. El primero de ellos nos hace aumentar el número de funciones que definimos, usamos y tenemos que nombrar y recordar. Además de esto, cualquier cosa que queramos hacer, ya sea representar gráficamente o calcular una integral tenemos que plantearlo nosotros. *Maxima* no se encarga de esto. La principal limitación del segundo método es que las funciones definidas de esta manera no nos sirven para derivarlas o integrarlas, aunque sí podremos dibujar su gráfica. Por ejemplo, la función

[4] En la sección 5.3 explicamos con más detalle este tipo de estructuras

$$f(x) = \begin{cases} x^2, & \text{si } x < 0 \\ x^3, & \text{en otro caso} \end{cases}$$

la podemos definir de la siguiente forma utilizando el segundo método

```
(%i13)   f(x):=if x< 0 then x^2 else x^3;
(%o13)   f(x):=if x< 0 then x^2 else x^3
```

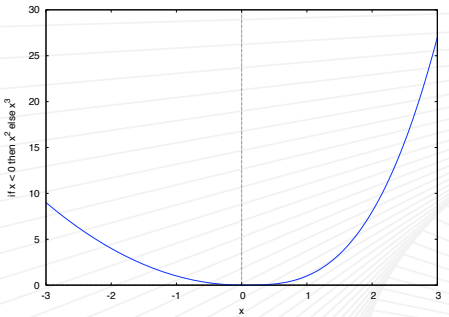
y podemos evaluarla en un punto

```
(%i14)   f(-2);
(%o14)   4
(%i15)   f(2);
(%o15)   8
```

o dibujarla

```
(%i16)   plot2d(f(x),[x,-3,3]);
```

```
(%o16)
```



pero no podemos calcular $\int_{-3}^{3} f(x)\,dx$:

```
(%i17)   integrate(f(x),x,-3,3);
(%o17)   ∫_{-3}^{3} if x<0 then x^2 else x^3 dx
```

La otra posibilidad es mucho más de andar por casa, pero muy práctica. Podemos definir las funciones

```
(%i18)   f1(x):=x^2$
(%i19)   f2(x):=x^3$
```

y decidir nosotros cuál es la que tenemos que utilizar:

```
(%i20)   integrate(f1(x),x,-3,0)+integrate(f2(x),x,0,3);
```

```
(%o20)    117/4
```

Evidentemente, si la función tiene "muchos" trozos, la definición se alarga; no cabe otra posibilidad. En este caso tenemos que anidar varias estructuras if-then-else o definir tantas funciones como trozos. Por ejemplo, la función

$$g(x) = \begin{cases} x^2, & \text{si } x \leq 1, \\ \operatorname{sen}(x), & \text{si } 1 \leq x \leq \pi, \\ -x+1, & \text{si } x > \pi \end{cases}$$

la podemos escribir como

```
(%i21)    g(x):=if x<=1 then x^2 else
            if x <= %pi then sin(x) else -x+1$
(%i22)    [g(-3),g(2),g(5)];
(%o22)    [9,sin(2),-4]
```

2.2 Gráficos en el plano con `plot2d`

2.2.1 Coordenadas cartesianas

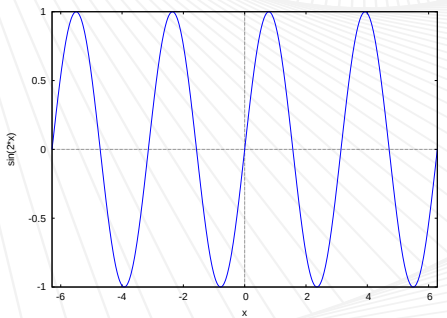
El comando que se utiliza para representar la gráfica de una función de una variable real es `plot2d` que actúa, como mínimo, con dos parámetros: la función (o lista de funciones a representar), y el intervalo de valores para la variable x. Al comando `plot2d` se puede acceder también a través del menú **Gráficos→Gráficos 2D** o, directamente, a través del botón Gráficos 2D.

`plot2d(f(x),[x,a,b])`	gráfica de $f(x)$ en $[a,b]$
`plot2d([f1(x),f2(x),...],[x,a,b])`	gráfica de una lista de funciones en $[a,b]$

Por ejemplo:

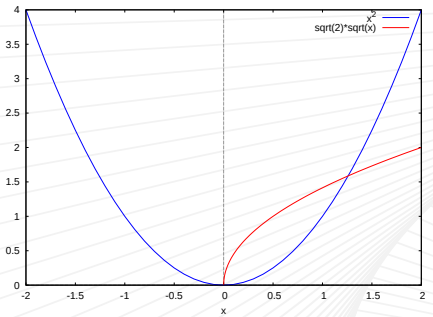
```
(%i23)  plot2d(sin(2*x),[x,-2*%pi,2*%pi]);
```

(%o23)




```
(%i24)  plot2d([x^2,sqrt(2*x)],[x,-2,2]);
```

(%o24)



Observa en esta última salida cómo el programa asigna a cada gráfica un color distinto para diferenciarlas mejor y añade la correspondiente explicación de qué color representa a cada función.

Cuando pulsamos el botón (Gráficos 2D), aparece una ventana de diálogo con varios campos que podemos completar o modificar:

a) Expresión(es). La función o funciones que queramos dibujar. Por defecto, *wxMaxima* rellena este espacio con % para referirse a la salida anterior.
b) Variable x. Aquí establecemos el intervalo de la variable x donde queramos representar la función.
c) Variable y. Ídem para acotar el recorrido de los valores de la imagen.
d) Graduaciones. Nos permite regular el número de puntos en los que el programa evalúa una función para su representación en polares. Veremos ejemplos en la sección siguiente.

e) Formato. *Maxima* realiza por defecto la gráfica con un programa auxiliar. Si seleccionamos `en línea`, dicho programa auxiliar es *wxMaxima* y obtendremos la gráfica en una ventana alineada con la salida correspondiente. Hay dos opciones más y ambas abren una ventana externa para dibujar la gráfica requerida: `gnuplot` es la opción por defecto que utiliza el programa *Gnuplot* para realizar la representación; también está disponible la opción `openmath` que utiliza el programa *XMaxima*. Prueba las diferentes opciones y decide cuál te gusta más.

f) Opciones. Aquí podemos seleccionar algunas opciones para que, por ejemplo, dibuje los ejes de coordenadas (`"set zeroaxis;"`); dibuje los ejes de coordenadas, de forma que cada unidad en el eje Y sea igual que el eje X (`"set size ratio 1; set zeroaxis;"`); dibuje una cuadrícula (`"set grid;"`) o dibuje una gráfica en coordenadas polares (`"set polar; set zeroaxis;"`). Esta última opción la comentamos más adelante.

g) Gráfico al archivo. Guarda el gráfico en un archivo con formato Postscript.

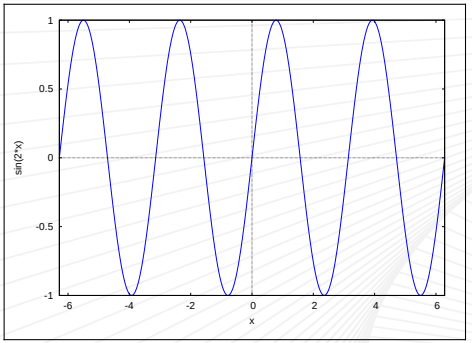
Figura 2.2 Gráficos en 2D

Evidentemente, estas no son todas las posibles opciones. La cantidad de posibilidades que tiene *Gnuplot* es inmensa.

Observación 2.1. El prefijo "wx" añadido a `plot2d` o a cualquiera del resto de las órdenes que veremos en este capítulo (`plot3d, draw2d, draw3d`) hace que *wxMaxima* pase automáticamente a mostrar los gráficos en la misma ventana y no en una ventana separada. Es lo mismo que seleccionar `en línea`. Por ejemplo,

```
(%i25)   wxplot2d(sin(2*x),[x,-2*%pi,2*%pi]);
```

(%t25)

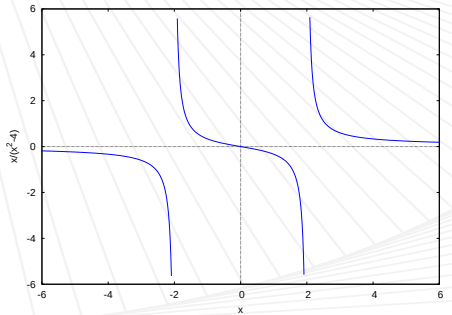


Es complicado representar una ventana separada en unas notas escritas así que, aunque no utilicemos `wxplot2d`, sí hemos representado todas las gráficas a continuación de la correspondiente orden.

Veamos algunos ejemplos de las opciones que hemos comentado. Podemos añadir ejes,

```
(%i26)  plot2d(x/(x^2-4),[x,-6,6],[y,-6,6],
          [gnuplot_preamble, "set zeroaxis;"])$
```

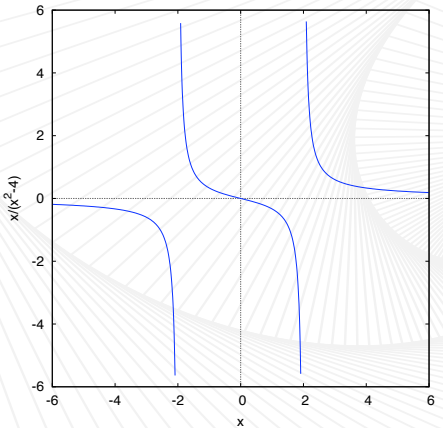
(%o26)



podemos cambiar la proporción entre ejes, por ejemplo, `set size ratio 1` dibuja ambos ejes con el mismo tamaño en pantalla, `set size ratio 2` o `set size ratio 0.5` dibuja el eje X el doble o la mitad de grande que el eje Y

```
(%i27)   plot2d(x/(x^2-4),[x,-6,6],[y,-6,6],
           [gnuplot_preamble, "set size ratio 1; set zeroaxis;"])$
```

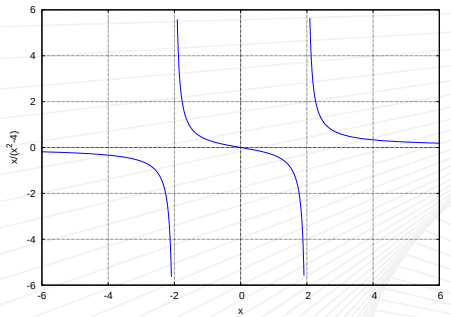
(%o27)



o podemos añadir una malla que nos facilite la lectura de los valores de la función.

```
(%i28)   plot2d(x/(x^2-4),[x,-6,6],[y,-6,6],
           [gnuplot_preamble, "set grid;"])$
```

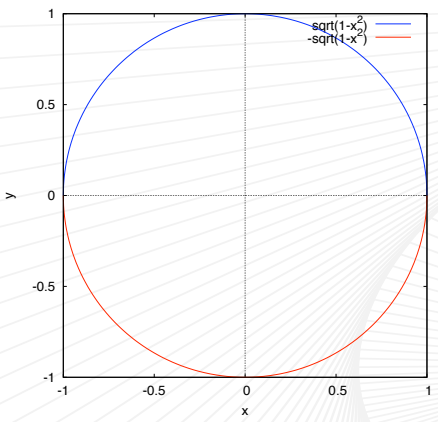
(%o28)



Con el siguiente ejemplo vamos a ver la utilidad de la opción `"set size ratio 1; set zeroaxis;"`. En primer lugar dibujamos las funciones $\sqrt{1-x^2}$ y $-\sqrt{1-x^2}$, con $x \in [-1,1]$. El resultado debería ser la circunferencia unidad. Sin embargo, aparentemente es una elipse. Lo arreglamos de la siguiente forma:

```
(%i29)  plot2d([sqrt(1-x^2,-sqrt(1-x^2)],[x,-1,1],[y,-1,1],
          [gnuplot_preamble, "set size ratio 1; set zeroaxis;"])$
```

(%o29)

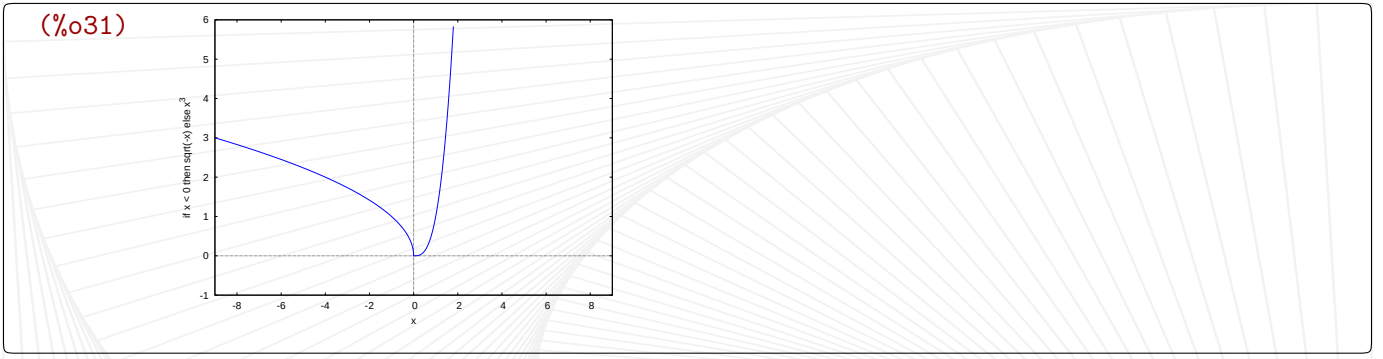


También podemos dibujar gráficas de funciones a trozos. Antes, tenemos que recordar cómo se definen estas funciones. Lo hacemos con un ejemplo. Consideremos la función $f : \mathbb{R} \to \mathbb{R}$ definida como

$$f(x) = \begin{cases} \sqrt{-x} & \text{si } x < 0 \\ x^3 & \text{si } x \geq 0. \end{cases}$$

Vamos, en primer lugar, a presentársela a *Maxima*:

```
(%i30)  f(x):= if x<0 then sqrt(-x) else x^3;
(%o30)  f(x):= if x<0 then √(-x) else x^3
(%i31)  plot2d(f(x),[x,-9,9],[y,-1,6],
          [gnuplot_preamble,"set zeroaxis;"])$
```



2.2.2 Gráficos en coordenadas polares

Para dar un punto del plano, tenemos que indicar los valores de las proyecciones sobre los ejes X e Y (esto es a lo que llamamos coordenadas cartesianas) o podemos indicar la distancia al origen y el ángulo que forma con una dirección fija (en nuestro caso la parte positiva del eje X).

Al representar una curva en coordenadas polares estamos escribiendo la longitud del vector como una función que depende del ángulo. En otras palabras, para cada ángulo fijo decimos cuál es el módulo del vector. El ejemplo más sencillo de función que a cualquiera se nos viene a la cabeza son las funciones constantes. La función $f : [0, 2\pi] \to \mathbb{R}$, $f(\vartheta) = 1$ tiene como imagen aquellos vectores que tienen módulo 1 y argumento entre 0 y 2π. Para ello, tenemos que seleccionar `"set polar; set zeroaxis;"` en el campo **Opciones** de `Gráficos 2D`.

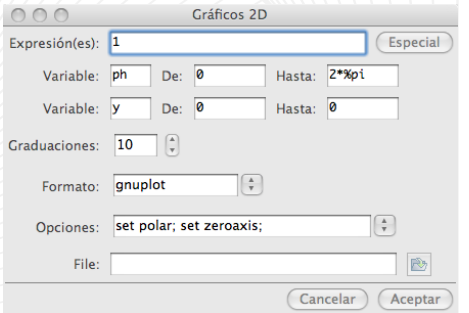
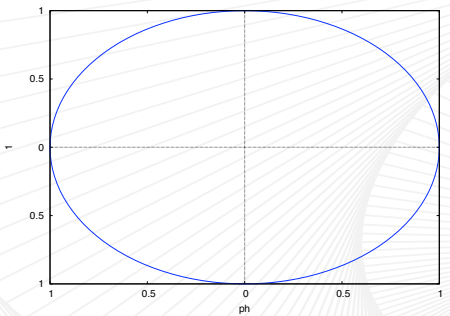


Figura 2.3 Gráfico en coordenadas polares

```
(%i32)   plot2d([1], [ph,0,2*%pi],
           [plot_format, gnuplot],
           [gnuplot_preamble, "set polar; set zeroaxis;"])$`
```

(%o32)

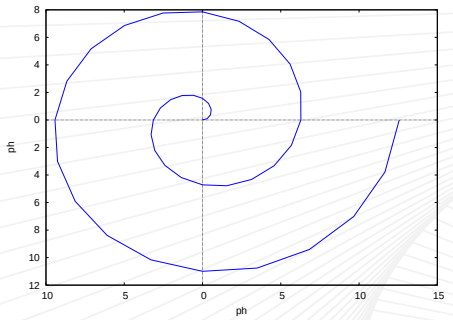


Si la función es creciente, la identidad por ejemplo, el módulo aumenta con el argumento. En este caso obtenemos una espiral.[5]

```
(%i33)   plot2d(ph,[ph,0,4*%pi],
           [gnuplot_preamble, "set polar; set zeroaxis;"])$
```

[5] Como puedes observar, la variable por defecto para indicar el argumento es ph y no la "x" usual cuando trabajamos en coordenadas cartesianas.

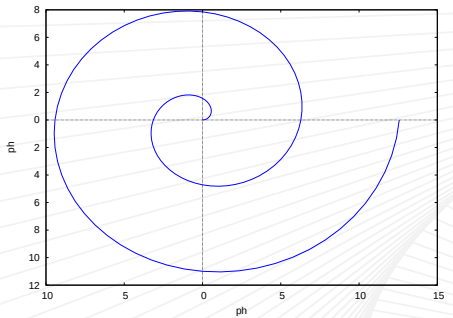
(%o33)



Observamos que la hélice resultante no es nada "suave". Para conseguir el efecto visual de una línea curva como es esta hélice, añadimos el parámetro `nticks`. Por defecto, para dibujar una gráfica en paramétricas el programa evalúa en 10 puntos. Para aumentar este número de puntos, aumentamos dicho parámetro, por ejemplo `nticks=30`, o bien, podemos regularlo desde el campo **Graduaciones** dentro de de `Gráficos 2D`.

```
(%i34)   plot2d(ph,[ph,0,4*%pi],
           [gnuplot_preamble, "set polar; set zeroaxis;"], [nticks,30])$
```

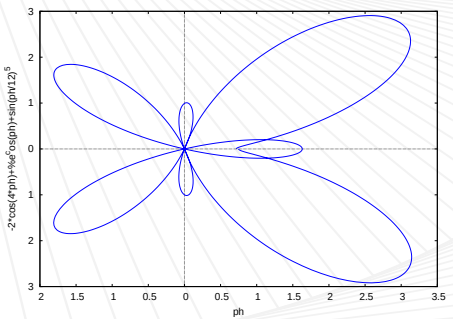
(%o34)



Si representamos la función: $r(\theta) = e^{\cos(\theta)} - 2\cos(4\theta) + \text{sen}\left(\frac{\theta}{12}\right)^5$ obtenemos algo parecido a una mariposa.

```
(%i35)  r(ph):=exp(cos(ph))-2*cos(4*ph)+sin(ph/12)**5$
(%i36)  plot2d(r(ph),[ph,0,2*%pi],
          [gnuplot_preamble, "set polar; set zeroaxis;"])$
```

(%o36)



2.2.3 Gráficos en coordenadas paramétricas

El programa *wxMaxima* nos permite también representar curvas en forma paramétrica, es decir, curvas definidas como $(x(t), y(t))$ donde el parámetro t varía en un determinado intervalo compacto $[a, b]$. Para ello, dentro del comando `plot2d` añadimos "parametric" de la forma siguiente:

`plot2d([parametric,x(t),y(t),[t,a,b]])` gráfica de la curva $(x(t), y(t))$ en $[a, b]$

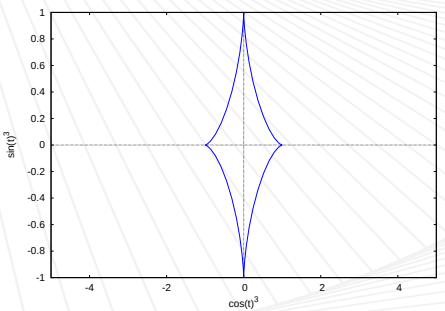
Para acceder a esta opción de la función `plot2d` podemos hacerlo a través del botón `Especial` que aparece en la parte superior derecha de la ventana de diálogo `Gráficos 2D`.

Para terminar, aquí tienes algunas curvas planas interesantes.

Astroide: Es la curva trazada por un punto fijo de un círculo de radio r que rueda sin deslizar dentro de otro círculo fijo de radio 4r. Sus ecuaciones paramétricas son:

```
(%i37)  plot2d([parametric,cos(t)^3,sin(t)^3,[t,0,2*%pi],[nticks,50]])
```

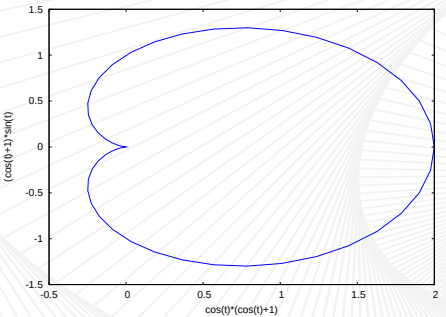
(%o37)



Cardioide: Es la curva trazada por un punto fijo de un círculo de radio r que rueda sin deslizar alrededor de otro círculo fijo del mismo radio. Sus ecuaciones paramétricas son.

```
(%i38)  plot2d([parametric,(1+cos(t))*cos(t),(1+cos(t)*sin(t)),
          [t,0,2*%pi],[nticks,50]])
```

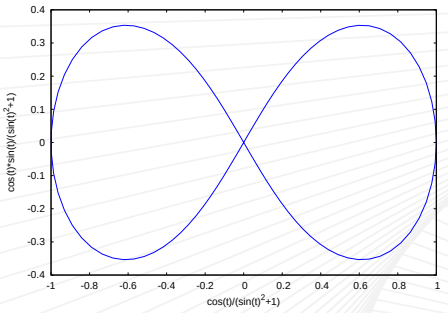
(%o38)



Lemniscata de Bernoulli: Es el lugar geométrico de los puntos P del plano, cuyo producto de distancias a dos puntos fijos F_1 y F_2, llamados focos, verifica la igualdad $|P-F_1|\,|P-F_2| = \frac{1}{4}\,|F_1-F_2|^2$. En coordenadas cartesianas esta curva viene dada por la ecuación $(x^2+y^2)^2 = x^2-y^2$. Aquí tienes sus ecuaciones paramétricas.

```
(%i39)  plot2d([parametric, (cos(t))/(1+sin(t)^2),
          (cos(y)*sin(t))/(1+sin(t)^2), [t,0,2*%pi],[nticks,70]])
```

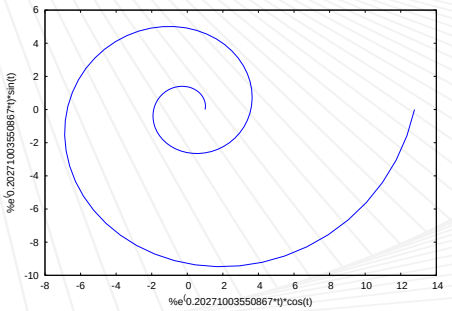
(%o39)



Espiral equiangular: También llamada espiral logarítmica. Es aquella espiral en la que el radio vector corta a la curva en un ángulo constante α. Sus ecuaciones paramétricas son:

```
(%i40)  %alpha:%pi/2-0.2$
(%i41)  plot2d([parametric,exp(t*cot(%alpha))*cos(t),
          exp(t*cot(%alpha))*sin(t),[t,0,4*%pi],[nticks,90]])
```

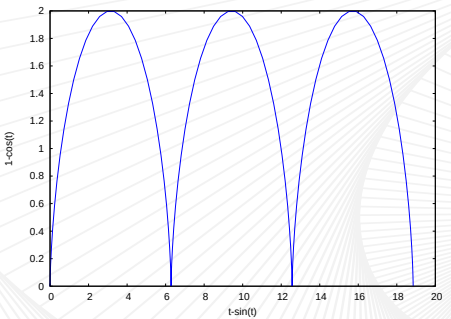
(%o41)



Cicloide: También conocida como tautocrona o braquistocrona. Es la curva que describiría un punto de una circunferencia que avanza girando sin deslizar. Sus ecuaciones paramétricas son:

```
(%i42)   plot2d([parametric,t-sin(t),1-cos(t),[t,0,6*%pi],[nticks,90]])
```

(%o42)

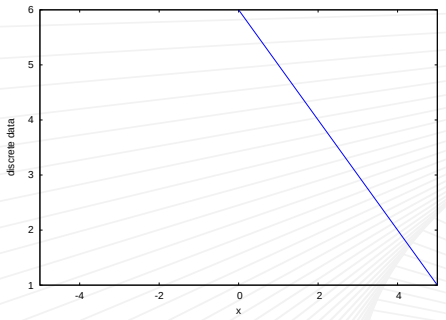


2.2.4 Gráficas de curvas poligonales

Además de dibujar curvas definidas de forma paramétrica, podemos dibujar líneas poligonales, haciendo uso de una opción más dentro de **Especial** (concretamente, `Gráfico discreto`) en la ventana de diálogo de **Gráficos 2D**. Cuando elegimos esta opción, aparece una nueva ventana en la que tendremos que escribir las coordenadas, separadas por comas, de los puntos que van a ser los vértices de la curva poligonal que queremos dibujar. Por ejemplo, para dibujar la recta que une los puntos $(0, 6)$ y $(5, 1)$, escribimos:

```
(%i43)   plot2d([discrete,[0,5],[6,1]],
           [x,-5,5]);
```

(%o43)

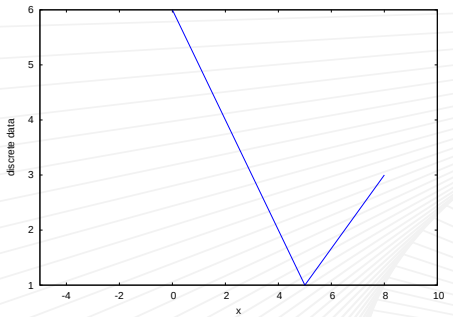


`plot2d([discrete,[x1,x2,...],[y1,y2,...]],`*opc*`)`	poligonal que une los puntos $(x1, y1)$, $(x2, y2)$,...
`plot2d([discrete,[[x1,y1],[x2,y2],...]],`*opc*`)`	poligonal que une los puntos $(x1, y1)$, $(x2, y2)$,...

Si lo que queremos es pintar una línea poligonal sólo tenemos que pasar la lista de las primeras y de las segundas coordenadas:

```
(%i44)  plot2d([discrete,[0,5,8],[6,1,3]],
          [x,-5,10]);
```

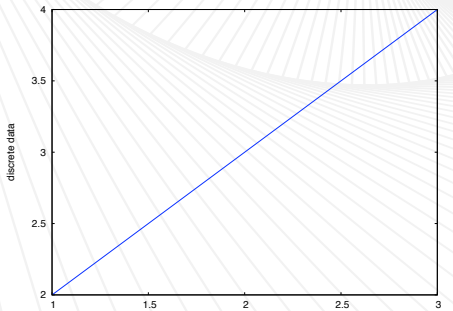
(%o44)



o, agrupar los puntos en una lista,

```
(%i45)  plot2d([discrete,[[1,2],[2,3],[3,4]]]);
```

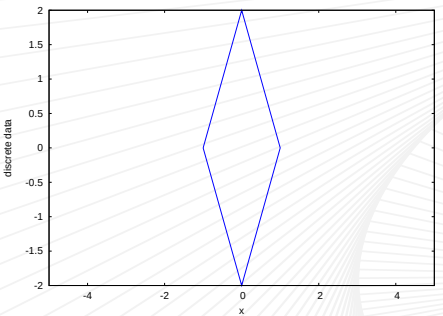
(%o45)



Si el primer y el último punto coinciden se obtiene lo esperable: una línea poligonal cerrada. Por ejemplo, el rombo de vértices $(-1,0)$, $(0,-2)$, $(1,0)$ y $(0,2)$:

```
(%i46)  plot2d([discrete,[-1,0,1,0,-1],[0,-2,0,2,0]],
          [x,-5,5]);
```

(%o46)

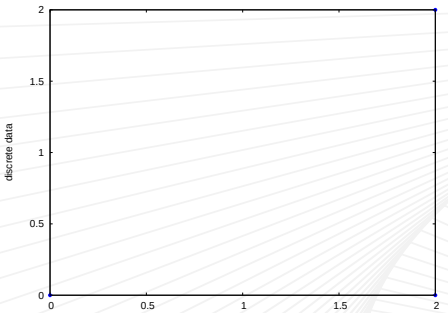


Puntos y segmentos

¿Y si lo que queremos es pintar puntos en el plano? Para ello definimos la lista de puntos que queramos pintar y a continuación, dentro del `plot2d` añadimos la opción `"style,points"` que dibuja los puntos y no los segmentos que los unen.

```
(%i47)  xy:[[0,0],[2,0],[2,2]]$
(%i48)  plot2d([discrete,xy],[style,points]);
```

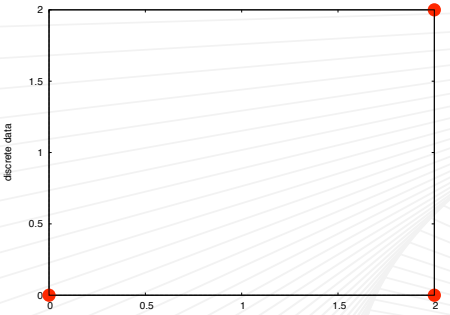
```
(%o48)
```



Si queremos modificar el aspecto de los puntos a pintar, escribiremos tres números (`[points,a,b,c]`) después de la opción `points` cuyo significado es el siguiente: a, radio de los puntos; b, índice del color (1 para azul, 2 para rojo,...) ; c, tipo de objeto utilizado (puntos, circunferencias, cruces, asteriscos, cuadrados,...). De la misma forma, si queremos modificar el aspecto de los segmentos: dentro de `style` elegimos la opción `lines` seguida de dos cifras (`[lines,`α`,`β`]`) que hacen referencia al ancho de la línea (α) y al color (β).

```
(%i49)   plot2d([discrete,xy],[style,[points,10,2]]);
```

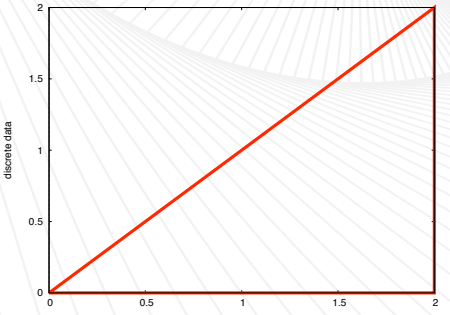
(%o49)



Dibujemos ahora un triángulo rectángulo de vértices: $(0,0)$, $(2,0)$ y $(2,2)$ con los lados en rojo.

```
(%i50)  plot2d([discrete,[0,2,2,0],[0,0,2,0]],[style,[lines,15,2]]);
```

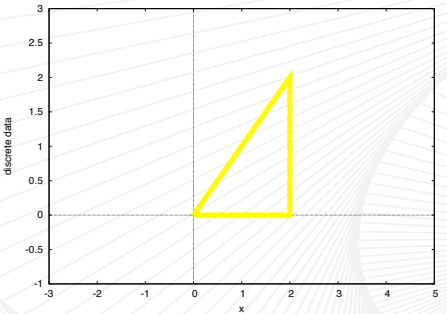
(%o50)



Observa que si no indicamos ningún rango, *Maxima* ajusta el dominio a los valores que estemos representando. Podemos ampliar el rango de las variables x e y y obtenemos algo así.

```
(%i51)  plot2d([discrete, [0,2,2,0],[0,0,2,0]],
          [x,-3,5],[y,-1,3],
          [style,[lines,25,5]]);
```

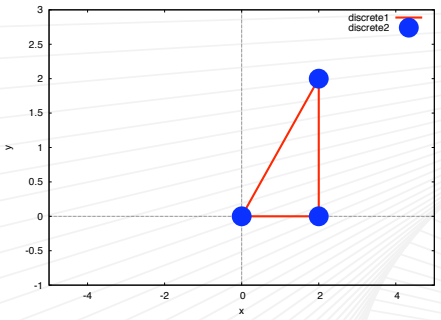
(%o51)



Y si ahora queremos dibujar el triángulo anterior (en rojo) junto con los vértices (en azul):

```
(%i52)  plot2d([[discrete,[0,2,2,0],[0,0,2,0]],[discrete,xy]],
          [x,-5,5],[y,-1,3],
          [style,[lines,10,2],[points,15,1,1]]);
```

(%o52)



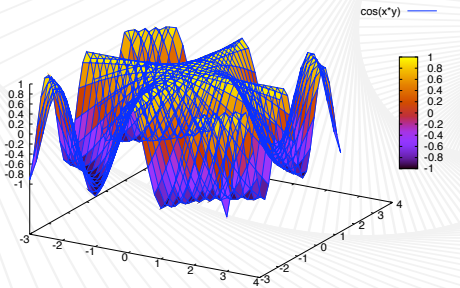
2.3 Gráficos en 3D

Con *Maxima* se pueden representar funciones de dos variables de forma similar a como hemos representado funciones de una. La principal diferencia es que vamos a utilizar el comando `plot3d` en lugar de `plot2d`, pero igual que en el caso anterior, son obligatorios la función o funciones y el dominio que tiene que ser de la forma $[a,b]\times[c,d]$.

`plot3d(f(x,y),[x,a,b],[y,c,d])` gráfica de $f(x,y)$ en $[a,b]\times[c,d]$

```
(%i53)  plot3d(cos(x*y),[x,-3,3],[y,-3,3]);
```

(%o53)



Observación 2.2. La gráfica que acabas de obtener se puede girar sin más que pinchar con el ratón sobre ella y deslizar el cursor.

Al comando `plot3d` se accede a través del menú **Gráficos**→**Gráficos 3D** o del botón (Gráficos 3D). Después de esto aparece la ventana de la Figura 2.4 con varios campos para rellenar:

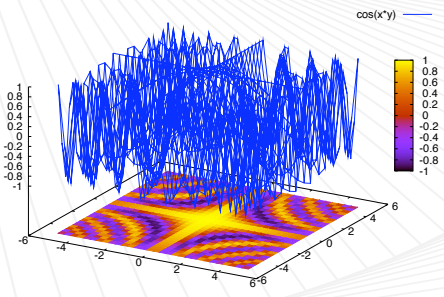
a) Expresión. La función o funciones que vayamos a dibujar.
b) Variable. Hay dos campos para indicar el dominio de las dos variables.
c) Cuadrícula. Indica cuántas valores se toman de cada variable para representar la función. Cuanto mayor sea, más suave será la representación a costa de aumentar la cantidad de cálculos.
d) Formato. Igual que en `plot2d`, permite escoger qué programa se utiliza para representar la función. Se puede girar la gráfica en todos ellos salvo si escoges "en línea".
e) Opciones. Las comentamos a continuación.
f) Gráfico al archivo. Permite elegir un fichero donde se guardará la gráfica.

Figura 2.4 Gráficos 3D

Quizá la mejor manera de ver el efecto de las opciones es repetir el dibujo anterior. La primera de ellas es `"set pm3d at b"` que dibuja la superficie usando una malla y en la base añade curvas de nivel (como si estuviéramos mirando la gráfica desde arriba):

```
(%i54)  plot3d(cos(x*y), [x,-5,5], [y,-5,5], [plot_format,gnuplot],
          [gnuplot_preamble, "set pm3d at b"])$
```

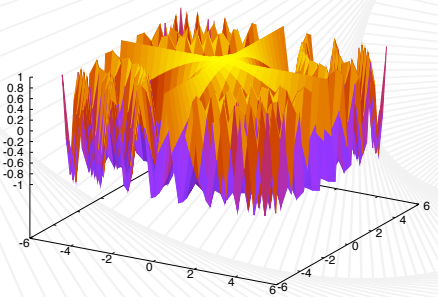
(%o54)



La segunda hace varias cosas, "set pm3d at s" nos dibuja la superficie coloreada, "unset surf elimina la malla y "unset colorbox" elimina la barra que teníamos en la derecha con la explicación de los colores y su relación con la altura (el valor de la función):

```
(%i55)   plot3d(cos(x*y), [x,-5,5], [y,-5,5],
           [plot_format,gnuplot],
           [gnuplot_preamble, "set pm3d at s; unset surf;
             unset colorbox"]$
```

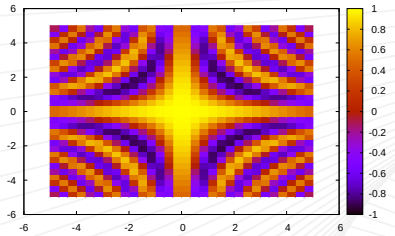
(%o55)



La tercera, "set pm3d map", nos dibuja un mapa de curvas de nivel con alguna ayuda adicional dada por el color:

```
(%i56)   plot3d(cos(x*y), [x,-5,5], [y,-5,5], [plot_format,gnuplot],
           [gnuplot_preamble, "set pm3d map; unset surf"]$
```

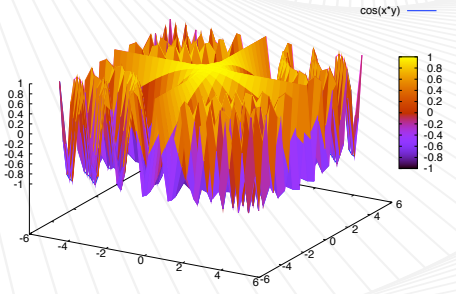
(%o56)



La cuarta, "set hidden3d", sólo muestra la parte de la superficie que sería visible desde nuestro punto de vista. En otras palabras, hace la superficie sólida y no transparente.

```
(%i57)  plot3d(cos(x*y), [x,-5,5], [y,-5,5], [plot_format,gnuplot],
          [gnuplot_preamble, "set hidden3d"])$
```

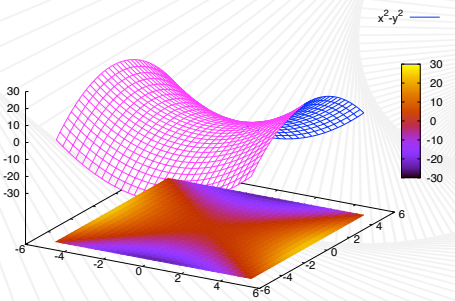
(%o57)



En el dibujo anterior (en el papel) es posible que no aprecie bien. A simple vista parece el mismo dibujo que teníamos dos salidas antes. Observa bien: hay una pequeña diferencia. El uso de `pm3d` hace que se coloree el dibujo, pero cuando decimos que no se muestra la parte no visible de la figura nos estamos refiriendo a la malla. Quizá es mejor dibujar la malla y el manto de colores por separado para que se vea la diferencia. Esta opción no viene disponible por defecto en *wxMaxima*. Ten en cuenta que las opciones que tiene *Gnuplot* son casi infinitas y sólo estamos comentando algunas.

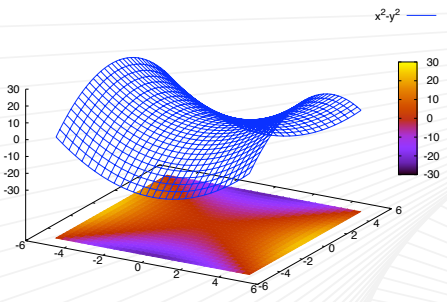
```
(%i58)  plot3d(x^2-y^2, [x,-5,5], [y,-5,5], [plot_format,gnuplot],
          [gnuplot_preamble, "set pm3d at b; set hidden3d"])$
```

(%o58)




```
(%i59)  plot3d(x^2-y^2, [x,-5,5], [y,-5,5], [plot_format,gnuplot],
          [gnuplot_preamble, "set pm3d at b"])$
```

(%o59)



La quinta y la sexta opciones nos permiten dibujar en coordenadas esféricas o cilíndricas. Ya veremos ejemplos más adelante.

2.4 Gráficos con `draw`

El módulo "draw" es relativamente reciente en la historia de *Maxima* y permite dibujar gráficos en 2 y 3 dimensiones con relativa comodidad. Se trata de un módulo adicional que hay que cargar previamente para poder usarlo. Comencemos por esto.

```
(%i60)   load(draw)$
```

gr2d(*opciones, objeto gráfico,...*)	gráfico dos dimensional
gr3d(*opciones, objeto gráfico,...*)	gráfico tres dimensional
draw(*opciones, objeto gráfico,...*)	dibuja un gráfico
draw2d(*opciones, objeto gráfico,...*)	dibuja gráfico dos dimensional
draw3d(*opciones, objeto gráfico,...*)	dibuja gráfico tres dimensional

Una vez cargado el módulo draw, podemos utilizar las órdenes `draw2d` y `draw3d` para dibujar gráficos en 2 y 3 dimensiones o `draw`. Un gráfico está compuesto por varias opciones y el objeto gráfico que queremos dibujar. Por ejemplo, en dos dimensiones tendríamos algo así:

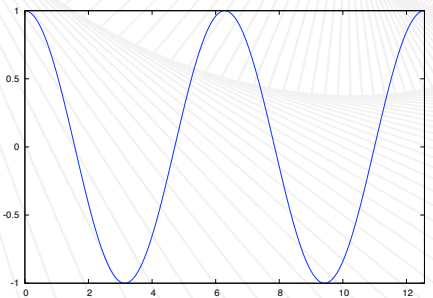
```
objeto:gr2d(
    color=blue,
    nticks=60,
    explicit(cos(t),t,0,2$*$\%pi)
    )
```

Las opciones son numerosas y permiten controlar prácticamente cualquier aspecto imaginable. Aquí comentaremos algunas de ellas pero la ayuda del programa es insustituible. En segundo lugar aparece el objeto gráfico. En este caso "`explicit(cos(t),t,0,2*%pi)`". Estos pueden ser de varios tipos aunque los que más usaremos son quizás `explicit` y `parametric`. Para dibujar un gráfico tenemos dos posibilidades

a) Si tenemos previamente definido el objeto, `draw(objeto)`, o bien,
b) `draw2d(definición del objeto)` si lo definimos en ese momento para dibujarlo.

Por ejemplo,

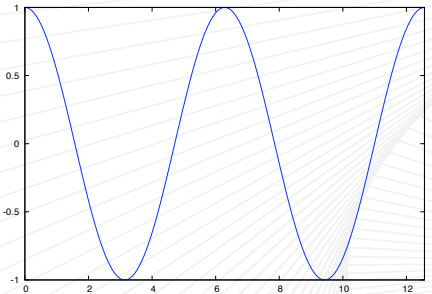
```
(%i61)   coseno:gr2d(
           color=blue,
           explicit(cos(x),x,0,4*%pi))$
(%i62)   draw(coseno);
(%o62)
```



da el mismo resultado que

```
(%i63)  draw2d(
          color=blue,
          explicit(cos(x),x,0,4*%pi)$
```

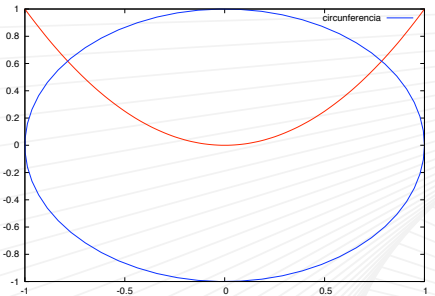
(%o63)



También podemos representar más de un objeto en un mismo gráfico. Simplemente escribimos uno tras otro separados por comas. En el siguiente ejemplo estamos mezclando una función dada explícitamente y una curva en coordenadas paramétricas.

```
(%i64)  draw2d(
          color=red,
          explicit(x^2,x,-1,1),
          color=blue,nticks=60,
          parametric(cos(t),sin(t),t,0,2*%pi));
```

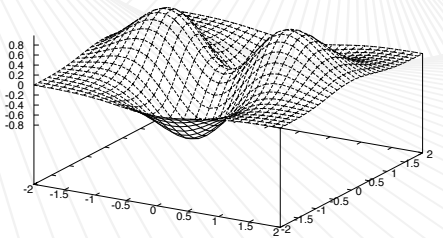
(%o64)



En tres dimensiones, la construcción es similar

```
(%i65)   draw3d(surface_hide=true,
           explicit((x^2-y^2)*exp(1-x^2-y^2),x,-2,2,y,-2,2));
```

(%o65)



Vamos a comentar brevemente alguno de los objetos y de las opciones del módulo `draw`. Comenzamos con las opciones e iremos poniendo ejemplos con cada uno de los objetos.

2.4.1 Opciones

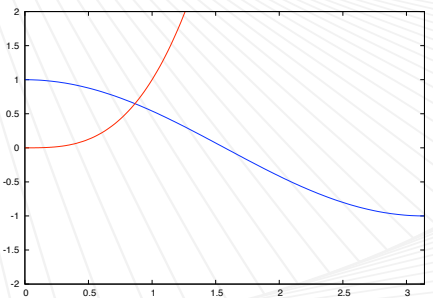
Es importante destacar que hay dos tipos de opciones: locales y globales. Las locales sólo afectan al objeto que les sigue y, obligatoriamente, tienen que precederlo. En cambio las globales afectan a todos los objetos dentro de la orden `draw` y da igual su posición (aunque solemos escribirlas todas juntas al final).

Opciones globales

`xrange, yrange, zrange`: rango de las variables x, y, z. Por defecto se ajusta automáticamente al objeto que se esté representando pero hay ocasiones en que es preferible fijar un rango común.

```
(%i66)  draw2d(color=blue,
          explicit(cos(x),x,0,4*%pi),
          color=red,
          explicit(x^3,x,-5,5),
          xrange=[0,%pi],yrange=[-2,2])$
```

(%o66)

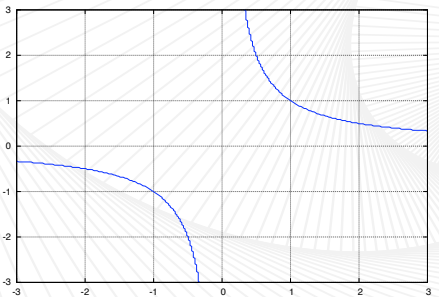


Si en el ejemplo anterior no limitamos el rango a representar, al menos en la coordenada y, es difícil poder ver a la vez la función coseno que toma valores entre 1 y -1 y la función x^3 que en 5 vale bastante más.

`grid`: dibuja una malla sobre el plano XY si vale `true`. Es una opción global.

```
(%i67)   draw2d(
           color=blue,nticks=100,
           implicit(x*y=1,x,-3,3,y,-3,3),
           grid=true)$
```

(%o67)

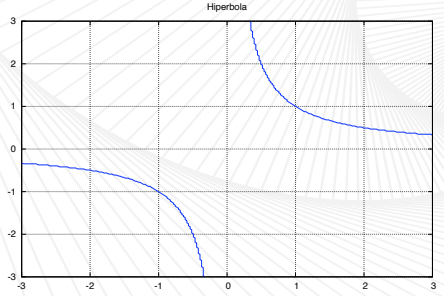


Acabamos de dibujar la hipérbola definida implícitamente por la ecuación $xy = 1$. La opción `grid` nos ayuda a hacernos una idea de los valores que estamos representando.

`title` ="título de la ventana" nos permite poner un título a la ventana donde aparece el resultado final. Es una opción global.

```
(%i68)  draw2d(
          color=blue,
          nticks=100,
          implicit(x*y=1,x,-3,3,y,-3,3),
          grid=true,
          title="Hiperbola"
        )$
```

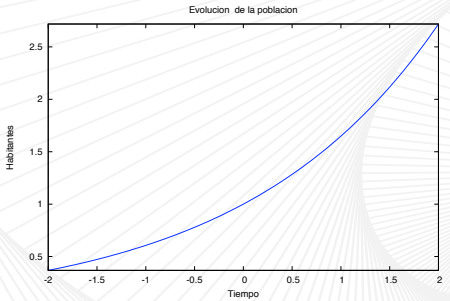
(%o68)



`xlabel, ylabel, zlabel`: indica la etiqueta de cada eje. Es una opción global.

```
(%i69)   draw2d(color=blue,
           explicit(exp(x/2),x,-2,2),
           xlabel="Tiempo",
           ylabel="Habitantes",
           title="Evolucion de la poblacion");
```

(%o69)



`xaxis, yaxis, zaxis`: si vale `true` se dibuja el correspondiente eje. Es una opción global.

`enhanced3d`: si vale `true` se colorean las superficies en gráficos tridimensionales.

Opciones locales

`point_size`: tamaño al que se dibujan los puntos. Su valor por defecto es 1. Afecta a los objetos de tipo `point`.

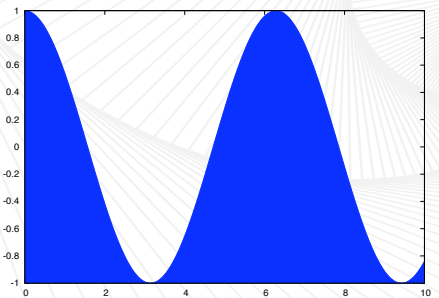
`point_type`: indica cómo se van a dibujar los puntos . El valor para esta opción puede ser un nombre o un número: none (-1), dot (0),plus (1), multiply (2), asterisk (3), square (4), filled_square (5), circle (6),

filled_circle (7), up_triangle (8), filled_up_triangle (9), down_triangle (10), filled_down_triangle (11), diamant (12) y filled_diamant (13). Afecta a los objetos de tipo `point`.

`filled_func`: esta orden nos permite rellenar con un color la gráfica de una función. Existen dos posibilidades: si `filled_func` vale `true` se rellena la gráfica de la función hasta la parte inferior de la ventana con el color establecido en `fill_color`

```
(%i70)   draw2d(fill_color=blue,
           filled_func=true,
           explicit(cos(x),x,0,10)
         ):
```

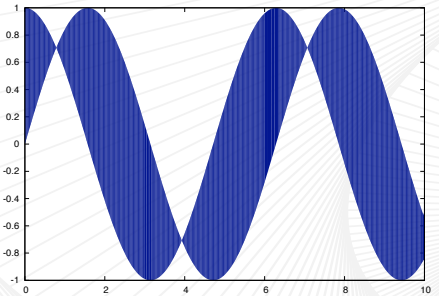
(%o70)



en cambio, si `filled_func` es una función, entonces se colorea el espacio entre dicha función y la gráfica que estemos dibujando

```
(%i71)  draw2d(
          filled_func=sin(x),
          fill_color=navy,
          explicit(cos(x),x,0,10));
```

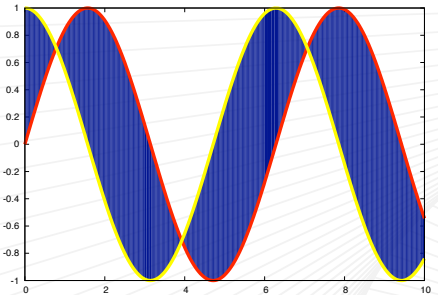
(%o71)



En este caso, tenemos sombreada el área entre las funciones seno y coseno. Podemos dibujar éstas también pero es necesario suprimir el sombreado si queremos que no tape a las funciones:

```
(%i72)  draw2d(
          filled_func=sin(x),fill_color=navy,
          explicit(cos(x),x,0,10),
          filled_func=false,color=red,line_width=10,
          explicit(sin(x),x,0,10),
          color=yellow,line_width=10,
          explicit(cos(x),x,0,10));
```

(%o72)



`fill_color`: ver el apartado anterior `filled_func`.

`color`: especifica el color en el que se dibujan líneas, puntos y bordes de polígonos. Directamente de la ayuda de *Maxima*:

```
Los nombres de colores disponibles son: ‘"white"’, ‘"black"’,
‘"gray0"’, ‘"grey0"’, ‘"gray10"’, ‘"grey10"’, ‘"gray20"’,
‘"grey20"’, ‘"gray30"’, ‘"grey30"’, ‘"gray40"’, ‘"grey40"’,
‘"gray50"’, ‘"grey50"’, ‘"gray60"’, ‘"grey60"’, ‘"gray70"’,
‘"grey70"’, ‘"gray80"’, ‘"grey80"’, ‘"gray90"’, ‘"grey90"’,
‘"gray100"’, ‘"grey100"’, ‘"gray"’, ‘"grey"’, ‘"light-gray"’,
‘"light-grey"’, ‘"dark-gray"’, ‘"dark-grey"’, ‘"red"’,
‘"light-red"’, ‘"dark-red"’, ‘"yellow"’, ‘"light-yellow"’,
‘"dark-yellow"’, ‘"green"’, ‘"light-green"’, ‘"dark-green"’,
‘"spring-green"’, ‘"forest-green"’, ‘"sea-green"’, ‘"blue"’,
‘"light-blue"’, ‘"dark-blue"’, ‘"midnight-blue"’, ‘"navy"’,
‘"medium-blue"’, ‘"royalblue"’, ‘"skyblue"’, ‘"cyan"’,
```

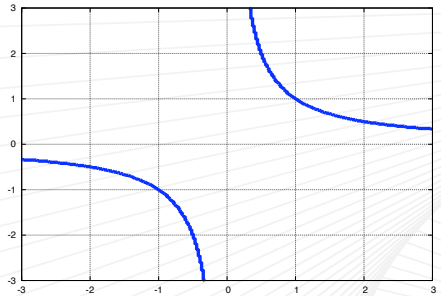
```
'"light-cyan"', '"dark-cyan"', '"magenta"', '"light-magenta"',
'"dark-magenta"', '"turquoise"', '"light-turquoise"',
'"dark-turquoise"', '"pink"', '"light-pink"', '"dark-pink"',
'"coral"', '"light-coral"', '"orange-red"', '"salmon"',
'"light-salmon"', '"dark-salmon"', '"aquamarine"', '"khaki"',
'"dark-khaki"', '"goldenrod"', '"light-goldenrod"',
'"dark-goldenrod"', '"gold"', '"beige"', '"brown"', '"orange"',
'"dark-orange"', '"violet"', '"dark-violet"', '"plum"' y
'"purple"'.
```

Ya lo hemos usado en casi todos los ejemplos anteriores.

`line_width`: grosor con el que se dibujan las líneas. Su valor por defecto es 1.

```
(%i73)   draw2d(
           color=blue,line_width=10,nticks=100,
           implicit(x*y=1,x,-3,3,y,-3,3),
           grid=true,
           )$
```

(%o73)

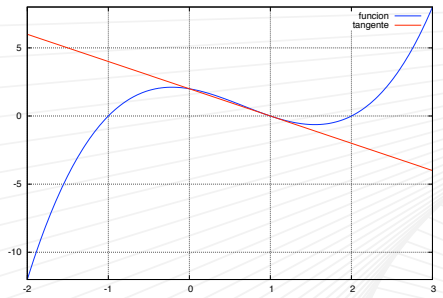


`nticks`: número de puntos que se utilizan para calcular los dibujos. Por defecto es 30. Un número mayor aumenta el detalle del dibujo aunque a costa de un mayor tiempo de cálculo y tamaño del fichero (si se guarda). Sólo afecta a los objetos de tipo `ellipse`, `explicit`, `parametric`, `polar` y `parametric`.

`key`: indica la leyenda con la que se identifica la función.

```
(%i74)   draw2d(color=blue,key="función",explicit(f(x),x,-2,3),
           color=red,key="tangente",explicit(tangente(x,1),x,-2,3),
           grid=true);
```

(%o74)

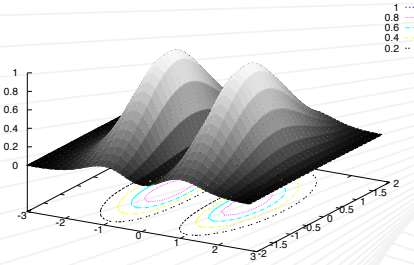


`surface_hide`: cuando vale `true` no se dibuja la parte no visible de las superficies.

`contour` nos permite dibujar o no las curvas de nivel de una superficie. Por defecto no se muestran (`none`) pero también podemos dibujarlas sobre el plano XY con `base`

```
(%i75)  draw3d(
          enhanced3d=true,palette=gray,
          colorbox=false,surface_hide=true,contour=base,
          explicit(x^2*exp(1-x^2-0.5*y^2),x,-3,3,y,-2,2));
```

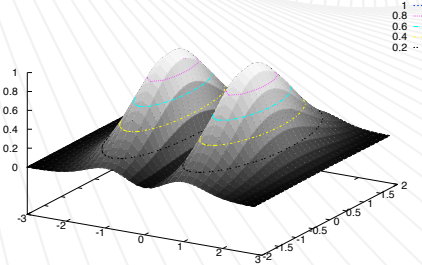
(%o75)



sobre la propia superficie con `surface`

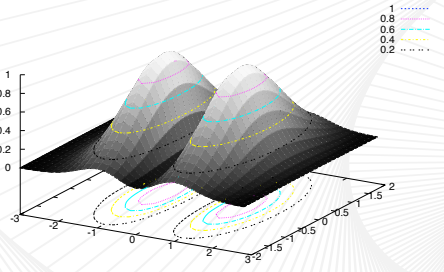
```
(%i76)   draw3d(
           enhanced3d=true,palette=gray,
           colorbox=false,surface_hide=true,contour=surface,
           explicit(x^2*exp(1-x^2-0.5*y^2),x,-3,3,y,-2,2));
```

(%o76)



las dos posibilidades anteriores se pueden usar a la vez con `both`

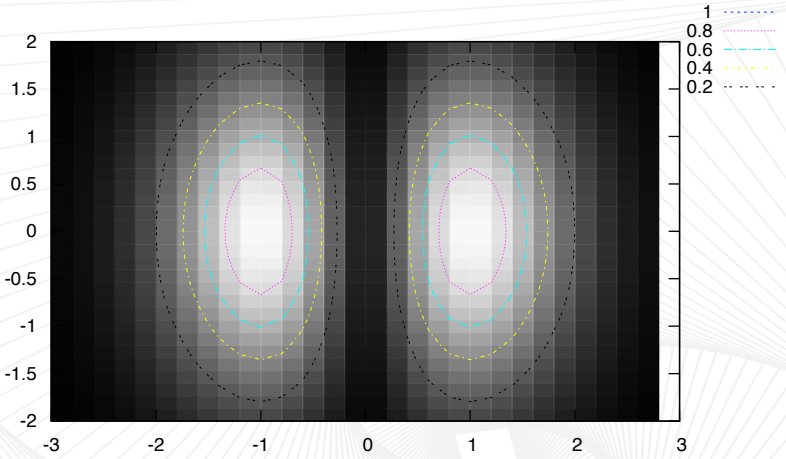
```
(%i77)   draw3d(
           enhanced3d=true,
           palette=gray, colorbox=false,
           surface_hide=true, contour=both,
           explicit(x^2*exp(1-x^2-0.5*y^2),x,-3,3,y,-2,2));
(%o77)
```



o, simplemente, podemos dibujar las curvas de nivel vistas desde arriba con `map`

```
(%i78)   draw3d(
           enhanced3d=true,
           palette=gray,
           colorbox=false,
           surface_hide=true,
           contour=map,
           explicit(x^2*exp(1-x^2-0.5*y^2),x,-3,3,y,-2,2));
```

(%o78)

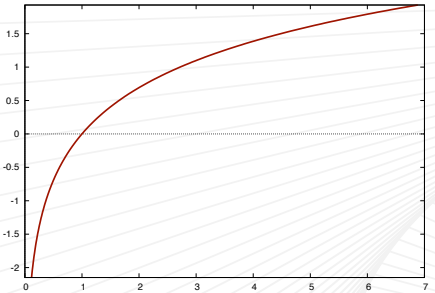


2.4.2 Objetos

`explicit`: nos permite dibujar una función de una o dos variables. Para funciones de una variable usaremos `explicit(`$f(x)$`,`x`,`a`,`b`)` para dibujar $f(x)$ en $[a, b]$. Con funciones de dos variables escribiremos `explicit(`$f(x, y)$`,`x`,`a`,`b`,`y

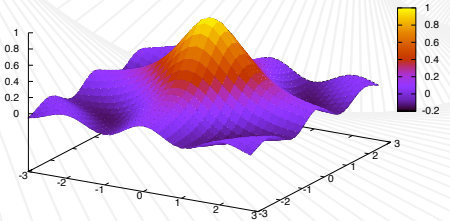
```
(%i79)    draw2d(
            color=dark-red,line_width=5,
            xaxis=true,yaxis=true,
            explicit(log(x),x,0,7) );
```

(%o79)



(%i80)
```
draw3d(enhanced3d=true,
  surface_hide=true,
  explicit((1+x^2+y^2)^(-1)*cos(x*y),x,-3,3,y,-3,3));
```

(%o80)



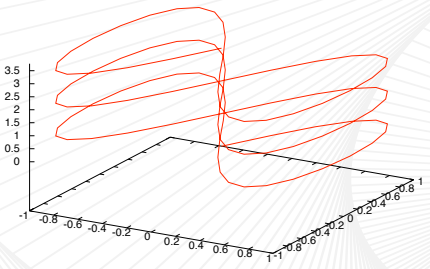
`parametric`: nos permite dibujar una curva en dos o en tres dimensiones. En coordenadas paramétricas, tenemos que dar las dos o tres coordenadas de la forma

$$\texttt{parametric}(f_1(x),f_2(x),x,a,b)$$

$$\texttt{parametric}(f_1(x),f_2(x),f_3(x),x,a,b)$$

Por ejemplo, en tres dimensiones la curva $t \mapsto (\text{sen}(t), \text{sen}(2t), t/5)$ se representa de la siguiente forma.

```
(%i81)  draw3d(color=red,
          nticks=100,
          parametric(sin(t),sin(2*t),t/5,t,0,6*%pi));
(%o81)
```

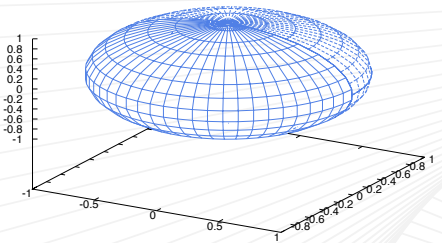


`parametric_surface`: nos permite dibujar una superficie en paramétricas. Tenemos pues que dar las tres coordenadas así que escribimos

`parametric_surface`$(f_1(x,y), f_2(x,y), f_3(x,y), x, a, b, y, c, d)$

```
(%i82)  draw3d(
          color=royalblue,
          surface_hide=true,
          parametric_surface(cos(u)*cos(v),cos(v)*sin(u),sin(v),
            u,0,%pi,v,0,2*%pi));
```

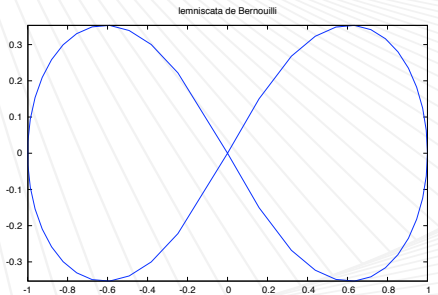
(%o82)



`polar`: dibuja el módulo del vector en función del ángulo t

```
(%i83)  draw2d(
          color=blue,
          nticks=100,
          title="lemniscata de Bernouilli",
          polar(sqrt(cos(2*t)),t,0,2*%pi));
```

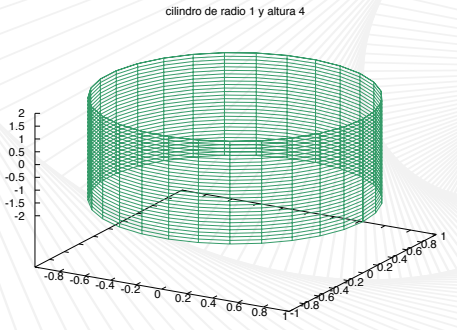
(%o83)



`cylindrical`: para dar un vector del espacio en coordenadas cilíndricas lo que hacemos es cambiar a coordenadas polares en las dos primeras variables (aunque tendría perfecto sentido hacerlo con cualquier par de ellas). La figura cuya representación es más sencilla en coordenadas cilíndricas te la puedes imaginar.

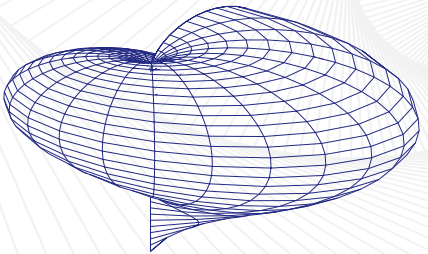
```
(%i84)  draw3d(
          color=sea-green,
          title="cilindro de radio 1 y altura 4",
          cylindrical(1,z,-2,2,t,0,2*%pi));
```

(%o84)



`spherical`: en coordenadas esféricas la construcción es muy parecida. Representamos el módulo y damos dónde varían los dos ángulos.

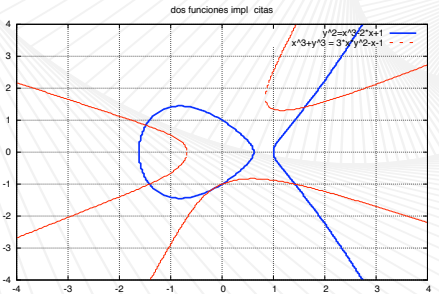
```
(%i85)   draw3d(
           color=midnight-blue,
           surface_hide = true,
           axis_3d=false,
           xtics=none,
           ytics=none,
           ztics=none,
           spherical(a*z,a,0,2.5*%pi,z,0,%pi));
(%o85)
```



`implicit`: nos permite dibujar el lugar de los puntos que verifican una ecuación en el plano

```
(%i86)  draw2d(
          grid=true,
          line_type=solid,
          color=blue,
          key="y^2=x^3-2x+1",
          implicit(y^2=x^3-2*x+1, x, -4,4, y, -4,4),
          line_type=dots,
          color=red,
          key="x^3+y^3 = 3xy^2-x-1",
          implicit(x^3+y^3 = 3*x*y^2-x-1, x,-4,4, y,-4,4),
          title="dos funciones implícitas");
```

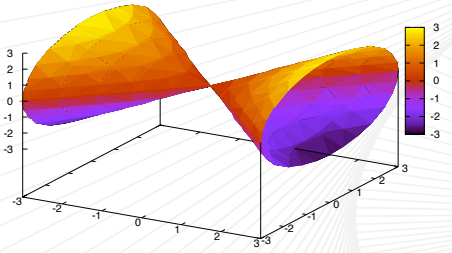
(%o86)



o en tres dimensiones. Además de la ecuación debemos indicar los intervalos dónde pueden tomar valores las variables.

```
(%i87)  draw3d(
          surface_hide=true,
          enhanced3d=true,
          implicit(x^2-y^2=z^2,x,-3,3,y,-3,3,z,-3,3));
```

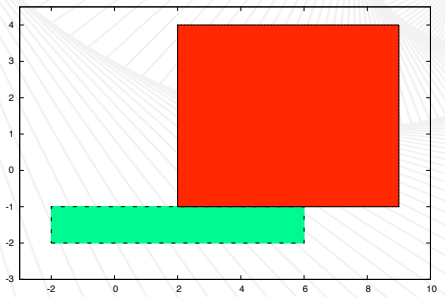
(%o87)



`rectangle`: para dibujar un rectángulo sólo tenemos que indicar el vértice inferior izquierdo y su opuesto.

```
(%i88)  draw2d(line_width=6,
          line_type=dots,
          transparent=false,
          fill_color=spring-green,
          rectangle([-2,-2],[6,-1]),
          transparent=false,
          fill_color=red,
          line_type=solid,
          line_width=2,
          rectangle([9,4],[2,-1]),
          xrange=[-3,10],
          yrange=[-3,4.5]);
```

(%o88)

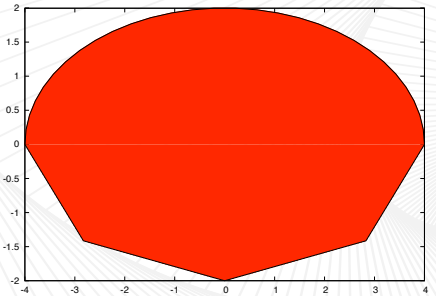


`ellipse`: la orden `ellipse` permite dibujar elipses indicando 3 pares de números: los dos primeros son las coordenadas del centro, los dos segundos indican la longitud de los semiejes y los últimos son los ángulos inicial y final.

En el dibujo siguiente puedes comprobar cómo la opción `nticks` permite mejorar, aquí empeorar, un gráfico aumentando o, como en este caso, disminuyendo el número de puntos que se utilizan para dibujarlo.

```
(%i89)  draw2d(
          ellipse(0,0,4,2,0,180),
          nticks = 5,
          ellipse(0,0,4,2,180,360));
```

(%o89)



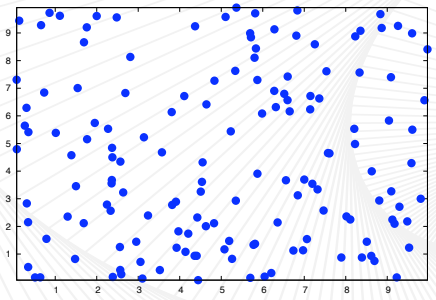
La parte superior de la elipse se ha dibujado utilizando 30 puntos y la inferior únicamente 5.

`points`: para representar una lista de puntos en el plano o en el espacio tenemos dos posibilidades. Podemos dar los vectores de la forma[6] `[[x1,y1],[x2,y2],...]`, como por ejemplo

[6] En el ejemplo usaremos la orden `makelist` que genera una lista de acuerdo a la regla que aparece como primera entrada con tantos elementos como indique el contador que le sigue. En el próximo capítulo lo comentaremos con más detalle.

```
(%i90)   draw2d(
           color=blue,
           point_type=filled_circle,
           point_size=2,
           points(makelist([random(10.0),random(10.0)],k,1,150)));
```

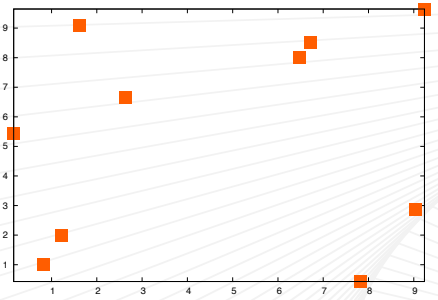
(%o90)



o podemos agrupar por coordenadas de la forma `[[x1,x2,x3,...],[y1,y2,y3,...]]` como aquí.

```
(%i91)   draw2d(
           color=orange-red,
           point_type=5,
           point_size=3,
           points(makelist(random(10.0),k,1,10),
           makelist(random(10.0),k,1,10)));
```

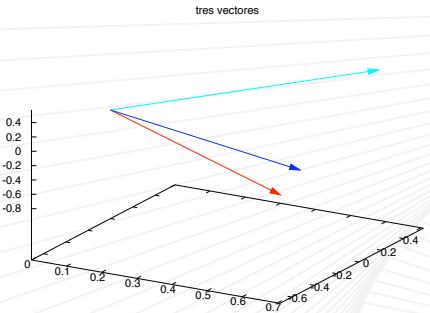
(%o91)



`vector`: dibuja vectores tanto en dos como en tres dimensiones. Para dar un vector hay que fijar el origen y la dirección.

```
(%i92)   draw3d(color = cyan,
           vector([0,0,0],[1,1,1]/sqrt(3)),
           color=red,vector([0,0,0],[1,-1,0]/sqrt(2)),
           color=blue,vector([0,0,0],[1,1,-2]/sqrt(6)),
           title="tres vectores");
```

(%o92)



En la ayuda puedes encontrar varios opciones sobre el aspecto como se representan los vectores.

2.5 Animaciones gráficas

Con *wxMaxima* es muy fácil hacer animaciones gráficas que dependen de un parámetro. Por ejemplo, la función $\text{sen}(x+n)$ depende del parámetro n. Podemos representar su gráfica para distintos valores de n y con ello logramos una buena visualización de su evolución (que en este caso será una onda que se desplaza). Para que una animación tenga calidad es necesario que todos los gráficos individuales tengan el mismo tamaño y que no "den saltos" para lo que elegimos un intervalo del eje de ordenadas común.

Para ver la animación, cuando se hayan representado las gráficas, haz clic con el ratón sobre ella y desplaza la barra (slider) que tienes bajo el menú. De esta forma tú mismo puedes controlar el sentido de la animación , así como la velocidad.

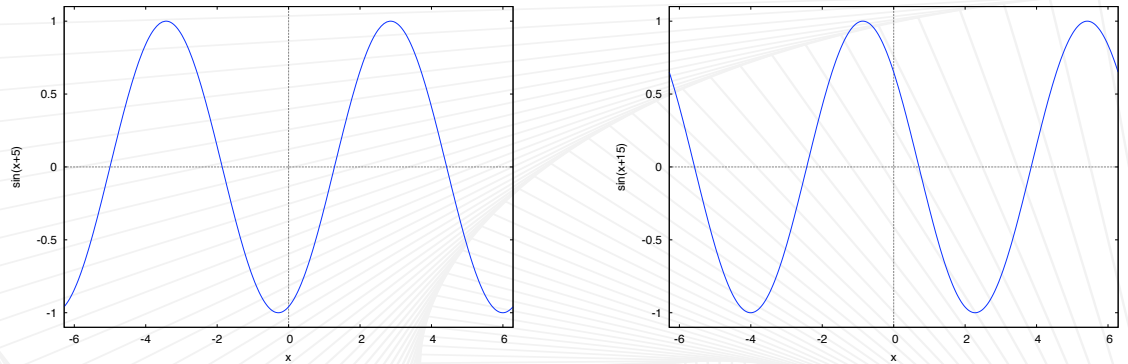
`with_slider`	animación de `plot2d`
`with_slider_draw`	animación de `draw2d`
`with_slider_draw3d`	animación de `draw3d`

Tenemos tres posibilidades para construir animaciones dependiendo de si queremos que *Maxima* utilice `plot2d`, `draw2d` o `draw3d`. En cualquier caso, en primer lugar siempre empezamos con el parámetro, una lista de valores del parámetro y el resto debe ser algo aceptable por el correspondiente comando con el que vayamos a dibujar.

Por ejemplo, vamos a crear una animación con la orden `with_slider` de la función $\text{sen}(x+n)$, donde el parámetro n va a tomar los valores desde 1 a 20. La orden `makelist(i,i,1,20)` nos da todos los números naturales comprendidos entre 1 y 20. Ya veremos con más detalle en el Capítulo 3 cómo podemos manejar listas.

```
(%i93)   with_slider(n,makelist(i,i,1,20),sin(x+n),[x,-2*%pi,2*%pi],[y,-1.1,1.1]);
```

En la Figura 2.5 tienes la representación de algunos valores



$n = 5$ $n = 15$

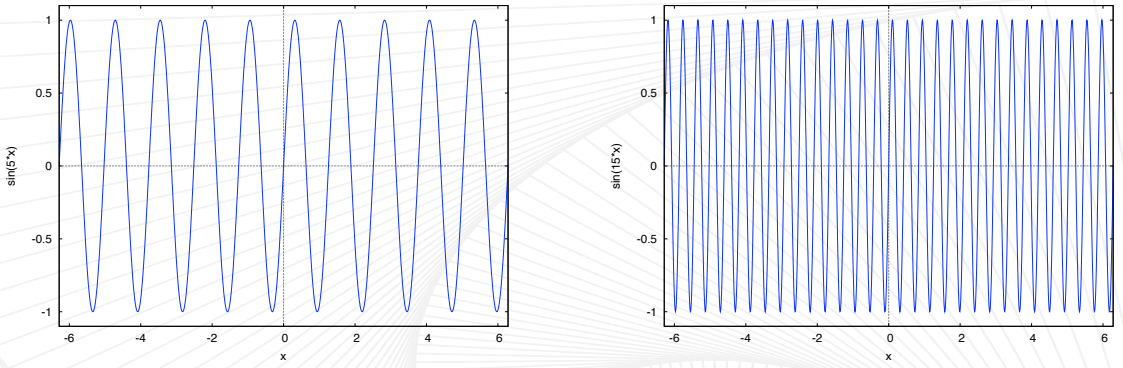
Figura 2.5 $\operatorname{sen}(x + n)$

Si en lugar de sumar el parámetro a la variable (que traslada la función), multiplicamos el parámetro y la variable conseguimos cambiar la frecuencia de la onda que estamos dibujando.

```
(%i94)   with_slider(n,makelist(i,i,1,20),sin(x*n),
           [x,-2*%pi,2*%pi],[y,-1.1,1.1]);
```

Puedes ver en la Figura 2.6 puedes ver cómo aumenta la frecuencia con n.

Si en lugar de `plot2d`, utilizamos el módulo `draw` para diseñar los dibujos, tenemos que usar `with_slider_draw` o `with_slider_draw3d`. De nuevo, en primer lugar va el parámetro, después, una lista que indica los valores que tomará el parámetro y el resto debe ser algo aceptable por la orden `draw` o `draw3d`, respectivamente. Un detalle importante en este caso es que el parámetro no sólo puede afectar a la función sino que podemos utilizarlo



$n = 5$ $n = 15$

Figura 2.6 $\operatorname{sen}(n\,x)$

en cualquier otra parte de la expresión. Por ejemplo, podemos utilizar esto para ir dibujando poco a poco una circunferencia en coordenadas paramétricas de la siguiente forma

```
(%i95)   with_slider_draw(
           t,makelist(%pi*i/10,i,1,20),
           parametric(cos(x),sin(x),x,0,t),
           xrange=[-1,1],
           yrange=[-1,1],
           user_preamble="set size ratio 1")$
```

En la Figura 2.7 tenemos representados algunos pasos intermedios

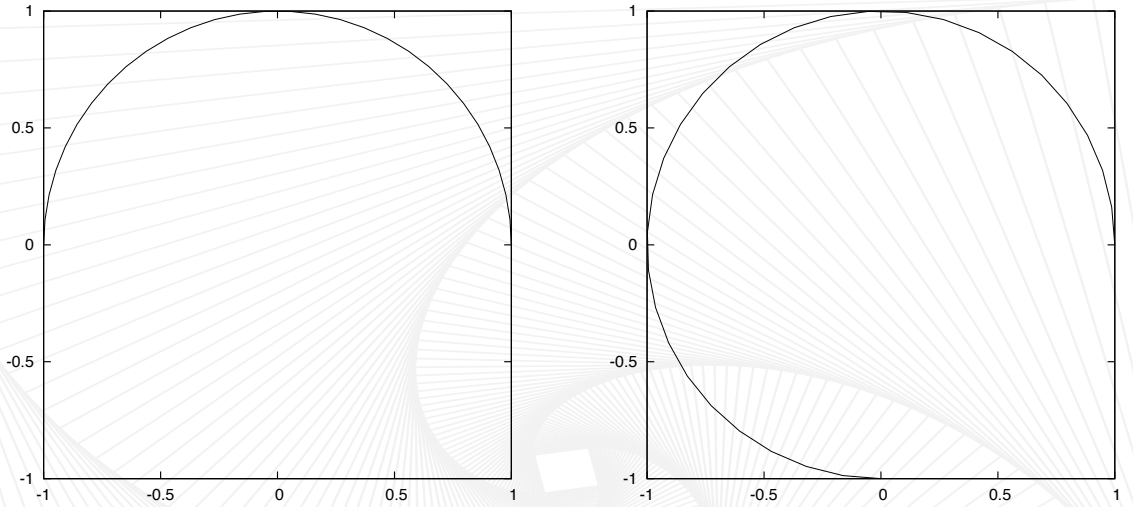


Figura 2.7 Construcción de una circunferencia en paramétricas

Por último, veamos algunos ejemplos de las posibilidades en tres dimensiones. Te recuerdo que el parámetro puede aparecer en cualquier posición. Podemos utilizarlo para indicar el ángulo de rotación y conseguir "dar la vuelta" a la superfice.

```
(%i96)   with_slider_draw3d(
           k,makelist(i*36,i,1,10),
           parametric_surface(cos(u)+.5*cos(u)*cos(v),
             sin(u)+.5*sin(u)*cos(v),
             .5*sin(v),
             u, -%pi, %pi, v, -%pi, %pi),
           parametric_surface(1+cos(u)+.5*cos(u)*cos(v),
             .5*sin(v),
             sin(u)+.5*sin(u)*cos(v),
             u, -%pi, %pi, v, -%pi, %pi),
           surface_hide=true,rot_horizontal=k
           )$
```

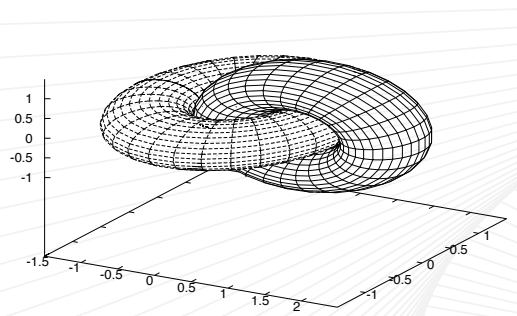
De nuevo, aquí representamos en la Figura 2.8 algunos pasos intermedios

También se puede utilizar de la forma "clásica" para representar una función que depende de un parámetro.

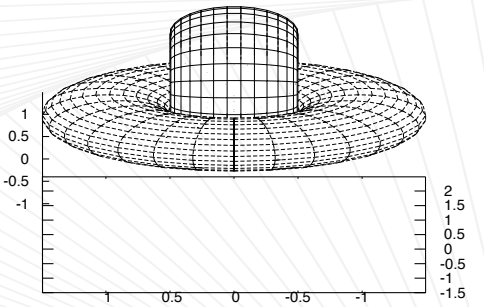
```
(%i97)   with_slider_draw3d(
           k,makelist(i,i,-4,4),
           explicit((x^2-k*y^2)*exp(1-x^2-y^2),x,-2,2,y,-2,2),
           surface_hide=true
         )$
```

como puedes ver en la Figura 2.9.

En el último ejemplo podemos ver cómo se pueden combinar funciones definidas explícita e implícitamente juntos con vectores para obtener una representación de las funciones seno y coseno.

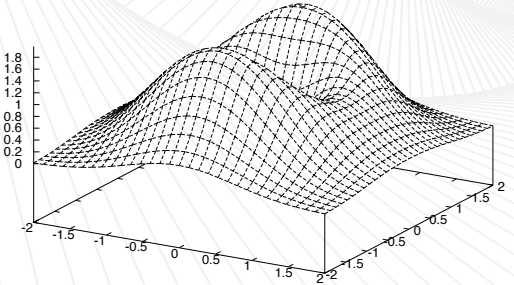


rotacion 0°

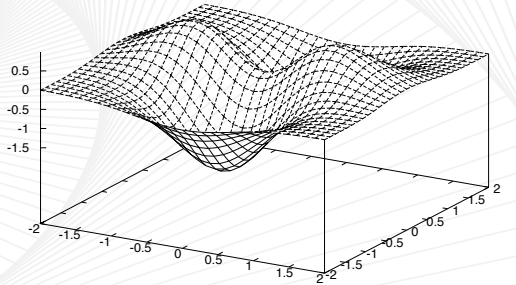


rotación 270°

Figura 2.8 Giro alrededor de una superficie en paramétricas



$k = -2$



$k = 2$

Figura 2.9 Función $f(x, y) = (x^2 - ky^2)e^{1-x^2-y^2}$

```
(%i98)   with_slider_draw(t,makelist(2*%pi*i/39,i,1,40),
           line_width=3, color=blue,
           parametric(cos(x),sin(x),x,0,t),
           color=light-red, key="seno",
           explicit(sin(x),x,0,t),
           color=dark-red, key="coseno",
           explicit(cos(x),x,0,t),
           line_type=dots, head_length=0.1,
           color=dark-red, key="",
           vector([0,0],[cos(t),0]),
           color=light-red, line_type=dots,
           head_length=0.1, key="",
           vector([0,0],[0,sin(t)]),
           line_type=dots, head_length=0.1, key="",
           vector([0,0],[cos(t),sin(t)]),
           xaxis=true,yaxis=true,
           title="Funciones seno y coseno",
           xrange=[-1,2*%pi],yrange=[-1,1]);
```

Para $t = 5$, el resultado lo puedes ver en la Figura 2.10

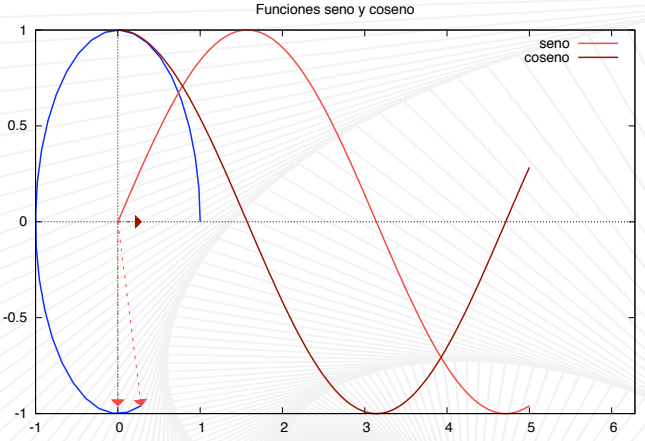


Figura 2.10 Las funciones seno y coseno

2.6 Ejercicios

Ejercicio 2.1. Representa en una misma gráfica las funciones seno y coseno en el intervalo $[-2\pi, 2\pi]$. Utiliza las opciones adecuadas para que una de las funciones se represente en azul y otra en rojo y, además, tengan grosores distintos.

Ejercicio 2.2. Compara las gráficas de las funciones $\cos(x)$ y $\cos(-x)$. ¿A qué conclusión llegas sobre la paridad o imparidad de la función coseno? Haz lo mismo con las funciones $\operatorname{sen}(x)$ y $\operatorname{sen}(-x)$.

Ejercicio 2.3. Representa las funciones logaritmo neperiano, exponencial y $f(x) = x^2$ con colores diferentes. Compara el crecimiento de estas funciones cerca de cero y lejos de cero. ¿Qué ocurre si la base de la exponencial y del logaritmo es menor que 1?

Ejercicio 2.4. Dibuja las gráficas de las funciones coseno hiperbólico, seno hiperbólico, argumento seno hiperbólico y argumento coseno hiperbólico. ¿Alguna de ellas es par o impar? ¿Son positivas?

Ejercicio 2.5. Representa la curva $\cos(x)^2 - x\,\operatorname{sen}(x)^2$ en el intervalo $[-\pi, \pi]$ y sobre ella 5 puntos cuyo tamaño y color debes elegir tú. ¿Sabrías hacer lo mismo con 8 puntos elegidos aleatoriamente?[7]

Ejercicio 2.6. Representa la gráfica de la función $f : \mathbb{R}_0^+ \to \mathbb{R}$ definida como

$$f(x) = \begin{cases} e^{3x+1}, & \text{si } 0 \leq x < 10, \\ \ln\left(x^2+1\right), & \text{si } x \geq 10. \end{cases}$$

Ejercicio 2.7. Dibuja un triángulo y colorea los vértices en rojo, verde y azul. Une con segmentos los puntos medios de cada lado del triángulo para dibujar otro triángulo.

[7] En el siguiente capítulo puedes encontrar una explicación más detallada sobre como definir y operar con listas.

Ejercicio 2.8. Representar las siguientes curvas y superficies dadas en forma polar o paramétrica.

a) $\gamma(\theta) = 2 + \cos(5\theta)$, $\forall\theta \in [0, 2\pi]$.
b) $\gamma(\theta) = 8\,\mathrm{sen}(5/2\theta)$, $\forall\theta \in [0, 2\pi]$.
c) $\gamma(t) = (\mathrm{sen}(t), \mathrm{sen}(2t), t/5)$, $t \in [0, 5]$,
d) $\gamma(t) = 2(\cos(3t), \mathrm{sen}(5t))$, $t \in [0, 2\pi]$.
e) $\gamma(u, v) = (\mathrm{sen}(u), \mathrm{sen}(v), v)$ con $u \in [-\pi, \pi]$, $v \in [0, 5]$.
f) $\gamma(u, v) = (u\cos(v)\,\mathrm{sen}(u), u\cos(u)\cos(v), -u\,\mathrm{sen}(v))$, con u, $v \in [0, 2\pi]$.

Ejercicio 2.9. Representa una elipse de semiejes 2 y 4. Inscribe en ella un rectángulo y, dentro del rectángulo, una circunferencia. Dibuja la elipse en azul, el rectángulo en verde y la circunferencia en rojo.

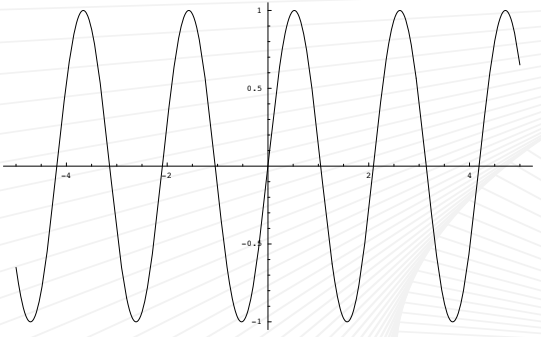
Ejercicio 2.10. Realiza una animación gráfica para representar la curva

$$t \mapsto (\mathrm{sen}(t), \cos(4t) + 0.2)\,(1 - s) + s\,(\cos(t), \mathrm{sen}(t))\,,$$

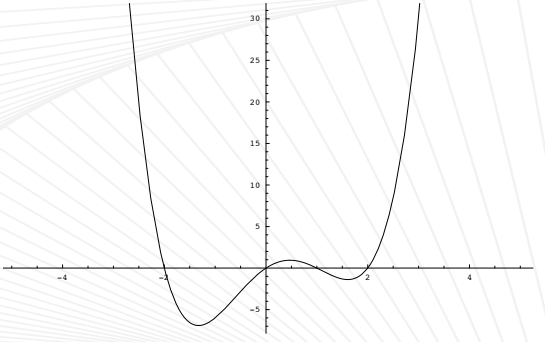
con $t \in [0, 2\pi]$ y donde el parámetro s toma los valores de 0 a 1 con incrementos de 0.1.

Ejercicio 2.11. Realiza una animación gráfica que represente la cicloide.

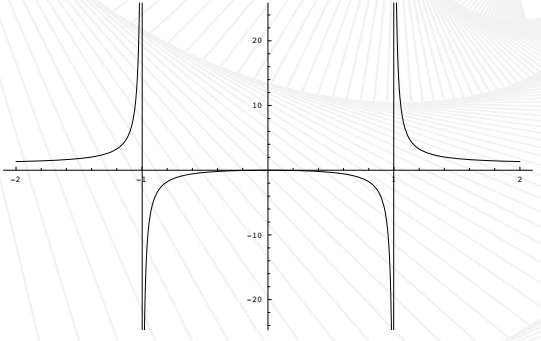
Ejercicio 2.12. Encuentra las funciones cuyas gráficas corresponden a las siguientes curvas:


1
0.5
-0.5
-1
-4
-2
2
4

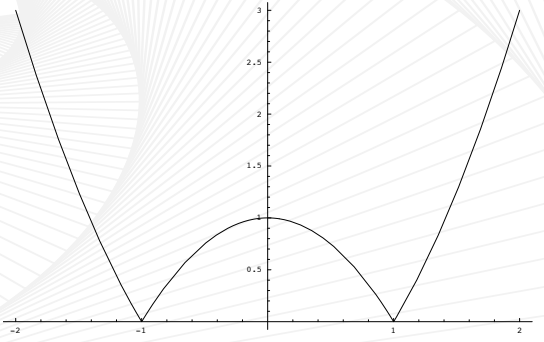
(a)


30
25
20
15
10
5
-5
-4
-2
2
4

(b)


20
10
-10
-20
-2
-1
1
2

(c)


3
2.5
2
1.5
1
0.5
-2
-1
1
2

(d)

3 Listas y matrices

3.1 Listas

Maxima tiene una manera fácil de agrupar objetos, ya sean números, funciones, cadenas de texto, etc. y poder operar con ellos. Una lista se escribe agrupando entre corchetes los objetos que queramos separados por comas. Por ejemplo,

```
(%i1)   [0,1,-3];
(%o1)   [0,1,-3]
```

es una lista de números. También podemos escribir listas de funciones

```
(%i2)   [x,x^2,x^3]
(%o2)   [x,x^2,x^3]
```

o mezclar números, variables y texto

```
(%i3)   [0,1,-3,a,"hola"];
(%o3)   [0,1,-3,a,hola]
```

`first, second,...,tenth`	primera, segunda,...,décima entrada de una lista
`lista[i]`	entrada i-ésima de la lista
`last`	último elemento de una lista
`part`	busca un elemento dando su posición en la lista
`reverse`	invertir lista
`sort`	ordenar lista
`flatten`	unifica las sublistas en una lista
`length`	longitud de la lista
`unique`	elementos que sólo aparecen una vez en la lista

Los elementos que forman la lista pueden ser, a su vez, listas (aunque no es exactamente lo mismo, piensa en matrices como "listas de vectores"):

```
(%i4)   lista:[[1,2],1,[3,a,1]]
(%o4)   [[1,2],1,[3,a,1]]
```

Podemos referirnos a una entrada concreta de una lista. De hecho *Maxima* tiene puesto nombre a las diez primeras: `first, second,..., tenth`

```
(%i5)   first(lista);
```

```
(%o5)    [1,2]
(%i6)    second(lista);
(%o6)    1
```

o podemos referirnos directamente al último término.

```
(%i7)    last(lista);
(%o7)    [3,a,1]
```

Si sabemos la posición que ocupa, podemos refererirnos a un elemento de la lista utilizando `part`. Por ejemplo,

```
(%i8)    part(lista,1)
(%o8)    [1,2]
```

nos da el primer elemento de la lista anterior. Obtenemos el mismo resultado indicando la posición entre corchetes. Por ejemplo,

```
(%i9)    lista[3];
(%o9)    [3,a,1]
```

y también podemos anidar esta operación para obtener elementos de una sublista

```
```

```
(%i10)   lista[3][1];
(%o10)   3
```

Con `part` podemos extraer varios elementos de la lista enumerando sus posiciones. Por ejemplo, el primer y el tercer elemento de la lista son

```
(%i11)   part(lista,[1,3]);
(%o11)   [[1,2],[3,a,1]]
```

o el segundo término del tercero que era a su vez una lista:

```
(%i12)   part(lista,3,2);
(%o12)   a
```

El comando `flatten` construye una única lista con todas los elementos, sean estos listas o no. Mejor un ejemplo:

```
(%i13)   flatten([[1,2],1,[3,a,1]])
(%o13)   [1,2,1,3,a,1]
```

La lista que hemos obtenido contiene todos los anteriores. Podemos eliminar los repetidos con `unique`

```
(%i14)   unique(%)
```

```
(%o14)    [1,2,3,a]
```

Vectores

En el caso de vectores, listas de números, tenemos algunas posibilidades más. Podemos sumarlos

```
(%i15)    v1:[1,0,-1];v2:[-2,1,3];
(%o15)    [1,0,-1]
(%o16)    [-2,1,3]
(%i17)    v1+v2;
(%o17)    [-1,1,2]
```

o multiplicarlos.

```
(%i18)    v1*v2;
(%o18)    [-2,0,-3]
```

Un momento, ¿cómo los hemos multiplicado? Término a término. Esto no tiene nada que ver con el producto escalar o con el producto vectorial. El producto escalar, por ejemplo, se indica con "."

```
(%i19)    v1.v2;
(%o19)    -5
```

Podemos ordenar los elementos de la lista (del vector en este caso)

```
(%i20)   sort(v1);
(%o20)   [-1,0,1]
```

o saber cuántos elementos tiene

```
(%i21)   length(v1);
(%o21)   3
```

3.1.1 Construir y operar con listas

`makelist`	genera lista
`apply`	aplicar un operador a una lista
`map`	aplicar una función a una lista
`listp(`*expr*`)`	devuelve `true` si la expresión es una lista

Los ejemplos que hemos visto de listas hasta ahora son mezcla de números y letras de forma bastante aleatoria. En la práctica, muchas de las listas que aparecen están definidas por alguna regla. Por ejemplo, queremos dibujar las funciones $\mathrm{sen}(x)$, $\mathrm{sen}(2x)$,..., $\mathrm{sen}(20x)$. Seguro que no tienes ganas de escribir la lista completa. Este es el papel de la orden `makelist`. Para escribir esa lista necesitamos la regla, la fórmula que la define, un parámetro y entre qué dos valores se mueve dicho parámetro:

```
(%i22)  makelist(sin(t*x),t,1,20)
(%o22)  [sin(x),sin(2 x),sin(3 x),sin(4 x),sin(5 x),sin(6 x),
        sin(7 x),sin(8 x),sin(9 x),sin(10 x),sin(11 x),
        sin(12 x),sin(13 x),
        sin(14 x),sin(15 x),
        sin(16 x),sin(17 x),sin(18 x),sin(19 x),sin(20 x)]
```

Las listas también se pueden utilizar como contadores. El caso que suele ser más útil es una lista cuyas entradas sean un rango de enteros. Por ejemplo, los primeros cien naturales empezamos en uno) son

```
(%i23)  makelist(i,i,1,100);
(%o23)  [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
        23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,
        42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,
        61,62,63,64,65,66,67, 68,69,70,71,72,73,74,75,76,77,78,79,
        80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,
        99,100]
```

o si sólo queremos los pares:

```
(%i24)  makelist(2*i,i,1,50);
(%o24)  [2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,
        42,44,46,48,50,52,54,56,58,60,62,64,66,68,70,72,74,76,78,
        80,82,84,86,88,90,92,94,96,98,100]
```

Ya que tenemos una lista, ¿cómo podemos "jugar" con sus elementos? Por ejemplo, ¿se puede calcular el cuadrado de los 100 primeros naturales? ¿Y su media aritmética o su media geométrica? Las órdenes `map` y `apply` nos ayudan a resolver este problema. La orden `map` permite aplicar una función a cada uno de los elementos de una lista. Por ejemplo, para calcular $\mathrm{sen}(1)$, $\mathrm{sen}(2)$,..., $\mathrm{sen}(10)$, hacemos lo siguiente

```
(%i25)  map(sin,makelist(i,i,1,10));
(%o25)  [sin(1),sin(2),sin(3),sin(4),sin(5),sin(6),sin(7),sin(8),
        sin(9),sin(10)]
```

o si queremos la expresión decimal

```
(%i26)  %,numer
(%o26)  [0.8414709848079,0.90929742682568,0.14112000805987,
        -0.75680249530793,-0.95892427466314,-0.27941549819893,
        0.65698659871879,0.98935824662338,0.41211848524176,
        -0.54402111088937]
```

La orden `apply`, en cambio, pasa todos los valores de la lista a un operador que, evidentemente, debe saber qué hacer con la lista. Ejemplos típicos son el operador suma o multiplicación. Por ejemplo

```
(%i27)  apply("+",makelist(i,i,1,100));
(%o27)  5050
```

nos da la suma de los primeros 100 naturales.

Ejemplo 3.1. Vamos a calcular la media aritmética y la media geométrica de los 100 primeros naturales. ¿Cuál será mayor? ¿Recuerdas la desigualdad entre ambas medias? La media aritmética es la suma de todos los elementos dividido por la cantidad de elementos que sumemos:

```
(%i28)  apply("+",makelist(i,i,1,100))/100;
(%o28)  101/2
```

La media geométrica es la raíz n-ésima del producto de los n elementos:

```
(%i29)  apply("*",makelist(i,i,1,100))^(1/100);
(%o29)  17^(1/20) 19^(1/20) 23^(1/25) 37^(1/50) 41^(1/50) 43^(1/50) 47^(1/50) 2401^(1/25) 15625^(1/25) 531441^(1/25)
        638714741182055044530[30digits]99766363898994157944832^(1/100)
(%i30)  float(%);
(%o30)  37.9926893448343
```

Parece que la media geométrica es menor. ◁

Ejemplo 3.2. ¿Cuál es el módulo del vector $(1, 3, -7, 8, 1)$? Tenemos que calcular la raíz cuadrada de la suma de sus coordenadas al cuadrado:

```
(%i31)  vector:[1,3,-7,8,1];
(%o31)  [1,3,-7,8,1]
(%i32)  sqrt(apply("+",vector^2));
```

(%o32) $2\sqrt{31}$

A la vista de estos dos ejemplos, ¿cómo podríamos definir una función que nos devuelva la media aritmética, la media geométrica de una lista o el módulo de un vector? ◁

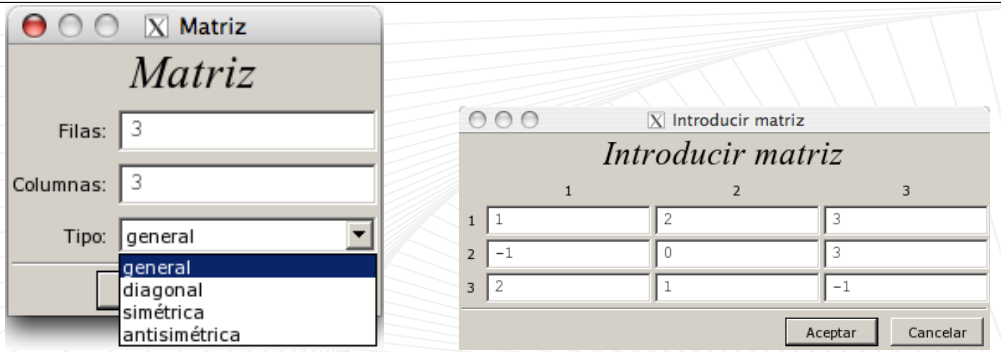
3.2 Matrices

Las matrices se escriben de forma parecida a las listas y, de hecho, sólo tenemos que agrupar las filas de la matriz escritas como listas bajo la orden `matrix`. Vamos a definir un par de matrices y un par de vectores que van a servir en los ejemplos en lo que sigue.

```
(%i33)  A:matrix([1,2,3],[-1,0,3],[2,1,-1]);
        B:matrix([-1,1,1],[1,0,0],[-3,7,2]);
        a:[1,2,1];
        b:[0,1,-1];
(%o34)  [ 1   2   3 ]
        [-1   0   3 ]
        [ 2   1  -1 ]
(%o35)  [-1   1   1 ]
        [ 1   0   0 ]
        [-3   7   2 ]
(%o36)  [1,2,1]
(%o36)  [0,-1,1]
```

En *wxMaxima* también podemos escribir una matriz usando el menú **Álgebra→Introducir matriz**. Nos aparece una ventana como las de la Figura 3.1 donde podemos rellenar los valores.

Las dimensiones de una matriz se pueden recuperar mediante la orden `matrix_size` que devuelve una lista con el número de filas y columnas.



Seleccionar tipo de matriz

Introducir valores

Figura 3.1 Introducir matriz

`matrix(`*`fila1,fila2,...`*`)`	matriz
`matrix_size(`*`matriz`*`)`	número de filas y columnas
`matrixp(`*`expresión`*`)`	devuelve `true` si *expresión* es una matriz

```
(%i37)   matrix_size(A);
(%o37)   [3,3]
```

Observación 3.3. Aunque muy similares, *Maxima* distingue entre listas y matrices. La orden `matrixp(`*`expresión`*`)` devuelve `true` o `false` dependiendo de si la expresión es o no una matriz. Por ejemplo, los vectores a y b que hemos definido antes, ¿son o no son matrices?

```
(%i38)   matrixp(a);
(%o38)   false
```

Aunque pueda parecer lo contrario, no son matrices, son listas.

```
(%i39)   listp(a);
(%o39)   true
```

Sólo es aceptado como matriz aquello que hallamos definido como matriz mediante la orden `matrix` o alguna de sus variantes. Al menos en *wxMaxima*, hay un pequeño truco para ver si algo es o no una matriz. ¿Cuál es la diferencia entre las dos siguientes salidas?

```
(%i40)   [1,2,3];
(%o40)   [1,2,3]
(%i41)   matrix([1,2,3]);
(%o41)   [1   2   3]
```

wxMaxima respeta algunas de las diferencias usuales entre vectores y matrices: no pone comas separando las entradas de las matrices y, además, dibuja los corchetes un poco más grandes en el caso de matrices.

3.2.1 Operaciones elementales con matrices

La suma y resta de matrices se indica como es usual,

```
(%i42)  A+B;
```

$$\text{(\%o42)} \quad \begin{bmatrix} 0 & 3 & 4 \\ 0 & 0 & 3 \\ -1 & 8 & 1 \end{bmatrix}$$

```
(%i43)  A-B;
```

$$\text{(\%o43)} \quad \begin{bmatrix} 2 & 1 & 2 \\ -2 & 0 & 3 \\ 5 & -6 & -3 \end{bmatrix}$$

en cambio el producto de matrices se indica con un punto, ".", como ya vimos con vectores. El operador $*$ multiplica los elementos de la matriz entrada a entrada.

```
(%i44)  A.B;
```

$$\text{(\%o44)} \quad \begin{bmatrix} -8 & 22 & 7 \\ -8 & 20 & 5 \\ 2 & -5 & 0 \end{bmatrix}$$

```
(%i45)  A*B;
```

$$\text{(\%o45)} \quad \begin{bmatrix} -1 & 2 & 3 \\ -1 & 0 & 0 \\ -6 & 7 & -2 \end{bmatrix}$$

Con las potencias ocurre algo parecido: "^ ^n" eleva toda la matriz a n, esto es, multiplica la matriz consigo misma n veces,

```
(%i46)  A^^2
```

$$\text{(\%o46)} \quad \begin{bmatrix} 5 & 5 & 6 \\ 5 & 1 & -6 \\ -1 & 3 & 10 \end{bmatrix}$$

y "^n" eleva cada entrada de la matriz a n.

```
(%i47)  A^2
```

$$\text{(\%o47)} \quad \begin{bmatrix} 1 & 4 & 9 \\ 1 & 0 & 9 \\ 4 & 1 & 1 \end{bmatrix}$$

Para el producto de una matriz por un vector sólo tenemos que tener cuidado con utilizar el punto.

```
(%i48)  A.a;
```

$$\text{(\%o48)} \quad \begin{bmatrix} 8 \\ 2 \\ 3 \end{bmatrix}$$

y no tenemos que preocuparnos de si el vector es un vector "fila" o "columna"

```
(%i49)  a.A
```

$$\text{(\%o49)} \quad \begin{bmatrix} 1 & 3 & 8 \end{bmatrix}$$

El único caso en que $*$ tiene el resultado esperado es el producto de una matriz o un vector por un escalar.

```
(%i50)  2*A;
```

$$\text{(\%o50)} \quad \begin{bmatrix} 2 & 4 & 6 \\ -2 & 0 & 6 \\ 4 & 2 & -2 \end{bmatrix}$$

3.2.2 Otras operaciones usuales

`rank(`*matriz*`)`	rango de la matriz
`col(`*matriz*`,i)`	columna i de la *matriz*
`row(`*matriz*`,j)`	fila j de la *matriz*
`minor(`*matriz*`,i,j)`	menor de la matriz obtenido al eliminar la fila i y la columna j
`submatrix(`*fila1*`,`*fila2*`,..,`*matriz*`,`*col1*`,`*col2*`,..)`	matriz obtenida al eliminar las filas y columnas mencionadas
`triangularize(`*matriz*`)`	forma triangular superior de la matriz
`determinant(`*matriz*`)`	determinante
`invert(`*matriz*`)`	matriz inversa
`transpose(`*matriz*`)`	matriz transpuesta
`nullspace(`*matriz*`)`	núcleo de la matriz

Existen órdenes para la mayoría de las operaciones comunes. Podemos calcular la matriz transpuesta con `transpose`,

```
(%i51)  transpose(A);
```

$$(\%o51)\quad \begin{bmatrix} 1 & -1 & 2 \\ 2 & 0 & 1 \\ 3 & 3 & -1 \end{bmatrix}$$

calcular el determinante,

```
(%i52)  determinant(A);
(%o52)  4
```

o, ya que sabemos que el determinante no es cero, su inversa:

```
(%i53)  invert(A);
```

$$(\%o53)\quad \begin{bmatrix} -\frac{3}{4} & \frac{5}{4} & \frac{3}{2} \\ \frac{5}{4} & -\frac{7}{4} & -\frac{3}{2} \\ -\frac{1}{4} & \frac{3}{4} & \frac{1}{2} \end{bmatrix}$$

Como $\det(A) \neq 0$, la matriz A tiene rango 3. En general, podemos calcular el rango de una matriz cualquiera $n \times m$ con la orden `rank`

```
(%i54)  m:matrix([1,3,0,-1],[3,-1,0,6],[5,-3,1,1])$
```

```
(%i55)  rank(m);
(%o55)  3
```

El rango es fácil de averiguar si escribimos la matriz en forma triangular superior utilizando el método de Gauss con la orden `triangularize` y le echamos un vistazo a la diagonal:

```
(%i56)  triangularize(m);
```

$$(\%o56)\quad \begin{bmatrix} 1 & 3 & 0 & -1 \\ 0 & -10 & 0 & 9 \\ 0 & 0 & -10 & 102 \end{bmatrix}$$

Cualquiera de estos métodos es más rapido que ir menor a menor buscando alguno que no se anule. Por ejemplo, el menor de la matriz A que se obtiene cuando se eliminan la segunda fila y la primera columna es

```
(%i57)  minor(A,2,1);
```

$$(\%o57)\quad \begin{bmatrix} 2 & 3 \\ 1 & -1 \end{bmatrix}$$

Caso de que no fuera suficiente con eliminar una única fila y columna podemos eliminar tantas filas y columnas como queramos con la orden `submatrix`. Esta orden elimina todas las filas que escribamos antes de una matriz y todas las columnas que escribamos después. Por ejemplo, para eliminar la primera y última columnas junto con la segunda fila de la matriz m escribimos:

```
(%i58)  submatrix(2,m,1,4);
```

```
(%o58)  [ 3   0 ]
        [-3   1 ]
```

En el extremo opuesto, si sólo queremos una fila o una columna de la matriz, podemos usar el comando `col` para extraer una columna

```
(%i59)  col(m,2);
(%o59)  [  3 ]
        [ -1 ]
        [ -3 ]
```

y el comando `row` para extraer una fila. El resultado de ambas órdenes es una matriz.

```
(%i60)  row(m,1);
(%o60)  [1   3   0   -1]
(%i61)  matrixp(%);
(%o61)  true
```

Para acabar con esta lista de operaciones, conviene mencionar cómo se calcula el núcleo de una matriz. Ya sabes que el núcleo de una matriz $A = (a_{ij})$ de orden $n \times m$ es el subespacio

$$\ker(A) = \{x;\ A.x = 0\}$$

y es muy útil, por ejemplo, en la resolución de sistemas lineales de ecuaciones. La orden `nullspace` nos da una base del núcleo de la matriz:

```
(%i62)  nullspace(matrix([1,2,4],[-1,0,2]));
```

$$(\%o62)\quad \mathrm{span}\left(\begin{bmatrix}-4\\6\\-2\end{bmatrix}\right)$$

3.2.3 Más sobre escribir matrices

Si has utilizado el menú **Álgebra→Introducir matriz** para escribir matrices ya has visto que tienes atajos para escribir matrices diagonales, simétricas y antisimétricas.

`diagmatrix(n,x)`	matriz diagonal $n \times n$ con x en la diagonal
`entermatrix(m,n)`	definir matriz $m \times n$
`genmatrix`	genera una matriz mediante una regla
`matrix[i,j]`	elemento de la fila i, columna j de la matriz

Existen otras formas de dar una matriz en *Maxima*. La primera de ellas tiene más interés si estás utilizando *Maxima* y no *wxMaxima*. Se trata de la orden `entermatrix`. Por ejemplo, para definir una matriz con dos filas y tres columnas, utilizamos `entermatrix(2,3)` y *Maxima* nos va pidiendo que escribamos entrada a entrada de la matriz:

```
(%i63)  c:entermatrix(2,3);
        Row 1 Column 1: 1;
        Row 1 Column 2: 2;
        Row 1 Column 3: 4;
        Row 2 Column 1: -1;
        Row 2 Column 2: 0;
        Row 2 Column 3: 2;
        Matrix entered.
```

$$(\%o63)\quad \begin{bmatrix} 1 & 2 & 4 \\ -1 & 0 & 2 \end{bmatrix}$$

También es fácil de escribir la matriz diagonal que tiene un mismo valor en todas las entradas de la diagonal: sólo hay que indicar el orden y el elemento que ocupa la diagonal. Por ejemplo, la matriz identidad de orden 4 se puede escribir como sigue.

```
(%i64)  diagmatrix(4,1);
```

$$(\%o64)\quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Por último, también podemos escribir una matriz si sabemos una regla que nos diga cuál es el valor de la entrada (i, j) de la matriz. Por ejemplo, para escribir la matriz que tiene como entrada $a_{ij} = i * j$, escribimos en primer lugar dicha regla[8]

```
(%i65)  a[i,j]:=i*j;
(%o65)  a_ij:=ij
```

y luego utilizamos genmatrix para construir la matriz (3×3 en este caso):

```
(%i66)  genmatrix(a,3,3);
(%o66)  [1 2 3]
        [2 4 6]
        [3 6 9]
```

Observa que hemos utilizado corchetes y no paréntesis para definir la regla a_{ij}. Bueno, que ya hemos definido la matriz a...un momento, ¿seguro?

```
(%i67)  matrixp(a);
(%o67)  false
```

¿Pero no acabábamos de definirla? En realidad, no. Lo que hemos hecho es definir la regla que define que permite construir los elementos de la matriz pero no le hemos puesto nombre:

```
(%i68)  c:genmatrix(a,4,5);
```

[8] Si no has borrado el vector a que definimos hace algunas páginas, *Maxima* te dará un error.

```
(%o68)
```
$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 6 & 8 & 10 \\ 3 & 6 & 9 & 12 & 15 \\ 4 & 8 & 12 & 16 & 20 \end{bmatrix}$$

Podemos utilizar la misma notación para referirnos a los elementos de la matriz. Por ejemplo, al elemento de la fila i y la columna j, nos referimos como $c[i,j]$ (de nuevo, observa que estamos utilizando corchetes):

```
(%i69)  c[2,3];
(%o69)  6
```

3.2.4 Valores propios

`charpoly(`*`matriz`*`,`*`variable`*`)`	polinomio característico
`eigenvalues(`*`matriz`*`)`	valores propios de la matriz
`eigenvectors(`*`matriz`*`)`	valores y vectores propios de la matriz

Los valores propios de una matriz cuadrada, A, son las raíces del polinomio característico $\det(A - xI)$, siendo I la matriz identidad. La orden `charpoly` nos da dicho polinomio.

```
(%i70)  S:matrix([-11/15,-2/15,-4/3],[-17/15,16/15,-1/3],[-8/15,4/15,5/3]);
```

```
(%o70)  [ -11/15   -2/15   -4/3 ]
        [ -17/15   16/15   -1/3 ]
        [  -8/15    4/15    5/3 ]
(%i71)  charpoly(S,x);
```

$$(\%o71)\quad \left(\left(\frac{16}{15}-x\right)\left(\frac{5}{3}-x\right)+\frac{4}{45}\right)\left(-x-\frac{11}{15}\right)+\frac{2\left(-\frac{17\left(\frac{5}{3}-x\right)}{15}-\frac{8}{45}\right)}{15}-\frac{4\left(\frac{8\left(\frac{16}{15}-x\right)}{15}-\frac{68}{225}\right)}{3}$$

```
(%i72)  expand(%);
(%o72)  -x^3+2x^2+x-2
```

Por tanto, sus valores propios son

```
(%i73)  solve(%,x);
(%o73)  [x=2,x=-1,x=1]
```

Todo este desarrollo nos lo podemos ahorrar: la orden `eigenvalues` nos da los valores propios junto con su multiplicidad.

```
(%i74)  eigenvalues(S);
(%o74)  [[2,-1,1],[1,1,1]]
```

En otras palabras, los valores propios son 2, -1 y 1 todos con multiplicidad 1. Aunque no lo vamos a utilizar, también se pueden calcular los correspondientes vectores propios con la orden `eigenvectors`:

```
(%i75)   eigenvectors(S);
(%o75)   [[[2,-1,1],[1,1,1]],[1,-1/2,-2],[1,4/7,1/7],[1,7,-2]]
```

La respuesta es, en esto caso, cinco listas. Las dos primeras las hemos visto antes: son los valores propios y sus multiplicidades. Las tres siguientes son los tres vectores propios asociados a dichos valores propios.

3.3 Ejercicios

Ejercicio 3.1. Consideremos los vectores $a = (1, 2, -1)$, $b = (0, 2, 3/4)$, $c = (e, 1, 0)$, y $d = (0, 0, 1)$. Realiza las siguientes operaciones

a) $a + b$,
b) $3c + 2b$,
c) $c.d$, y
d) $b.d + 3a.c$.

Ejercicio 3.2. Consideremos las matrices

$$A = \begin{pmatrix} 1 & -2 & 0 \\ 2 & 5 & 3 \\ -3 & 1 & -4 \end{pmatrix} \qquad B = \begin{pmatrix} 0 & -2 & 6 \\ 12 & 2 & 0 \\ -1 & -1 & 3 \end{pmatrix}$$

$$C = \begin{pmatrix} 1 & 2 & 0 & -5 \\ -4 & -2 & 1 & 0 \\ 3 & 2 & -1 & 3 \\ 5 & 4 & -1 & -5 \end{pmatrix} \qquad D = \begin{pmatrix} -1 & 2 & 3 & 0 \\ 12 & -5 & 0 & 3 \\ -6 & 0 & 0 & 1 \end{pmatrix}$$

a) Calcular $A.B$, $A + B$, $D.C$.
b) Extraer la segunda fila de A, la tercera columna de C y el elemento $(3, 3)$ de D.
c) Calcular $\det(A)$, $\det(B)$ y $\det(C)$. Para las matrices cuyo determinante sea no nulo, calcular su inversa. Calcular sus valores propios.
d) Calcular el rango de las matrices A, B, C, D, $D.C$ y $A + B$.
e) Construye una matriz del orden 3×3, de forma que el elemento (i, j) sea $i * j + j - i$. Calcula el determinante, su inversa si la tiene, y su rango. ¿Cuáles son sus valores propios?

Ejercicio 3.3. Calcula el rango de la matriz

$$A = \begin{pmatrix} 2 & 7 & -4 & 3 & 0 & 1 \\ 0 & 0 & 5 & -4 & 1 & 0 \\ 2 & 1 & 0 & -2 & 1 & 3 \\ 0 & 6 & 1 & 1 & 0 & -2 \end{pmatrix}$$

Ejercicio 3.4. Calcula los valores y vectores propios de las siguientes matrices:

$$A = \begin{pmatrix} 0 & 4 \\ 4 & -4 \end{pmatrix}, \quad B = \begin{pmatrix} 3 & 0 & 4 \\ 0 & 3 & 1 \\ 4 & 1 & -4 \end{pmatrix} \text{ y } C = \begin{pmatrix} 0 & 3 & 9 \\ -4 & 8 & 10 \\ 8 & -4 & -2 \end{pmatrix}$$

Ejercicio 3.5.

a) Genera una lista de 10 números aleatorios entre 5 y 25 y reordénala en orden decreciente.

Ejercicio 3.6. Define `listauno:makelist(i,i,2,21)`, `listados:makelist(i,i,22,31)`. Realiza las siguientes operaciones usando algunos de los comandos antes vistos.

a) Multiplica cada elemento de "listauno" por todos los elementos de "listados". El resultado será una lista con 20 elementos (que a su vez serán listas de 10 elementos), a la que llamarás "productos".
b) Calcula la suma de cada una de las listas que forman la lista "productos" (no te equivoques, comprueba el resultado). Obtendrás una lista con 20 números.
c) Calcula el producto de los elementos de la lista obtenida en el apartado anterior.

Ejercicio 3.7. Genera una lista de 30 elementos cuyos elementos sean listas de dos números que no sean valores exactos.

Ejercicio 3.8.

a) Calcula la suma de los números de la forma $\frac{(-1)^{k+1}}{\sqrt{k}}$ desde $k = 1$ hasta $k = 1000$.

b) Calcula el producto de los números de la forma $\left(1 + \frac{1}{k^2}\right)$ desde $k = 1$ hasta $k = 1000$.

4 Resolución de ecuaciones

Maxima nos va a ser de gran ayuda en la resolución de ecuaciones, ya sean sistemas de ecuaciones lineales con un número grande de incógnitas (y parámetros) o ecuaciones no lineales. Un ejemplo típico es encontrar las soluciones de un polinomio. En este caso es fácil que alguna de las soluciones sea compleja. No importa. *Maxima* se maneja bien con números complejos. De hecho, *siempre* trabaja con números complejos. Si tienes alguna duda de cómo operar con números complejos, en el Apéndice A tienes una breve introducción sobre su uso.

4.1 Ecuaciones y operaciones con ecuaciones

En *Maxima*, una ecuación es una igualdad entre dos expresiones algebraicas escrita con el símbolo =.

expresión1=expresión2	ecuación
lhs (*expresión1=expresión2*)	expresión1
rhs (*expresión1=expresión2*)	expresión2

Si escribimos una ecuación, *Maxima* devuelve la misma ecuación.

```
(%i1)  3*x^2+2*x+x^3-x^2=4*x^2;
(%o1)  x^3+2x^2+2x=4x^2
```

además podemos asignarle un nombre para poder refererirnos a ella

```
(%i2)   eq:3*x^2+2*x+x^3-a*x^2=4*x^2;
(%o2)   x^3-ax^2+3x^2+2x=4x^2
```

y operar como con cualquier otra expresión

```
(%i3)   eq-4*x^2;
(%o3)   x^3-ax^2-x^2+2x=0
```

Podemos seleccionar la expresión a la izquierda o la derecha de la ecuación con las órdenes `lhs` y `rhs` respectivamente.

```
(%i4)   lhs(eq);
(%o4)   x^3-ax^2+3x^2+2x
```

4.2 Resolución de ecuaciones

Maxima puede resolver los tipos más comunes de ecuaciones y sistemas de ecuaciones algebraicas de forma exacta. Por ejemplo, sabe encontrar las raíces de polinomios de grado bajo (2,3 y 4). En cuanto a polinomios de grado más alto o ecuaciones más complicadas, no siempre será posible encontrar la solución exacta. En este caso, podemos intentar encontrar una solución aproximada en lugar de la solución exacta.

4.2.1 La orden `solve`

La primera orden que aparece en el menú dentro de *Maxima* para resolver ecuaciones es `solve`. Esta orden intenta dar todas las soluciones, ya sean reales o complejas de una ecuación o sistema de ecuaciones. Se puede acceder a esta orden desde el menú **Ecuaciones→Resolver** o escribiendo directamente en la ventana de *Maxima* la ecuación a resolver.

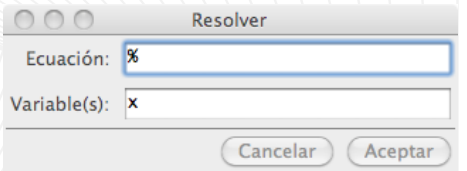


Figura 4.1 Resolver una ecuación

```
(%i5)  solve(x^2-3*x+1=0,x);
```

$$(\%o5)\quad [x=-\frac{\sqrt{5}-3}{2},x=\frac{\sqrt{5}+3}{2}]$$

`solve(`*ecuación*`,`*incógnita*`)`	resuelve la ecuación
`solve(`*expr*`,`*incógnita*`)`	resuelve la expresión igualada a cero
`solve([`*ecuaciones*`],[`*variables*`])`	resuelve el sistema
`multiplicities`	guarda la multiplicidad de las soluciones

También podemos resolver ecuaciones que dependan de algún parámetro. Consideremos la ecuación "eq1":

```
(%i6)   eq1:x^3-a*x^2-x^2+2*x=0;
(%o6)   x^3-a*x^2-x^2+2*x=0
(%i7)   solve(eq1,x);
(%o7)   [x=-(sqrt(a^2+2*a-7)-a-1)/2,x=(sqrt(a^2+2*a-7)+a+1)/2,x=0]
```

Sólo en el caso de ecuaciones con una única variable podemos ahorrarnos escribirla

```
(%i8)   solve(x^2+2*x=3);
(%o8)   [x=-3,x=1]
```

También podemos no escribir el segundo miembro de una ecuación cuando éste sea cero

```
(%i9)   solve(x^2+2*x);
(%o9)   [x=-2,x=0]
(%i10)   solve(x^2+2*x=0);
(%o10)   [x=-2,x=0]
```

Cuando buscamos las raíces de un polinomio hay veces que es conveniente tener en cuenta la multiplicidad de las raíces. Ésta se guarda automáticamente en la variable `multiplicities`. Por ejemplo, el polinomio $x^7 - 2\,x^6 + 2\,x^5 - 2\,x^4 + x^3$ tiene las raíces 0, 1, i, $-i$,

```
(%i11)   solve(x^7-2*x^6+2*x^5-2*x^4+x^3);
(%o11)   [x=-%i,x=%i,x=1,x=0]
```

pero estas raíces no pueden ser simples: es un polinomio de grado 7 y sólo tenemos 4 raíces. ¿Cuál es su multiplicidad?

```
(%i12)   multiplicities;
(%o12)   [1,1,2,3]
```

O sea que i y $-i$ son raíces simples, 1 tiene multiplicidad 2 y 0 es una raíz triple.

Observación 4.1. Mucho cuidado con olvidar escribir cuáles son las incógnitas.

```
(%i13)   solve(eq1);
         More unknowns than equations - 'solve'
         Unknowns given :
         [a,x]
         Equations given:
         [x^3-ax^2+3x^2+2x=4x^2]
         - an error. To debug this try debugmode(true);
```

Hay dos variables y *Maxima* no sabe con cuál de ellas quedarse como incógnita. Aunque nosotros estemos acostumbrados a utilizar las letras x, y, z como incógnitas, para *Maxima* tanto a como x tienen perfecto sentido como incógnitas y la respuesta en uno de ellos no nos interesa:

```
(%i14)  solve(eq1,a);
(%o14)  [a=(x^2-x+2)/2]
```

La orden `solve` no sólo puede resolver ecuaciones algebraicas.

```
(%i15)  solve(sin(x)*cos(x)=0,x);
        solve: is using arc-trig functions to get a solution.
        Some solutions will be lost.
(%o15)  [x=0,x=%pi/2]
```

¿Qué ocurre aquí? La expresión $\operatorname{sen}(x)\cos(x)$ vale cero cuando el seno o el coseno se anulen. Para calcular la solución de $\operatorname{sen}(x) = 0$ aplicamos la función arcoseno a ambos lados de la ecuación. La función arcoseno vale cero en cero pero la función seno se anula en muchos más puntos. Nos estamos dejando todas esas soluciones y eso es lo que nos está avisando *Maxima*.

Como cualquiera puede imaginarse, *Maxima* no resuelve todo. Incluso en las ecuaciones más "sencillas", las polinómicas, se presenta el primer problema: no hay una fórmula en términos algebraicos para obtener las raíces de un polinomio de grado 5 o más. Pero no hay que ir tan lejos. Cuando añadimos raíces, logaritmos, exponenciales, etc., la resolución de ecuaciones se complica mucho. En esas ocasiones lo más que podemos hacer es ayudar a *Maxima* a resolverlas.

```
(%i16)  eq:x+3=sqrt(x+1);
(%o16)  x+3=sqrt(x+1)
```

```
(%i17)  solve(eq,x);
(%o17)  [x=sqrt(x+1)-3]
(%i18)  solve(eq^2);
(%o18)  [x=-(sqrt(7)%i+5)/2,x=(sqrt(7)%i-5)/2]
```

Cómo hacer referencia a las soluciones

Uno de los ejemplos usuales en los que utilizaremos las soluciones de una ecuación es en el estudio de una función. Necesitaremos calcular puntos críticos, esto es, ceros de la derivada. El resultado de la orden `solve` no es una lista de puntos, es una lista de ecuaciones.

Una primera solución consiste en usar la orden `rhs` e ir recorriendo uno a uno las soluciones:

```
(%i19)  sol:solve(x^2-4*x+3);
(%o19)  [x=3,x=1]
```

La primera solución es

```
(%i20)  rhs(part(sol,1));
(%o20)  3
```

y la segunda

```
(%i21)   rhs(part(sol,2));
(%o21)   1
```

Este método no es práctico en cuanto tengamos un número un poco más alto de soluciones. Tenemos que encontrar una manera de aplicar la orden `rhs` a toda la lista de soluciones. Eso es justamente para lo que habíamos presentado la orden `map`:

```
(%i22)   sol:map(rhs,solve(x^2-4*x+3));
(%o22)   [3,1]
```

Sistemas de ecuaciones

También podemos resolver sistemas de ecuaciones. Sólo tenemos que escribir la lista de ecuaciones y de incógnitas. Por ejemplo, para resolver el sistema

$$\left.\begin{aligned} x^2+y^2 &= 1 \\ (x-2)^2+(y-1)^2 &= 4 \end{aligned}\right\}$$

escribimos

```
(%i23)   solve([x^2+y^2=1,(x-2)^2+(y-1)^2=4],[x,y]);
(%o23)   [[x=4/5,y=-3/5],[x=0,y=1]]
```

Siempre hay que tener en cuenta que, por defecto, *Maxima* da todas las soluciones incluyendo las complejas aunque muchas veces no pensemos en ellas. Por ejemplo, la recta $x+y=5$ no corta a la circunferencia $x^2+y^2=1$:

```
(%i24)  solve([x^2+y^2=1,x+y=5],[x,y]);
(%o24)  [[x=-(sqrt(23)%i-5)/2,y=(sqrt(23)%i+5)/2],[x=(sqrt(23)%i+5)/2,y=-(sqrt(23)%i-5)/2]]
```

Si la solución depende de un parámetro o varios, *Maxima* utilizará %r1, %r2,... para refererirse a estos. Por ejemplo,

```
(%i25)  solve([x+y+z=3,x-y=z],[x,y,z]);
(%o25)  [[x=3/2,y=-(2%r1-3)/2,z=%r1]]
```

¿Qué pasa si el sistema de ecuaciones no tiene solución? Veamos un ejemplo (de acuerdo, no es muy difícil)

```
(%i26)  solve([x+y=0,x+y=1],[x,y]);
[]
```

¿Y si todos los valores de x cumplen la ecuación?

```
(%i27)   solve((x+1)^2=x^2+2x+1,x);
(%o27)   [x=x]
```

Maxima nos dice que el sistema se reduce a $x = x$ que claramente es cierto para todo x. El siguiente caso es similar. Obviamente $(x+y)^2 = x^2 + 2xy + y^2$. ¿Qué dice al respecto *Maxima*?

```
(%i28)   solve((x+y)^2=x^2+2*x*y+y^2,[x,y]);
         Dependent equations eliminated: (1)
(%o28)   [[x=%r3,y=%r2]]
```

En otras palabras, x puede tomar cualquier valor e y lo mismo.

4.2.2 to_poly_solve

Hay una segunda forma de resolver ecuaciones en *Maxima*. puedes acceder a ella desde el menú **Ecuaciones→Resolver (to_poly)**. Sin entrar en detalles, algunas ecuaciones las resuelve mejor. Por ejemplo, cuando hay radicales por medio la orden `solve` no siempre funciona bien:

```
(%i29)   solve(3*x=sqrt(x^2+1),x);
(%o29)   [x=sqrt(x^2+1)/3]
```

En cambio, tiene un poco más de éxito

```
(%i30)  to_poly_solve(3*x=sqrt(x^2+1),x);
(%o30)  %union([x = 1/2^(3/2)])
```

Como puedes ver, la respuesta es el *conjunto* de soluciones que verifica la ecuación. De ahí la palabra `union` delante de la respuesta.

`to_poly_solve[`*ecuación, variable*`]`	resuelve la ecuación
`to_poly_solve[`*expr, variable*`]`	resuelve la expresión igualada a cero

Además de este ejemplo, hay otras ocasiones en las que la respuesta de `to_poly_solve` es mejor o más completa. Por ejemplo, con funciones trigonométricas ya hemos visto que `solve` no da la lista completa de soluciones

```
(%i31)  solve[x*cos(x)];
        solve: using arc-trig functions to get a solution.
        Some solutions will be lost.
(%o31)  [x=0]
```

En cambio, con `to_poly_solve` la respuesta es un poco más amplia

```
(%i32)  to_poly_solve(x*cos(x),x);
```

```
(%o32)   %union([x=0],[x=2π%z101 − π/2],[x=2π%z103 + π/2])
```

Los parámetros "z101" y "z103" indican un número entero arbitrario y la numeración depende del número de operaciones que hayas realizado. No es quizá la forma más elemental de escribir la solución, pero sí que tenemos todas las soluciones de la ecuación. Observa también que en este caso no hemos escrito una ecuación sino una expresión y `to_poly_solve` ha resuelto dicha expresión igualada a cero lo mismo que ocurría con la orden `solve`.

4.2.3 Sistemas de ecuaciones lineales

`linsolve([`*ecuaciones*`],[`*variables*`])`	resuelve el sistema

En el caso particular de sistemas de ecuaciones lineales puede ser conveniente utilizar `linsolve` en lugar de `solve`. Ambas órdenes se utilizan de la misma forma, pero `linsolve` es más eficiente en estos casos. Sólo una observación: sigue siendo importante escribir correctamente qué variables se consideran como incógnitas. El resultado puede ser muy diferente dependiendo de esto.

```
(%i33)   eq:[x+y+z+w=1,x-y+z-w=-2,x+y-w=0]$
(%i34)   linsolve(eq,[x,y,z]);
(%o34)   [x=(4 w-3)/2,y=-(2 w-3)/2,z=1-2 w]
```

¿Cuál es el resultado de `linsolve(eq,[x,y,z,w])`?

4.2.4 Algsys

`algsys([`*ecuaciones*`],[`*variables*`])`	resuelve la ecuación o ecuaciones
`realonly`	si vale `true`, `algsys` muestra sólo soluciones reales

La orden `algsys` resuelve ecuaciones o sistemas de ecuaciones algebraicas. La primera diferencia con la orden `solve` es pequeña: `algsys` siempre tiene como entrada listas, en otras palabras, tenemos que agrupar la ecuación o ecuaciones entre corchetes igual que las incógnitas.

```
(%i35)   eq:x^2-4*x+3;
(%o35)   x^2-4*x+3
(%i36)   algsys([eq],[x]);
(%o36)   [[x=3],[x=1]]
```

La segunda diferencia es que `algsys` intenta resolver numéricamente la ecuación si no es capaz de encontrar la solución exacta.

```
(%i37)   solve(eq:x^6+x+1);
(%o37)   [0=x^6+x+1]
(%i38)   algsys([eq],[x]);
```

```
(%o38)  [[x=-1.038380754458461%i-0.15473514449684],
        [x=1.038380754458461%i-0.15473514449684],
        [x=-0.30050692030955%i-0.79066718881442],
        [x=0.30050692030955%i-0.79066718881442],
        [x=0.94540233331126-0.61183669378101%i],
        [x=0.61183669378101%i+0.94540233331126]]
```

En general, para ecuaciones polinómicas `algsys` nos permite algo más de flexibilidad ya que funciona bien con polinomios de grado alto y, además, permite seleccionar las raíces reales. El comportamiento de `algsys` está determinado por la variable `realonly`. Su valor por defecto es `false`. Esto significa que `algsys` muestra todas las raíces. Si su valor es `true` sólo muestra las raíces reales.

```
(%i39)  eq:x^4-1=0$
(%i40)  realonly;
(%o40)  false
(%i41)  algsys([eq],[x]);
(%o41)  [[x=1],[x=-1],[x=%i],[x=-%i]]
(%i42)  realonly:true$
(%i43)  algsys([eq],[x]);
(%o43)  [[x=1],[x=-1]
```

4.3 Ejercicios

Ejercicio 4.1. Calcula los puntos donde se cortan las parábolas $y = x^2$, $y = 2x^2 + ax + b$. Discute todos los casos posibles dependiendo de los valores de a y b.

Ejercicio 4.2. Dibuja, en un mismo gráfico, la elipse de semieje horizontal $a = 3$ y de semieje vertical $b = 5$ y la bisectriz del primer cuadrante. Calcula los puntos donde se cortan ambas curvas.

Ejercicio 4.3. Consideremos la circunferencia de centro $(0, 0)$ y radio 2. Dibújala. Ahora consideremos un rectángulo centrado en el origen e inscrito en ella. Determina el rectángulo así construido cuya área sea 1.

Ejercicio 4.4. Representa gráficamente y determina los puntos de corte de las siguientes curvas:

a) la recta $x - y = 5$ y la parábola $(x - 1)^2 + y = 4$;
b) la hipérbola equilátera y la circunferencia de centro $(-1, 1)$ y radio 1;
c) las circunferencias de centro $(0, 0)$ y radio 2 y la de centro $(-1, 3)$ y radio 3.

Ejercicio 4.5. Resolver la ecuación logarítmica:

$$\log(x) + \log(x + 1) = 3$$

5 Métodos numéricos de resolución de ecuaciones

En este capítulo vamos a ver cómo encontrar soluciones aproximadas a ecuaciones que no podemos resolver de forma exacta. En la primera parte, presentamos algunos de los comandos incluidos en *Maxima* para este fin. En la segunda parte, mostramos algunos métodos para el cálculo de soluciones como el método de bisección o el de Newton-Raphson

Comenzamos la primera sección hablando sobre las ventajas e inconvenientes de trabajar en modo numérico.

5.1 Introducción al análisis numérico

Los ordenadores tienen un capacidad limitada para almacenar cada número real por lo que en un ordenador únicamente pueden representarse un número finito de números reales: los números máquina. Si un número real no coincide con uno de estos números máquina, entonces se aproxima al más próximo. En este proceso se pueden producir, y de hecho se producen, errores de redondeo al eliminar decimales. También se pueden introducir errores en la conversión entre sistema decimal y sistema binario: puede ocurrir que un número que en sistema decimal presente un número finito de dígitos, en sistema binario presente un número infinito de los mismos.

Como consecuencia de esto, algunas propiedades aritméticas dejan de ser ciertas cuando utilizamos un ordenador.

La precisión de un número máquina depende del número de bits utilizados para ser almacenados.

Puede producirse una severa reducción en la precisión si al realizar los cálculos se restan dos números similares. A este fenómeno se le conoce como cancelación de cifras significativas. Lo que haremos para evitar este fenómeno será reorganizar los cálculos en un determinado desarrollo.

5.1.1 Números y precisión

Todos los números que maneja Maxima tienen precisión arbitraria. Podemos calcular tantos decimales como queramos. Si es posible, Maxima trabaja de forma exacta

```
(%i1)   sqrt(2);
(%o1)   √2
```

o podemos con la precisión por defecto

```
(%i2)   sqrt(2),numer;
(%o2)   1.414213562373095
```

Cuando decimos que $\sqrt{2}$ es un número de precisión arbitraria no queremos decir que podamos escribir su expresión decimal completa (ya sabes que es un número irracional) sino que podemos elegir el número de digitos que deseemos y calcular su expresión decimal con esa precisión.

```
(%i3)   fpprec:20;
(%o3)   20
```

```
(%i4)   bfloat(sqrt(2));
(%o4)   1.414213562373095048 8b0
```

Vamos a comentar un par de detalles que tenemos que tener en cuenta en este proceso.

5.1.1.1 Errores de redondeo

Si sólo tenemos 5 dígitos de precisión, ¿cómo escribimos el número 7.12345? Hay dos métodos usuales: podemos truncar o podemos redondear. Por truncar se entiende desechar los dígitos sobrantes. El redondeo consiste en truncar si los últimos dígitos están entre 0 y 4 y aumentar un dígito si estamos entre 5 y 9. Por ejemplo, 7.46 se convertiría en 7.4 si truncamos y en 7.5 si redondeamos. El error es siempre menor en utilizando rendondeo. ¿Cuál de las dos formas usa Maxima? Puedes comprobarlo tu mismo.

```
(%i5)   fpprec:5;
(%o5)   5
(%i6)   bfloat(7.12345);
(%o6)   7.1235b0
```

¿Qué pasa si aumentamos la precisión en lugar de disminuirla?

```
(%i7)   fpprec:20;
(%o7)   20
(%i8)   bfloat(0.1);
```

```
(%o8)    1.0000000000000000555b-1
```

¿Qué ha pasado? 0.1 es un número exacto. ¿Porqué la respuesta no ha sido 0.1 de nuevo? Fíjate en la siguiente respuesta

```
(%i9)    bfloat(1/10);
(%o9)    1.0b-1
```

¿Cuál es la diferencia entre una otra? ¿Porqué una es exacta y la otra no? La diferencia es el error que se puede añadir (y acabamos de ver que se añade) cuando pasamos de representar un número en el sistema decimal a binario y viceversa.

5.1.2 Aritmética de ordenador

Sabemos que el ordenador puede trabajar con números muy grandes o muy pequeños; pero, por debajo de cierto valor, un número pequeño puede hacerse cero debido al error de redondeo. Por eso hay que tener cuidado y recordar que propiedades usuales en la aritmética real (asociatividad, elemento neutro) no son ciertas en la aritmética de ordenador.

5.1.2.1 Elemento neutro

Tomamos un número muy pequeño, pero distinto de cero y vamos a ver cómo *Maxima* interpreta que es cero:

```
(%i10)   h:2.22045*10^(-17);
```

```
(%o10)    2.22045 10^-17
```

Y si nos cuestionamos si h funciona como elemento neutro:

```
(%i11)    is(h+1.0=1.0);
(%o11)    true
```

la respuesta es que sí que es cierto que h+1.0=1.0, luego h sería cero.

Por encima, con números muy grandes puede hacer cosas raras.

```
(%i12)    g:15.0+10^(20);
(%o12)    1.10^20
(%i13)    is(g-10^(20)=0);
(%o13)    false
(%i14)    g-10^(20);
(%o14)    0.0
```

Aquí no sale igual, pero si los restáis cree que la diferencia es cero.

5.1.2.2 Propiedad asociativa de la suma

Con aritmética de ordenador vamos a ver que no siempre se cumple que: $(a+b)+c = a+(b+c)$

```
(%i15)   is((11.3+10^(14))+(-(10)^14)=11.3+(10^(14)+(-(10)^14)));
(%o15)   false
```

Si ahora trabajamos con números exactos, vamos a ver qué pasa:

```
(%i16)   is((113/10+10^(14))+(-(10)^14)=113/10+(10^(14)+(-(10)^14)));
(%o16)   true
```

5.1.3 Cancelación de cifras significativas

Como hemos visto, uno de los factores que hay que tener en cuenta a la hora de realizar cálculos, son aquellas operaciones que involucren valores muy grandes o cercanos a cero. Esta situación se presenta por ejemplo, el cálculo de la diferencia de los cuadrados de dos números muy similares

```
(%i17)   a:1242123.78$
         b:1242123.79$
         a^2-b^2;
         (a-b)*(a+b);
(%o18)   -24842.4755859375
(%o18)   -24842.47572313636
```

¿Por cierto? ¿Cuál es el resultado correcto? Probemos de otra forma

```
(%i19)  a:124212378$
        b:124212379$
        a^2-b^2;
        (a-b)*(a+b);
(%o20)  -248424757
(%o21)  -248424757
```

Parece que el resultado correcto es -24842.4757. Ninguno de los dos anteriores. Vale. Veamos otro ejemplo usando basado en la misma idea. Fijemos la precisión 40 y consideremos el número a:

```
(%i22)  fpprec:40;
(%o22)  40
(%i23)  a:bfloat(1-(10)^(-30));
(%o23)  9.999999999999999999999999999999b-1
```

Ahora vamos a calcular: $1+a$ y $(a^2-1)/(a-1)$. Deberían ser iguales, ya que ambas expresiones matemáticamente son equivalentes:

$$1+a=\frac{(a-1)(a+1)}{a-1}=\frac{a^2-1}{a-1},$$

en cambio,

```
(%i24)  b:1+a$
```

```
(%i25)   c:(a^2-1)/(a-1)$
(%i26)   is(b=c);
(%o26)   false
```

No las reconoce como iguales. Este es el resultado del efecto de cancelación de cifras significativas que tiene lugar cuando se restan dos cantidades muy parecidas. En este caso es claro cuál de ambas formas de realizar el cálculo es mejor.

5.2 Resolución numérica de ecuaciones con *Maxima*

Las ecuaciones polinómicas se pueden resolver de manera aproximada. Los comandos `allroots` y `realroots` están especializados en encontrar soluciones racionales aproximadas de polinomios en una variable.

`allroots(`*polinomio*`)`	soluciones aproximadas del polinomio
`bfallroots(`*polinomio*`)`	soluciones aproximadas del polinomio con precisión arbitraria
`realroots(`*polinomio*`)`	soluciones aproximadas reales del polinomio
`realroots(`*polinomio, error*`)`	soluciones aproximadas reales del polinomio con cota del error
`nroots(`*polinomio,a,b*`)`	número de soluciones reales del polinomio entre a y b
`algsys([`*ecuaciones*`],[`*variables*`])`	resuelve la ecuación o ecuaciones

Estás órdenes nos dan todas las soluciones reales y complejas de un polinomio en una variable y son útiles en polinomios de grado alto cuando falla la orden `solve`. La primera de ellas, `allroots`, nos da las soluciones con la precisión por defecto

```
(%i27)   eq:x^9+x^7-x^4+x$
(%i28)   allroots(eq);
```

```
(%o28)  [x=0.0,x=0.30190507748312%i+0.8440677798278,
        x=0.8440677798278-0.30190507748312%i,
        x=0.8923132916888%i-0.32846441923834,
        x=-0.8923132916888%i-0.32846441923834,
        x=0.51104079208431%i-0.80986929589487,
        x=-0.51104079208431%i-0.80986929589487,
        x=1.189238256723466%i+0.29426593530541,
        x=0.29426593530541-1.189238256723466%i]
```

Si queremos una precisión determinada, usamos la orden `bfallroots`.

```
(%i29)  fpprec:6$
        bfallroots(eq);
(%o29)  [x=0.0b0,x=3.0191b-1%i+8.44063b-1,
        x=8.44063b-1-3.0191b-1%i,x=8.92279b-1%i-3.28481b-1,
        x=-8.92279b-1%i-3.28481b-1,x=5.11037b-1%i-8.09838b-1,
        x=-5.11037b-1%i-8.09838b-1,x=1.18924b0%i+2.94256b-1,
        x=2.94256b-1-1.18924b0%i]
```

Si sólo nos interesan las soluciones reales, la orden `realroots` calcula soluciones racionales aproximadas del polinomio.

```
(%i30)  eq1:x^4-3*x^3+x^2-4*x+12$
(%i31)  realroots(eq1);
```

```
(%o31)    [x=2,x=81497599/33554432]
```

Si comparas con la salida de `allroots`, comprobarás que 2 es solución, pero que $\frac{81497599}{33554432}$ sólo es una solución aproximada. La precisión con la que se realiza la aproximación se puede controlar con un segundo parámetro. Por ejemplo, si queremos que el error sea menor que 10^{-5}, escribimos lo siguiente.

```
(%i32)    realroots(eq1,10^(-5));
(%o32)    [x=2,x=636699/262144]
```

Recuerda que la variable `multiplicities` guarda la multiplicidad de cada una de las raíces de la última ecuación que has resuelto.

```
(%i33)    realroots((x-2)^2*eq1,10^(-5));
(%o33)    [x=2,x=636699/262144]
(%i34)    multiplicities;
(%o34)    [3,1]
```

Por último, comentar que es posible saber el número de raíces de un polinomio en una variable en un intervalo concreto[9]

[9] Se admiten $\pm\infty$ como posibles extremos del intervalo

```
(%i35)   nroots(eq1,0,2);
(%o35)   1
```

eso sí, ten cuidado porque se cuentan raíces con su multiplicidad

```
(%i36)   nroots((x-2)^2*eq1,0,2);
(%o36)   3
```

El teorema de los ceros de Bolzano

Uno de los primeros resultados que aprendemos sobre funciones continuas es que si cambian de signo tienen que valer cero en algún momento. Para que esto sea cierto nos falta añadir un ingrediente: la funciones tienen que estar definidas en intervalos. Este resultado se conoce como teorema de los ceros de Bolzano y es una variante del teorema del valor intermedio.

Teorema 5.1. *Sea $f : [a, b] \to \mathbb{R}$ una función continua verificando que $f(a)f(b) < 0$, entonces existe $c \in]a, b[$ tal que $f(c) = 0$.*

Ejemplo 5.2. Una de las utilidades más importantes del Teorema de los ceros de Bolzano es garantizar que una ecuación tiene solución. Por ejemplo, para comprobar que la ecuación $e^x + \log(x) = 0$ tiene solución, estudiamos la función $f(x) = e^x + \log(x)$: es continua en $\mathbb{R}^+$ y se puede comprobar que $f(e^{-10}) < 0$ y $0 < f(e^{10})$. Por tanto, la ecuación $e^x + \log(x) = 0$ tiene al menos una solución entre e^{-10} y e^{10}. En particular, tiene solución en $\mathbb{R}^+$. ◁

`find_root(f(x),x,a,b)`	solución de f en $[a,b]$

El comando `find_root` encuentra una solución de una función (ecuación) continua que cambia de signo por el método de bisección, esto es, dividiendo el intervalo por la mitad y quedándose con aquella mitad en la que la función sigue cambiando de signo. En realidad el método que utiliza *Maxima* es algo más elaborado pero no vamos a entrar en más detalles.

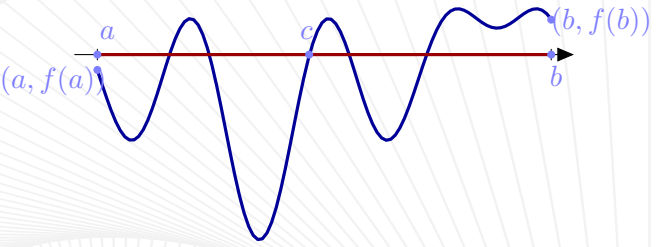


Figura 5.1 Teorema de los ceros de Bolzano

```
(%i37)   f(x):=exp(x)+log(x);
(%o37)   f(x):=exp(x)+log(x)
```

Buscamos un par de puntos donde cambie de signo

```
(%i38)   f(1)
(%o38)   %e
(%i39)   f(exp(-3));
(%o39)   %e^%e^-3+3
```

¿Ese número es negativo?

```
(%i40)   is(f(exp(-3))<0);
```

```
(%o40)    true
```

o bien,

```
(%i41)    f(exp(-3)),numer;
(%o41)    -1.948952728663784
```

Vale, ya que tenemos dos puntos donde cambia de signo podemos utilizar `find_root`:

```
(%i42)    find_root(f(x),x,exp(-3),1);
(%o42)    0.26987413757345
```

Observación 5.3. Este método encuentra *una* solución pero no nos dice cuántas soluciones hay. Para eso tendremos que echar mano de otras herramientas adicionales como, por ejemplo, el estudio de la monotonía de la función.

5.2.1 Ejercicios

Ejercicio 5.1. Calcula las soluciones de $8\,\mathrm{sen}(x) + 1 - \frac{x^2}{3} = 0$.

Ejercicio 5.2. Encuentra una solución de la ecuación $\tan(x) = \frac{1}{x}$ en el intervalo $\left]0, \frac{\pi}{2}\right[$.

Ejercicio 5.3. ¿Cuantas soluciones tiene la ecuación $\frac{e^x}{2} - 2\,\mathrm{sen}(x) = 1$ en el intervalo $[-3, 3]$?

5.3 Breves conceptos de programación

Hemos visto cómo resolver ecuaciones y sistemas de ecuaciones con *Maxima* mediante la orden `solve` o `algsys`. La resolución de ecuaciones y sistemas de ecuaciones de manera exacta está limitada a aquellas para los que es posible aplicar un método algebraico sencillo. En estas condiciones, nos damos cuenta de la necesidad de encontrar o aproximar soluciones para ecuaciones del tipo $f(x) = 0$, donde, en principio, podemos considerar como f cualquier función real de una variable. Nuestro siguiente objetivo es aprender a "programar" algoritmos con *Maxima* para aproximar la solución de estas ecuaciones.

Lo primero que tenemos que tener en cuenta es que no existe ningún método general para resolver todo este tipo de ecuaciones en un número finito de pasos. Lo que sí tendremos es condiciones para poder asegurar, bajo ciertas hipótesis sobre la función f, que un determinado valor es una aproximación de la solución de la ecuación con un error prefijado.

El principal resultado para asegurar la existencia de solución para la ecuación $f(x) = 0$ en un intervalo $[a, b]$, es el Teorema de Bolzano que hemos recordado más arriba. Dicho teorema asegura que si f es continua en $[a, b]$ y cambia de signo en el intervalo, entonces existe al menos una solución de la ecuación en el intervalo $[a, b]$.

Vamos a ver dos métodos que se basan en este resultado. Ambos métodos nos proporcionan un algoritmo para calcular una sucesión de aproximaciones, y condiciones sobre la función f para poder asegurar que la sucesión que obtenemos converge a la solución del problema. Una vez asegurada esta convergencia, bastará tomar alguno de los términos de la sucesión que se aproxime a la sucesión con la exactitud que deseemos.

5.3.1 Bucles

Antes de introducirnos en el método teórico de resolución, vamos a presentar algunas estructuras sencillas de programación que necesitaremos más adelante.

La primera de las órdenes que vamos a ver es el comando `for`, usada para realizar bucles. Un bucle es un proceso repetitivo que se realiza un cierto número de veces. Un ejemplo de bucle puede ser el siguiente: supongamos que queremos obtener los múltiplos de siete comprendidos entre 7 y 70; para ello, multiplicamos 7 por cada uno de los números naturales comprendidos entre 1 y 10, es decir, repetimos 10 veces la misma operación: multiplicar por 7.

`for` *var:valor1* `step` *valor2* `thru` *valor3* `do` *expr*	bucle `for`
`for` *var:valor1* `step` *valor2* `while` *cond* `do` *expr*	bucle `for`
`for` *var:valor1* `step` *valor2* `unless` *cond* `do` *expr*	bucle `for`

En un bucle `for` nos pueden aparecer los siguientes elementos (no necesariamente todos)

a) *var:valor1* nos sitúa en las condiciones de comienzo del bucle.
b) *cond* dirá a *Maxima* el momento de detener el proceso.
c) *step valor2* expresará la forma de aumentar la condición inicial.
d) *expr* dirá a *Maxima* lo que tiene que realizar en cada paso; *expr* puede estar compuesta de varias sentencias separadas mediante punto y coma.

En los casos en que el paso es 1, no es necesario indicarlo.

`for` *var:valor1* `thru` *valor3* `do` *expr*	bucle `for` con paso 1
`for` *var:valor1* `while` *cond* `do` *expr*	bucle `for` con paso 1
`for` *var:valor1* `unless` *cond* `do` *expr*	bucle `for` con paso 1

Para comprender mejor el funcionamiento de esta orden vamos a ver algunos ejemplos sencillos.

En primer lugar, generemos los múltiplos de 7 hasta 70:

```
(%i43)   for i:1 step 1 thru 10 do print(7*i)
         7
         14
         21
         28
         35
         42
         49
         56
         63
         70
(%o43)   done
```

Se puede conseguir el mismo efecto sumando en lugar de multplicando. Por ejemplo, los múltiplos de 5 hasta 25 son

```
(%i44)   for i:5 step 5 thru 25 do print(i);
         5
         10
         15
         20
         25
(%o44)   done
```

Ejemplo 5.4. Podemos utilizar un bucle para sumar una lista de números pero nos hace falta una variable adicional en la que ir guardando las sumas parciales que vamos obteniendo. Por ejemplo, el siguiente código suma los cuadrados de los 100 primeros naturales.

```
(%i45)  suma:0$
        for i:1 thru 100 do suma:suma+i^2$
        print("la suma de los cuadrados de los 100 primeros
          naturales vale ",suma);

(%o45)  la suma de los cuadrados de los 100 primeros
        naturales vale 338350
```

◁

`print(`*expr1*`,`*expr2*`,...)`	escribe las expresiones en pantalla

En la suma anterior hemos utilizado la orden `print` para escribir el resultado en pantalla. La orden `print` admite una lista, separada por comas, de literales y expresiones.

Por último, comentar que no es necesario utilizar una variable como contador. Podemos estar ejecutando una serie de expresiones mientras una condición sea cierta (bucle `while`) o mientras sea falsa (bucle `unless`). Incluso podemos comenzar un bucle infinito con la orden `do`, sin ninguna condición previa, aunque, claro está, en algún momento tendremos que ocuparnos nosotros de salir (recuerda el comando `return`).

Este tipo de construcciones son útiles cuando no sabemos cuántos pasos hemos de dar pero tenemos clara cuál es la condición de salida. Veamos un ejemplo bastante simple: queremos calcular $\cos(x)$ comenzando en $x = 0$ e ir aumentando de 0.3 en 0.3 hasta que el coseno deje de ser positivo.

while *cond* do *expr*	bucle while
unless *cond* do *expr*	bucle unless
do *expr*	bucle for
return (*var*)	bucle for

```
(%i46)  i:0;
(%o46)  0
(%i47)  while cos(i)>0 do (print([i,cos(i)]),i:i+0.3);
        0 1
        [0.3, 0.95533648560273]
        [0.6, 0.82533560144755]
        [0.9, 0.62160994025671]
        [1.2, 0.36235771003359]
        [1.5, 0.070737142212368]
(%o47)  done
```

5.3.2 Condicionales

La segunda sentencia es la orden condicional `if`. Esta sentencia comprueba si se verifica una condición, después, si la condición es verdadera *Maxima* ejecutará una *expresión1*, y si es falsa ejecutará otra *expresión2*.

`if` *condición* `then` *expr1* `else` *expr2*	condicional if-then-else
`if` *condición* `then` *expr*	condicional if-then

La expresiones 1 y 2 pueden estar formadas por varias órdenes separadas por comas. Como siempre en estos casos, quizá un ejemplo es la mejor explicación:

```
(%i48)   if log(2)<0 then x:5 else 3;
(%o48)   3
```

Observa que la estructura if-then-else devuelve la expresión correspondiente y que esta expresión puede ser una asignación, algo más complicado o algo tan simple como "3".

La última sentencia de programación que vamos a ver es la orden `return(var)` cuya única finalidad es la de interrumpir un bucle en el momento que se ejecuta y devolver un valor. En el siguiente ejemplo se puede comprender rápidamente el uso de esta orden.

```
(%i49)   for i:1 thru 10 do
           (
           if log(i)<2 then print("el logaritmo de",i,"es menor
             que 2")   else return(x:i)
           )$
         print("el logaritmo de",x,"es mayor que 2")$
         el logaritmo de 1 es menor que 2
         el logaritmo de 2 es menor que 2
         el logaritmo de 3 es menor que 2
         el logaritmo de 4 es menor que 2
         el logaritmo de 5 es menor que 2
         el logaritmo de 6 es menor que 2
         el logaritmo de 7 es menor que 2
         el logaritmo de 8 es mayor que 2
```

Observación 5.5. La variable que se utiliza como contador, `i` en el caso anterior, es siempre local al bucle. No tiene ningún valor asignado fuera de él. Es por esto que hemos guardado su valor en una variable auxiliar, `x`, para poder usarla fuera del bucle.

5.3.3 Ejercicios

Ejercicio 5.4. Usa el comando `for` en los siguientes ejemplos:

a) Sumar los números naturales entre 400 y 450.
b) Calcula la media de los cuadrados de los primeros 1000 naturales.

Ejercicio 5.5. Dado un número positivo x, se puede conseguir que la suma

$$1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$$

sea mayor que x tomando un número n suficientemente grande. Encuentra la forma de calcular dicho número de manera general. ¿Cuál es el valor para $x=10,11$ y 13?

Ejercicio 5.6. Calcula las medias geométricas de los n primeros naturales y averigua cuál es el primer natural para el que dicha media sea mayor que 20.

Ejercicio 5.7. Calcula la lista de los divisores de una natural n.

5.4 Método de bisección

El método de bisección es una de las formas más elementales de buscar una solución de una ecuación. Ya sabemos que si una función continua cambia de signo en un intervalo, entonces se anula en algún punto. ¿Cómo buscamos dicho punto?

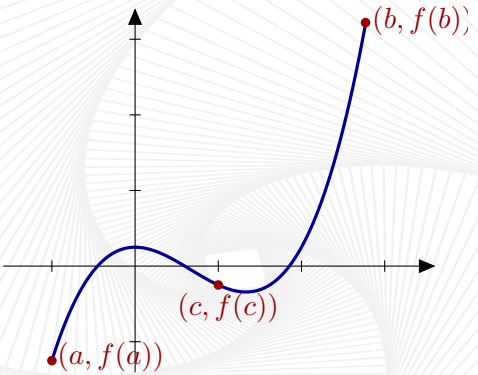


Figura 5.2 Método de bisección

Comencemos con una función $f : [a, b] \to \mathbb{R}$ continua y verificando que tiene signos distintos en los extremos del intervalo. La idea básica es ir dividiendo el intervalo por la mitad en dos subintervalos, $[a, \frac{1}{2}(a+b)]$ y $[\frac{1}{2}(a+b), b]$, y elegir aquel en el que la función siga cambiando de signo. Si repetimos este proceso, obtenemos un intervalo cada vez más pequeño donde se encuentra la raíz.

Más concretamente el proceso sería,

Datos iniciales: función f, números a, b
Se verifica que: $f(a)f(b) < 0$

bucle

- ▷ Calculamos el punto medio $c = (a + b)/2$,
- ▷ Comparamos los signos de $f(a)$, $f(c)$ y $f(b)$
- ▷ y elegimos aquél donde la función cambie de signo

final bucle

5.4.1 Un ejemplo concreto

Vamos a aplicar este método a la función $f(x) = x^6 + x - 5$ en el intervalo $[0, 2]$.

Definiremos en primer lugar la función y el intervalo y luego un bucle que nos calcula

```
(%i50)    f(x):=x^6+x-5;
          a:0.0;
          b:2.0;
(%o51)    x^6+x-5
(%o52)    0.0
(%o53)    2.0
```

Observa que hemos declarado a y b como valores numéricos. Comprobamos que la función cambia de signo.

```
(%i54)    f(a)*f(b)
(%o54)    -305.0
```

Ahora el bucle,[10]

```
(%i55)  for i:1 thru 10 do
          (
          c:(a+b)/2, /* calculamos el punto medio */
          if f(a)*f(c)<0 /* ¿cambia de signo en [a,c]? */
            then b:c /* elegimos [a,c] */
            else a:c, /* elegimos [c,b] */
          print(a,b) /* escribimos los resultados por pantalla */
          )$
                1.0 2.0
                1.0 1.5
                1.0 1.25
                1.125 1.25
                1.1875 1.25
(%o55)          1.21875 1.25
                1.234375 1.25
                1.2421875 1.25
                1.24609375 1.25
                1.24609375 1.248046875
```

Fíjate que ya sabemos que la solución es aproximadamente 1.24. No podemos estar seguros todavía del tercer decimal. Quizá sería mejor repetir el bucle más de diez veces, pero, ¿cuantas? Podemos establecer como control

[10] Los símbolos /* y */ indican el principio y el fin de un comentario. No se ejecutan.

que la distancia entre los extremos sea pequeña. Eso sí, habría que añadir un tope al número de pasos asegurarnos de que el bucle termina.

Observación 5.6. Si no se quiere ralentizar mucho la ejecución de este bucle y del resto de programas en el resto del tema, es conveniente trabajar en modo númerico. Recuerda que este comportamiento se controla con la variable `numer`. Puedes cambiarlo en el menú **Numérico→Conmutar salida numérica** o directamente estableciendo el valor de `numer` en verdadero.

```
(%i56)   numer:true;
(%o56)   true
```

5.4.1.1 Control del error

En este método es fácil acotar el error que estamos cometiendo. Sabemos que la función f se anula entre a y b. ¿Cuál es la mejor elección sin tener más datos? Si elegimos a la solución, teóricamente, podría ser b. El error sería en este caso $b-a$. ¿Hay alguna elección mejor? Sí, el punto medio $c=(a+b)/2$. ¿Cuál es el error ahora? Lo peor que podría pasar sería que la solución fuera alguno de los extremos del intervalo. Por tanto, el error sería como mucho $\frac{b-a}{2}$. En cada paso que damos dividimos el intervalo por la mitad y al mismo tiempo también el error cometido que en el paso n-ésimo es menor o igual que $\frac{b-a}{2^n}$.

A partir de aquí, podemos deducir el número de iteraciones necesarias para obtener una aproximación con un error o exactitud prefijados. Si notamos por "`err`" a la exactitud prefijada, entonces para conseguir dicha precisión, el número "n" de iteraciones necesarias deberá satisfacer

$$\frac{b-a}{2^n} \leq err$$

así,

$$n \geq \log_2\left(\frac{b-a}{err}\right) = \frac{\log\left(\frac{b-a}{err}\right)}{\log(2)}.$$

`ceiling(a)`	menor entero mayor o igual que a

La orden `ceiling(x)` nos da el menor entero mayor o igual que x. Bueno, ya sabemos cuántos pasos tenemos que dar. Reescribimos nuestro algoritmo con esta nueva información:

Datos iniciales: función f, números a, b, error err, contador i
Se verifica que: $f(a)f(b) < 0$
▷ Calculamos el número de pasos
para i:1 hasta número de pasos **hacer**
  ▷ Calculamos el punto medio $c = (a+b)/2$,
  ▷ Comparamos los signos de $f(a)$, $f(c)$ y $f(b)$
  ▷ y elegimos aquél donde la función cambie de signo
final bucle

Volviendo a nuestro ejemplo, nos quedaría algo así.

```
(%i57)  f(x):=x^6+x-5;
        a:0$
        b:2$
        err:10^(-6)$
        log2(x):=log(x)/log(2)$ /* hay que definir el logaritmo en base 2 */
        pasos:ceiling(log2((b-a)/err))$
        for i:1 thru pasos do
          (
          c:(a+b)/2,
          if f(a)*f(c)<0
            then b:c
            else a:c,
          print(a,b) /* escribimos los resultados por pantalla */
          )$
```

5.4.1.2 ¿Y si hay suerte?

Si encontramos la solución en un paso intermedio no habría que hacer más iteraciones. Deberíamos parar y presentar la solución encontrada. En cada paso, tenemos que ir comprobando que $f(c)$ vale o no vale cero. Podríamos comprobarlo con una orden del tipo `is(f(c)=0)`, pero recuerda que con valores numéricos esto puede dar problemas. Mejor comprobemos que es "suficientemente" pequeño.

Datos iniciales: función f, números a, b, error err, contador i, precisión pr
Se verifica que: $f(a)f(b) < 0$
▷ Calculamos el número de pasos

para i:1 hasta número de pasos **hacer**
  ▷ Calculamos el punto medio $c = (a + b)/2$,
  si f(c)<pr **entonces**
    ▷ La solución es c
  en otro caso
    ▷ Comparamos los signos de $f(a)$, $f(c)$ y $f(b)$
    ▷ y elegimos aquél donde la función cambie de signo
  final si
  ▷ La solución aproximada es c
final del bucle

En nuestro ejemplo, tendríamos lo siguiente

```
(%i58)   f(x):=x^6+x-5;
         a:0$
         b:2$
         err:10^(-6)$
         pr:10^(-5)$
         log2(x):=log(x)/log(2)$
         pasos:ceiling(log2((b-a)/err))$
         for i:1 thru pasos do
           (
           c:(a+b)/2,
           if abs(f(c))<pr
             then (print("La solucion es exacta"), return(c))
             else if f(a)*f(c)<0
               then b:c
               else a:c
           )$
         print("la solucion es ",c)$ /* aproximada o exacta, es la solución */
```

¿Se te ocurren algunas mejoras del algoritmo? Algunas ideas más:

a) el cálculo de $f(a)f(c)$ en cada paso no es necesario: si sabemos el signo de $f(a)$, sólo necesitamos saber el signo de $f(c)$ y no el signo del producto,
b) habría que comprobar que $f(a)$ y $f(b)$ no son cero (eso ya lo hemos hecho) ni están cerca de cero como hemos hecho con c.
c) Si queremos trabajar con una precisión mayor de 16 dígitos, sería conveniente utilizar numéros en coma flotante grandes.

5.4.2 Funciones y bloques

Una vez que tenemos más o menos completo el método de bisección, sería interesante tener una forma cómoda de cambiar los parámetros iniciales: la función, la precisión, los extremos, etc. Un bloque es la estructura diseñada para esto: permite evaluar varias expresiones y devuelve el último resultado salvo petición expresa.

La forma más elemental de "programa" en *Maxima* es lo que hemos hecho dentro del cuerpo del bucle anterior: entre paréntesis y separados por comas se incluyen comandos que se ejecutan sucesivamente y devuelve como salida la respuesta de la última sentencia.

```
(%i59)   (a:3,b:2,a+b);
(%o59)   5
```

5.4.2.1 Variables y funciones locales

Es conveniente tener la precaución de que las variables que se utilicen sean locales a dicho programa y que no afecten al resto de la sesión. Esto se consigue agrupando estas órdenes en un bloque

```
(%i60)  a:1;
(%o60)  1
(%i61)  block([a,b],a:2,b:3,a+b);
(%o61)  5
(%i62)  a;
(%o62)  1
```

Como puedes ver, la variable a global sigue valiendo uno y no cambia su valor a pesar de las asignaciones dentro del bloque. Esto no ocurre con las funciones que definamos dentro de un bloque. Su valor es global a menos que lo declaremos local explícitamente con la sentencia `local`. Observa la diferencia entre las funcione f y g.

```
(%i63)  block([a,b],
          local(g),g(x):=x^3,
          f(x):=x^2,
          a:2,b:3,g(a+b));
(%o63)  125
```

Si preguntamos por el valor de f o de g fuera del bloque, f tiene un valor concreto y g no:

```
(%i64)  f(x);
(%o64)  x^2
```

```
(%i65)   g(x);
(%o65)   g(x)
```

`local(`*`funciones`*`)`	declara funciones locales a un bloque
`return(`*`expr`*`)`	detiene la ejecución de un bloque y devuelve *expr*
`block([`*`var1,var2,..`*`],`*`expr1,expr2,..`*`)`	evalúa *expr1, expr2,...* y devuelve la última expresión evaluada

El último paso suele ser definir una función que permite reutilizar el bloque. Por ejemplo, el factorial de un número natural n se define recursivamente como

$$1! = 1, \quad (n+1)! = (n+1) \cdot n!.$$

Podemos calcular el factorial de un natural usando un bucle: usaremos la variable f para ir acumulando los productos sucesivos y multiplicamos todos los naturales hasta llegar al pedido.

```
(%i66)   fact(n):=block([f:1], for k:1 thru n do f:f*k,f );
(%o66)   fact(n):=block([f:1],for k thru n do f:f*k,f)
(%i67)   fact(5);
(%o67)   120
```

5.4.2.2 ¿Y si quiero acabar antes?

Si queremos salir de un bloque y devolver un resultado antes de llegar a la última expresión, podemos usar la orden `return`. Por ejemplo, recuerda la definición que hicimos en el primer tema de la función logaritmo con base arbitraria.

```
(%i68)   loga(x):=log(x)/log(a)$
```

Esto podemos mejorarlo algo utilizando dos variables:

```
(%i69)   loga(x,a):=log(x)/log(a)$
```

pero deberíamos tener en cuenta si a es un número que se puede tomar como base para los logaritmos. Sólo nos valen los números positivos distintos de 1. Vamos a utilizar un bloque y un condicional.

```
(%i70)   loga(x,a):=block(
           if a<0 then print("La base es negativa"),
           if a=1 then print("La base es 1"),
           log(x)/log(a)
         )$
```

Si probamos con números positivos

```
(%i71)   loga(3,4);
```

```
(%o71)   log(3)
         ------
         log(4)
```

Funciona. ¿Y si la base no es válida?

```
(%i72)   log(3,-1);
         No calculamos logaritmos con base 1
(%o72)   log(3)
         ------
         log(-1)
```

Fíjate que no hemos puesto ninguna condición de salida en el caso de que la base no sea válida. Por tanto, *Maxima* evalúa una tras otra cada una de las sentencias y devuelve la última. Vamos a arreglarlo.

```
(%i73)   loga(x,a):=block(
           if a<0 then
             (print("La base es negativa"),return()),
           if a=1 then
             (print("La base es 1"),return()),
           log(x)/log(a)
         )$
```

5.4.2.3 Parámetros opcionales

Para redondear la definición de la función logaritmo con base cualquiera, podría ser interesante que la función "loga" calcule el logaritmo neperiano si sólo ponemos una variable y el logaritmo en base a si tenemos dos variables.

Las entradas opcionales se pasan a la definición de una función entre corchetes. Por ejemplo, la función

```
(%i74)   f(a,[b]):=block(print(a),print(b))$
```

da por pantalla la variable a y el parámetro o parámetros adicionales que sean. Si solo escribimos una coordenada

```
(%i75)   f(2);
         2
         []
(%o75)   []
```

nos devuelve la primera entrada y la segunda obviamente vacía en este caso. Pero si añadimos una entrada más

```
(%i76)   f(2,3);
         2
         [3]
(%o76)   [3]
```

o varias

```
(%i77)   f(2,3,4,5);
         2
         [3,4,5]
(%o77)   [3,4,5]
```

Como puedes ver, "[b]" en este caso representa una lista en la que incluimos todos los parámetros opcionales que necesitemos. Ahora sólo es cuestión de utilizar las sentencias que nos permiten manejar los elementos de una lista para definir la función logaritmo tal y como queríamos.

```
(%i78)   loga(x,[a]):=block([res], /* una variable para el resultado */
           if length(a)=0
             then return(res:log(x))
             else (
               if a[1]<0 then (print("La base es negativa"),return()),
               if a[1]=1 then (print("La base es 1"),return()),
               log(x)/log(a[1])
             )
         )$
```

Observación 5.7. Se puede salir del bloque de definición de la función usando la sentencia `error(`*`mensaje`*`)` en los casos en que la base no sea la adecuada.

```
(%i79)   loga(x,[a]):=block([res], /* una variable para el resultado */
           if length(a)=0
             then return(res:log(x))
             else (
               if a[1]<0 then error("Cambia la base"),
               if a[1]=1 then error("La base es 1"),
               log(x)/log(a[1])
             )
         )$
```

5.4.3 De nuevo bisección

Si unimos todo lo que hemos aprendido, podemos definir una función que utilice el método de bisección. Hemos usado la sentencia `subst` para definir la función a la que aplicamos bisección dentro del bloque. La orden `subst(a,b,c)` sustituye `a` por `b` en `c`.

```
biseccion(expr,var,ext_inf,ext_sup):=
    block(
        [a,b,c,k,err:10^(-8),prec:10^(-9)],
        a:ext_inf,
        b:ext_sup,
        /* extremos del intervalo */

        /* número de pasos */
        local(log2,f),
```

```
        define(log2(x),log(x)/log(2)),
        define(f(x),subst(x,var,expr)),
        pasos:ceiling(log2((b-a)/err)),

       /* comprobamos las condiciones iniciales */
        if f(a)*f(b)>0 then error("Error: no hay cambio de signo"),

        /* ¿se alcanza la solución en los extremos? */
        if abs(f(a)) < prec then return(a),
        if abs(f(b)) < prec then return(b),

        for k:1 thru pasos do
           (
           c:(a+b)/2,
           if abs(f(c))< prec then return (c),
           if f(a)*f(c)< 0 then b:c else a:c
           ),
        c);
```

A partir de este momento, podemos utilizarlo usando

```
(%i80)   biseccion(x^2-2,x,0.0,3.0);
(%o80)   1.414213562384248
```

Observa que las cotas del error y la precisión la hemos fijado dentro del bloque. Prueba a añadirlo como valores opcionales.

5.4.4 Ejercicios

Prueba a cambiar la función, los extremos del intervalo (en los cuales dicha función cambia de signo), así como la exactitud exigida. Intenta también buscar un caso simple en el que se encuentre la solución exacta en unos pocos pasos. Por último, intenta usar el algoritmo anterior para calcular $\sqrt[3]{5}$ con una exactitud de 10^{-10}.

5.5 Métodos de iteración funcional

En esta sección tratamos de encontrar una solución aproximada de una ecuación de la forma $x = f(x)$. Es usual referirse a dichas soluciones como *puntos fijos* de la función f. Los puntos fijos de una función f no son más los puntos de intersección de las gráficas de la función f y de la identidad. Por ejemplo, la función de la Figura 5.3 tiene tres puntos fijos.

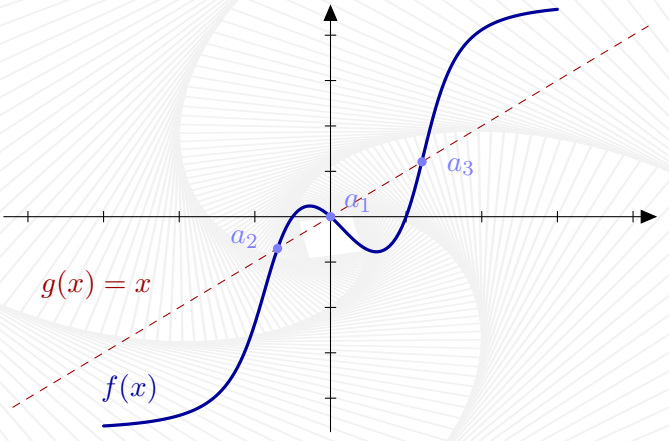


Figura 5.3 Puntos fijos de una función

Hay algunas condiciones sencillas que nos garantizan que una función tiene un único punto fijo.

Teorema 5.8. *Sea* $f : [a, b] \to [a, b]$.

a) Si f *es continua,* f *tiene al menos un punto fijo.*
b) Si f *es derivable y* $|f'(x)| \leq L < 1$, $\forall\, x \in [a, b]$, *entonces tiene un* único *punto fijo.*

Iteración funcional

Para buscar un punto fijo de una función, se elige un punto inicial $x_1 \in [a,b]$ cualquiera y aplicamos la función repetidamente. En otras palabras, consideramos la sucesión definida por recurrencia como

$$x_{n+1} = f(x_n)$$

para $n \geq 1$. Si la sucesión es convergente y llamamos s a su límite, entonces

$$s = \lim_{n\to\infty} x_{n+1} = \lim_{n\to\infty} f(x_n) = f(s),$$

o, lo que es lo mismo, s es un punto fijo de la función f. Más aún, en algunos casos es posible controlar el error.

Teorema 5.9. *Sea $f : [a,b] \to [a,b]$ derivable verificando que $|f'(x)| \leq L < 1$, $\forall\, x \in [a,b]$. Sea s el único punto fijo de la función f. Sea $x_1 \in [a,b]$ cualquiera y $x_{n+1} = f(x_n)$, entonces*

$$|x_n - s| \leq \frac{L}{1-L}\ |x_n - x_{n-1}| \leq \frac{L^{n-1}}{1-L}|x_2 - x_1|\,.$$

El método de construcción de la sucesión es lo que se conoce como un método de iteración funcional.

Ejemplo 5.10. Consideremos la función $f(x) = \frac{1}{4}\left(\cos(x) + x^2\right)$ con $x \in [0,1]$. Acotemos la derivada,

$$\left|f'(x)\right| = \left|\frac{-\operatorname{sen}(x) + 2x}{4}\right| \leq \left|\frac{\operatorname{sen}(x)}{4}\right| + \left|\frac{2x}{4}\right| \leq \frac{1}{4} + \frac{2}{4} = \frac{3}{4} < 1.$$

Por tanto, la función f tiene un único punto fijo en el intervalo $[0,1]$. Podemos encontrar el punto fijo resolviendo la correspondiente ecuación.

```
(%i81)   find_root((cos(x)+x^2)/4-x,x,0,1);
```

```
(%o81)    .2583921443715997
```

También podemos calcular las iteraciones comenzando en un punto inicial dado de manera sencilla utilizando un bucle

```
(%i82)    define(f(x),(cos(x)+x^2)/4)$
(%i83)    x0:0;
          for i:1 thru 10 do(
            x1:f(x0),print("Iteración",i,"vale", x1),x0:x1
          );
          0
          Iteración 1 vale 0.25
          Iteración 2 vale .2578531054276612
          Iteración 3 vale .2583569748525884
          Iteración 4 vale .2583898474528139
(%o83)    Iteración 5 vale .2583919943502456
          Iteración 6 vale .2583921345730372
          Iteración 7 vale .2583921437316118
          Iteración 8 vale .2583921443297992
          Iteración 9 vale .2583921443688695
          Iteración 10 vale .2583921443714213
```

◁

Ejercicio 5.8. Escribe un programa que dada una función, un punto inicial y un número de iteraciones, devuelva la última de ellas.

Observación 5.11. Existen muchas formas de cambiar una ecuación de la forma $f(x) = 0$ en un problema de puntos fijos de la forma $g(x) = x$. Por ejemplo, consideremos la ecuación $x^2 - 5x + 2 = 0$.

a) Sumando x en los dos miembros

$$x^2 - 5x + 2 = 0 \iff x^2 - 4x + 2 = x,$$

y las soluciones de f son los puntos fijos de $g_1(x) = x^2 - 4x + 2$ (si los tiene).

b) Si despejamos x,

$$x^2 - 5x + 2 = 0 \iff x = \frac{x^2 + 2}{5}$$

y, en este caso, los puntos fijos de la función $g_2(x) = \frac{x^2+2}{5}$ son las soluciones buscadas.

c) También podemos despejar x^2 y extraer raíces cuadradas

$$x^2 - 5x + 2 = 0 \iff x^2 = 5x - 2 \iff x = \sqrt{5x - 2}\,.$$

En este caso, nos interesan los puntos fijos de la función $g_3(x) = \sqrt{5x - 2}$.

Como puedes ver, la transformación en un problema de puntos fijos no es única. Evidentemente, algunas de las transformaciones mencionadas antes dependen de que x sea distinto de cero, mayor o menor que $2/5$, etc. Además de eso las funciones g_i pueden tener mejores o peores propiedades, algunas verificarán las condiciones del teorema anterior y otras no.

Ejercicio 5.9. Utiliza el método de iteración con las 3 funciones anteriores empezando en cada uno de los puntos 0.5, 1.5 y 6. ¿En cuáles obtienes convergencia a un punto fijo? ¿Es siempre el mismo?

5.5.1 Representación gráfica con el paquete *dynamics*

Para representar gráficamente los puntos de la sucesión, comenzamos con el primer punto de la sucesión $(x_1, f(x_1))$ y, a partir de ese momento, nos vamos moviendo horizontalmente hasta cruzar la bisectriz y verticalmente hasta encontrar de nuevo la gráfica de la función. Más concretamente,

a) comenzamos con $(x_1, f(x_1))$;
b) nos movemos horizontalmente hasta cortar la bisectriz. El punto de corte será $(f(x_1), f(x_1))$;
c) nos movemos verticalmente hasta cortar a la gráfica de f o, lo que es lo mismo, tomamos $x_2 = f(x_1)$ y le calculamos su imagen. El punto de corte será esta vez $(x_2, f(x_2))$.
d) Repetimos.

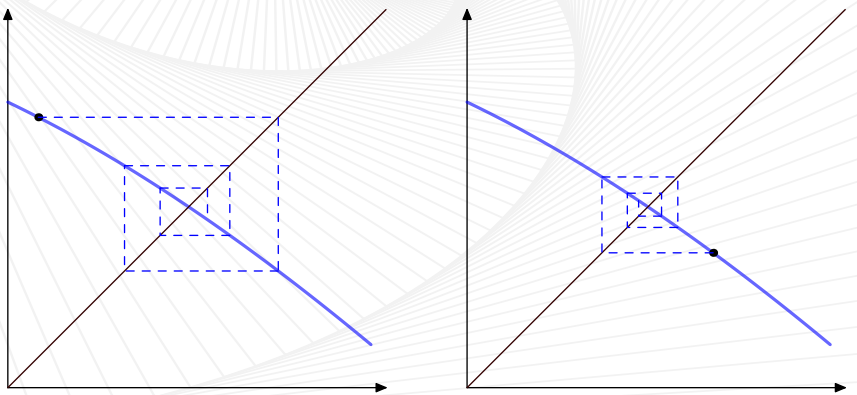


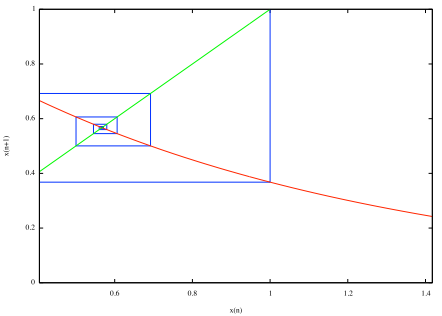
Figura 5.4 Método de iteración funcional

El paquete *dynamics* permite hacer estas representaciones de forma muy sencilla. Primero lo cargamos

```
(%i84)   load(dynamics)$
```

y luego podemos usar los comandos `evolution` y `staircase` indicando el punto inicial y el número de iteraciones. Por ejemplo, el diagrama de escalera para la función e^{-x} tomando como punto inicial 1 y 10 pasos

```
(%i85)   staircase(exp(-x),1,10,[y,0,1]);
(%o85)
```



Observa que hemos añadido `[y,0,1]` para indicar un rango más apropiado que el se dibuja por defecto.

`evolution(`*`func,pto1,pasos,opciones`*`)`	Gráfico de las iteraciones de *func*
`staircase(`*`func,pto1,pasos,opciones`*`)`	Gráfico de escalera de las iteraciones de *func*

5.5.2 Criterios de parada

Suele cuando trabajamos con métodos iterativos que tenemos una sucesión que sabemos que es convergente, pero no conocemos cuál es el valor exacto de su límite. En estos casos lo que podemos hacer es sustituir el valor desconocido del límite por uno de los términos de la sucesión que haría el papel de una aproximación de dicho límite. Por ejemplo, si consideramos el término general de una sucesión $\{a_n\}_{n\in\mathbb{N}}$ dada, con la ayuda del ordenador podemos calcular un número finito de términos. La idea es pararse en los cálculos en un determinado elemento a_{k_0} para que haga el papel del límite. Se impone entonces un *criterio de parada* para que dicho valor sea una buena aproximación del límite de la sucesión.

Una forma de establecer un criterio de parada es considerar un número pequeño, al que llamaremos *tolerancia* y denotaremos por T, y parar el desarrollo de la sucesión cuando se de una de las dos circunstancias siguientes:

a) $|a_n - a_{n-1}| < T$,

b) $\dfrac{|a_n - a_{n-1}|}{|a_n|} < T$.

La primera es el error absoluto y la segunda el error relativo. Suele ser mejor utilizar esta última.

Ejercicio 5.10. Añade una condición de parada al método de iteración.

5.5.3 Método de Newton-Raphson

El método de Newton-Rapshon nos proporciona un algoritmo para obtener una sucesión de puntos que aproxima un cero de una función dada.

La forma de construir los términos de la sucesión de aproximaciones es sencilla. Una vez fijado un valor inicial x_1, el término x_2 se obtiene como el punto de corte de la recta tangente a f en x_1 con el eje OX. De la misma

forma, obtenemos x_{n+1} como el punto de corte de la recta tangente a f en el punto x_n con el eje OX. De lo dicho hasta aquí se deduce:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Como observarás se trata de un método de iteración funcional. Para comprender el algoritmo observa la Figura 5.5 donde se ve cómo se generan los valores de las aproximaciones.

Para asegurar la convergencia de la sucesión (hacia la solución de la ecuación) usaremos el siguiente resultado.

Teorema 5.12. *Sea f una función de clase dos en el intervalo $[a, b]$ que verifica:*

a) $f(a)f(b) < 0$,
b) $f'(x) \neq 0$, *para todo* $x \in [a, b]$,
c) $f''(x)$ *no cambia de signo en* $[a, b]$.

Entonces, tomando como primera aproximación el extremo del intervalo $[a, b]$ donde f y f'' tienen el mismo signo, la sucesión de valores x_n del método de Newton-Raphson es convergente hacia la única solución de la ecuación $f(x) = 0$ en $[a, b]$.

Una vez que tenemos asegurada la convergencia de la sucesión hacia la solución de la ecuación, deberíamos decidir la precisión. Sin embargo, veremos que el método es tan rápido en su convergencia que por defecto haremos siempre 10 iteraciones. Otra posibilidad sería detener el cálculo de cuando el valor absoluto de la diferencia entre x_n y x_{n+1} sea menor que la precisión buscada (lo cual no implica necesariamente que el error cometido sea menor que la precisión).

Utilizaremos ahora *Maxima* para generar la sucesión de aproximaciones. Resolvamos de nuevo el ejemplo de $x^3 - 5 = 0$ en el intervalo $[1, 3]$.

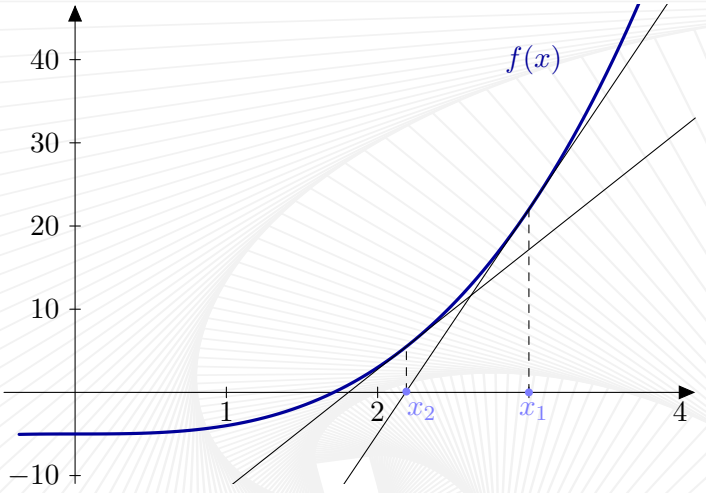
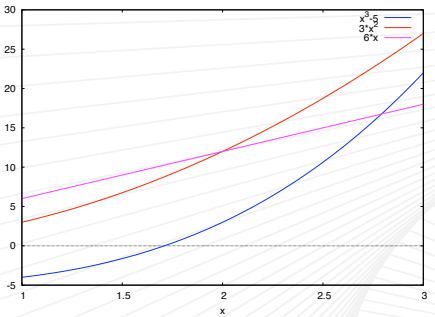


Figura 5.5 Método de Newton-Raphson

Podemos comprobar, dibujando las gráficas de $f(x) = x^3 - 5$, $f'(x)$ y $f''(x)$ en el intervalo $[1, 3]$, que estamos en las condiciones bajo las cuales el Teorema de Newton-Raphson nos asegura convergencia.

```
(%i86)   f(x):=x^3-5$
(%i87)   define(df(x),diff(f(x),x))$
(%i88)   define(df2(x),diff(f(x),x,2))$
(%i89)   plot2d([f(x),df(x),df2(x)],[x,1,3]);
```

(%o89)



A continuación, generaremos los términos de la sucesión de aproximaciones mediante el siguiente algoritmo. Comenzaremos por definir la función f y el valor de la primera aproximación. Inmediatamente después definimos el algoritmo del método de Newton-Raphson, e iremos visualizando las sucesivas aproximaciones. Como dijimos, pondremos un límite de 10 iteraciones, aunque usando mayor precisión decimal puedes probar con un número mayor de iteraciones.

```
(%i90)   y:3.0$
         for i:1 thru 10 do
           (y1:y-f(y)/df(y),
           print(i,"- aproximación",y1),
           y:y1
           );
         1 - aproximación 2.185185185185185
         2 - aproximación 1.80582775632091
         3 - aproximación 1.714973662124988
         4 - aproximación 1.709990496694424
         5 - aproximación 1.7099759468005
         6 - aproximación 1.709975946676697
         7 - aproximación 1.709975946676697
         8 - aproximación 1.709975946676697
         9 - aproximación 1.709975946676697
         10 - aproximación 1.709975946676697
```

Observarás al ejecutar este grupo de comandos que ya en la séptima iteración se han "estabilizado" diez cifras decimales. Como puedes ver, la velocidad de convergencia de este método es muy alta.

El módulo mnewton

El método que acabamos de ver se encuentra implementado en *Maxima* en el módulo `mnewton` de forma mucho más completa. Esta versión se puede aplicar tanto a funciones de varias variables, en otras palabras, también sirve para resolver sistemas de ecuaciones.

Primero cargamos el módulo

```
(%i91)   load(mnewton)$
```

y luego podemos buscar una solución indicando función, variable y punto inicial

```
(%i92)   mnewton(x^3-5,x,3);
(%o92)   [[x=1.7099759466676697]]
```

5.5.4 Ejercicios

Ejercicio 5.11.

El método de regula falsi o de la falsa posición es muy parecido al método de bisección. La única diferencia es que se cambia el punto medio por el punto de corte del segmento que une los puntos $(a, f(a))$ y $(b, f(b))$ con el eje de abscisas.

Escribe un programa que utilice este método. Para la función $f(x) = x^2 - 5$ en el intervalo $[0, 4]$, compara los resultados obtenidos. ¿Cuál es mejor?

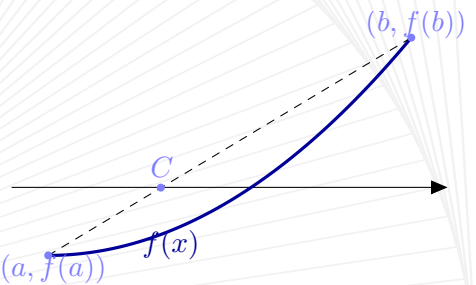


Figura 5.6 Método de regula falsi

Ejercicio 5.12. Reescribe el método de Newton-Raphson añadiendo una condición de salida (cuándo el error relativo o absoluto sea menor que una cierta cantidad) y que compruebe que la primera derivada está "lejos" de cero en cada paso.

Ejercicio 5.13.

El método de la secante evita calcular la derivada de una función utilizando recta secantes. Partiendo de dos puntos iniciales x_0 y x_1, el siguiente es el punto de corte de la recta que pasa por $(x_0, f(x_0))$ y $(x_1, f(x_1))$ y el eje de abscisas. Se repite el proceso tomando ahora los puntos x_1 y x_2 y así sucesivamente.

La convergencia de este método no está garantizada, pero si los dos puntos iniciales están próximos a la raíz no suele haber problemas. Su convergencia es más lenta que el método de Newton-Raphson aunque a cambio los cálculos son más simples.

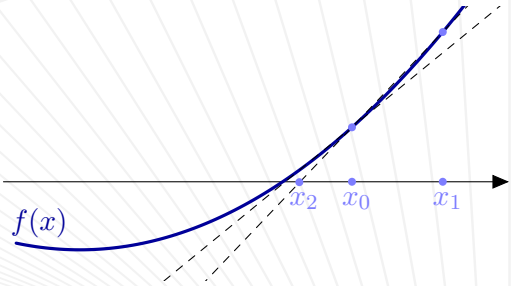


Figura 5.7 Método de la secante

Escribe un programa que utilice este método. Para la función $f(x) = x^2 - 5$, compara los resultados obtenidos con el método de Newton-Raphson. ¿Cuál es mejor?

Ejercicio 5.14. Resuelve las ecuaciones

a) $e^{-x} + x^2 - 3\,\mathrm{sen}(x) = 0$,
b) $e^{|x|} = \arctan(x)$,
c) $x^{15} - 2 = 0$

utilizando los métodos que hemos estudiado. Compara cómo se comportan y decide en cuál la convergencia es más rápida.

Ejercicio 5.15.

a) Considérese la ecuación $e^{(x^2+x+1)} - e^{x^3} - 2 = 0$. Calcular programando los métodos de bisección y de Newton-Raphson, la solución de dicha ecuación en el intervalo $[-0.3, 1]$ con exactitud 10^{-10}.
b) Buscar la solución que la ecuación $\tan(x) = \frac{1}{x}$ posee en el intervalo $[0, \frac{\pi}{2}]$ usando los métodos estudiados.

A Números complejos

Después de calcular la raíz cuadrada de 2, la primera idea que se nos ocurre a todos es probar con números negativos "a ver que pasa":

```
(%i1)   sqrt(-2);
(%o1)   √2 %i
```

Correcto. Ya habíamos comentado con anterioridad que %i representa a la unidad imaginaria. En *Maxima* podemos trabajar sin problemas con números complejos. Se pueden sumar, restar, multiplicar, dividir

```
(%i2)   ((2+3*%i)+(3-%i);
(%o2)   2 %i+5
```

Si multiplicamos o dividimos números complejos, hemos visto que *Maxima* no desarrolla completamente el resultado, pero si pedimos que nos lo de en forma cartesiana, por ejemplo,

```
(%i3)   (2+3*%i)/(1-%i);
(%o3)   (3 %i+2)/(1-%i)
(%i4)   rectform((2+3*%i)/(1-%i));
(%o4)   5 %i/2 - 1/2
```

`rectform(`*expresión*`)`	*expresión* en forma cartesiana o binómica
`realpart(`*expresión*`)`	parte real de *expresión*
`imagpart(`*expresión*`)`	parte imaginaria de *expresión*
`polarform(`*expresión*`)`	forma polar de *expresión*
`abs(`*expresión*`)`	módulo o valor absoluto de *expresión*
`cabs(`*expresión*`)`	módulo de *expresión* compleja
`carg(`*expresión*`)`	argumento de *expresión*
`conjugate(`*expresión*`)`	conjugado de *expresión*
`demoivre(`*expresión*`)`	expresa el número complejo utilizando senos y cosenos
`exponentialize(`*expresión*`)`	expresa el número complejo utilizando exponenciales

De la misma forma podemos calcular la parte real y la parte imaginaria, el módulo o la forma polar de un número complejo

```
(%i5)  realpart(2-%i);
(%o5)  2
(%i6)  abs(1+3*%i);
(%o6)  sqrt(10)
(%i7)  polarform(1+3*%i);
(%o7)  sqrt(10)%e^(%i atan(3))
```

No hace falta calcular la forma polar para conocer el argumento principal de un número complejo, carg se encarga de de ello:

```
(%i8)   carg(1+3*%i);
(%o8)   atan(3)
(%i9)   carg(exp(%i));
(%o9)   1
```

Recuerda que muchas funciones reales tienen una extensión al plano complejo. Por ejemplo, exp nos da la exponencial compleja,

```
(%i10)   exp(%i*%pi/4);
(%o10)   sqrt(2)%i/2 + sqrt(2)/2
```

log nos da el logaritmo principal

```
(%i11)   log(%i);
(%o11)   log(%i)
(%i12)   log(-3);
(%o12)   log(-3)
```

siempre que se lo pidamos

```
(%i13)  rectform(log(%i));
(%o13)  %i π/2
(%i14)  rectform(log(-3));
(%o14)  log(3)+%i π
```

Podemos calcular senos o cosenos,

```
(%i15)  cos(1+%i);
(%o15)  cos(%i+1)
(%i16)  rectform(%);
(%o16)  cos(1)cosh(1)-%i sin(1)sinh(1)
```

si preferimos la notación exponencial, `exponentialize` escribe todo en términos de exponenciales

```
(%i17)  exponentialize(%);
(%o17)  (%e+%e^-1)(%e^%i+%e^-%i)/4 - (%e-%e^-1)(%e^%i-%e^-%i)/4
```

y demoivre utiliza senos y cosenos en la salida en lugar de las exponenciales:

```
(%i18)  demoivre(%);
(%o18)  (%e+%e^-1) cos(1)/2 - (%e-%e^-1) i sin(1)/2
```

Observación A.1. La orden abs calcula el módulo de la expresión pedida como ya hemos comentado

```
(%i19)  abs(1+%i);
(%o19)  sqrt(2)
```

pero en algunas ocasiones el resultado no es el esperado

```
(%i20)  abs(log(-3));
(%o20)  -log(-3)
```

En este caso, cabs nos da el resultado correcto

```
(%i21)  cabs(log(-3));
(%o21)  sqrt(log(3)^2+%pi^2)
```

B Avisos y mensajes de error

Este apéndice no es, ni pretende ser, un repaso exhaustivo de los errores que podemos cometer utilizando *Maxima*, sino más bien una recopilación de los errores más comunes que hemos cometido mientras aprendíamos a usarlo. La lista no está completa y habría que añadirle la manera más típica de meter la pata: inventarse los comandos. Casi todos utilizamos más de un programa de cálculo, simbólico o numérico, y algún lenguaje de programación. Es muy fácil mezclar los corchetes de *Mathematica* (© by Wolfram Research) y los paréntesis de *Maxima*. ¿Se podía utilizar `ln` como logaritmo neperiano o eso era en otro programa? Muchas veces utilizamos lo que creemos que es una función conocida por *Maxima* sin que lo sea. En esos casos un vistazo a la ayuda nos puede sacar de dudas. Por ejemplo, ¿qué sabe *Maxima* sobre `ln`?

```
(%i1)   ??ln  - Función: belln (<n>)
          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 ...
```

No es precisamente lo que esperábamos (al menos yo).

```
(%i2)   2x+1
        Incorrect syntax: X is not an infix operator
        2x+
          ^
```

Es necesario escribir el símbolo de multiplicación, $*$ entre 2 y x.

```
(%i3)   factor((2 x+1)^2)
        Incorrect syntax: X is not an infix operator
        factor((2Spacex+
                  ^
```

Es necesario escribir el símbolo de multiplicación, $*$ entre 2 y x. No es suficiente con un espacio en blanco.

```
(%i4)   plot2d(sin(x),[x,0,3]])
        Incorrect syntax: Missing )
        ot2d(sin(x),[x,0,3]]
                           ^
```

Repasa paréntesis y corchetes: en este caso hay un corchete de más.

```
(%i5)   g(x,y,z):=(2*x,3*cos(x+y))$
(%i6)   g(1,%pi);
        Too few arguments supplied to g(x,y,z):
        [1,π]
        - an error. To debug this try debugmode(true);
```

La función tiene tres variables y se la estamos aplicando a un vector con sólo dos componentes.

```
(%i7)   f(x):3*x+cos(x)
        Improper value assignment:
        f(x)
        - an error. To debug this try debugmode(true);
```

Para definir funciones se utiliza el signo igual y dos puntos y no sólo dos puntos.

```
(%i8)   solve(sin(x)=0,x);
        'solve' is using arc-trig functions to
        get a solution. Some solutions will be lost.
(%o8)   [x=0]
```

Para resolver la ecuación $\mathrm{sen}(x) = 0$ tiene que usar la función arcoseno (su inversa) y *Maxima* avisa de que es posible que falten soluciones y, de hecho, faltan.

```
(%i9)   integrate(1/x,x,0,1);
        Is x + 1 positive, negative, or zero? positive;
        Integral is divergent
        - an error. To debug this try debugmode(true);
```

La función $\frac{1}{x}$ no es integrable en el intervalo $[0, 1]$.

```

```

```
(%i10)  find_root(x^2,-3,3);
        function has same sign at endpoints
        [f(-3.0)=9.0,f(3.0)=9.0]
        - an error. To debug this try debugmode(true);
```

Aunque x^2 se anula entre 3 y -3 (en $x = 0$ obviamente) la orden `find_root` necesita dos puntos en los que la función cambie de signo.

```
(%i11)  A:matrix([1,2,3],[2,1,4],[2.1,4]);
        All matrix rows are not of the same length.
        - an error. To debug this try debugmode(true);
```

En una matriz, todas las filas deben tener el mismo número de columnas: hemos escrito un punto en lugar de una coma en la última fila.

```
(%i12)  A:matrix([1,2,3],[2,1,4],[2,1,4])$
(%i13)  B:matrix([1,2],[2,1],[3,1])$
(%i14)  B.A
        incompatible dimensions - cannot multiply
        - an error. To debug this try debugmode(true);
```

No se pueden multiplicar estas matrices. Repasa sus órdenes.

C Bibliografía

a) La primera fuente de documentación sobre *Maxima* es el propio programa. La ayuda es muy completa y detallada.

b) En la página del programa *Maxima*, http://maxima.sourceforge.net/es/, existe una sección dedicada a documentación y enlaces a documentación. El manual de referencia de *Maxima* es una fuente inagotable de sorpresas. Para cualquier otra duda, las listas de correo contienen mucha información y, por supuesto, siempre se puede pedir ayuda.

c) "*Primeros pasos en Maxima*" de Mario Rodríguez Riotorto es, junto con la siguiente referencia, la base de estas notas. Se puede encontrar en

http://www.telefonica.net/web2/biomates/

d) "*Maxima con wxMaxima: software libre en el aula de matemáticas*" de Rafael Rodríguez Galván es el motivo de que hayamos usado *wxMaxima* y no cualquier otro entorno sobre *Maxima*. Este es un proyecto que se encuentra alojado en el repositorio de software libre de RedIris

https://forja.rediris.es/projects/guia-wxmaxima/

e) Edwin L. Woollett está publicando en su página, capítulo a capítulo unas notas (más bien un libro) sobre *Maxima* y su uso. El título lo dice todo: "*Maxima by example*". Su página es

http://www.csulb.edu/ woollett/

f) Esta bibliografía no estaría completa sin mencionar que la parte más importante de esto sigue siendo la asignatura de Cálculo. La bibliografía de ésta ya la conoces.

Glosario