

# Prácticas de ordenador con Maxima

---

---

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



---

# Índice

---

## Introducción iii

### 1 Primeros pasos 5

1.1 Introducción 5 1.2 Resultados exactos y aproximación decimal 9 1.3 Funciones usuales 11 1.4 Operadores lógicos y relacionales 15 1.5 Variables 16 1.6 Expresiones simbólicas 19 1.7 La ayuda de *Maxima* 25 1.8 Ejercicios 28

### 2 Gráficos 29

2.1 Funciones 29 2.2 Gráficos en el plano con `plot2d` 33 2.3 Gráficos en 3D 43 2.4 Gráficos con `draw` 46 2.5 Animaciones gráficas 61 2.6 Ejercicios 65

### 3 Listas y matrices 67

3.1 Listas 67 3.2 Matrices 72 3.3 Ejercicios 79

### 4 Números complejos. Resolución de ecuaciones 83

4.1 Números complejos 83 4.2 Ecuaciones y operaciones con ecuaciones 85 4.3 Resolución de ecuaciones 86 4.4 Hágalo usted mismo 93 4.5 Ejercicios 100

### 5 Límites y continuidad 103

5.1 Límites 103 5.2 Sucesiones 105 5.3 Continuidad 107 5.4 Ejercicios 108

### 6 Derivación 111

6.1 Cálculo de derivadas 111 6.2 Rectas secante y tangente a una función 114 6.3 Máximos y mínimos relativos 117 6.4 Polinomio de Taylor 122 6.5 Ejercicios 126

### 7 Series numéricas y series de Taylor 127

7.1 Series numéricas 127 7.2 Desarrollo de Taylor 129 7.3 Ejercicios 130

### 8 Integración 133

8.1 Cálculo de integrales 133 8.2 Sumas de Riemann 139 8.3 Aplicaciones 142 8.4 Ejercicios 148

### 9 Diferenciación 151

9.1 Derivadas parciales de una función 151 9.2 Curvas y vectores tangentes 154 9.3 Funciones definidas implícitamente 156 9.4 Extremos relativos 161 9.5 Extremos condicionados 163 9.6 Extremos absolutos 165 9.7 Ejercicios 166

### 10 Integrales múltiples 169

10.1 Ejercicios 173

### 11 Ecuaciones diferenciales ordinarias 175

11.1 Resolución de edos de primer y segundo orden 175 11.2 Resolución de sistemas de ecuaciones lineales 178 11.3 Campos de direcciones y curvas integrales 179 11.4 Ejercicios 183

A Avisos y mensajes de error 185

B Bibliografía 189

Índice alfabético 191

---

# 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 MacOSX.

## 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. Sobre la instalación en alguna distribución Linux es mejor consultar la ayuda sobre su correspondiente programa para gestionar software.

## Para acabar

Las matemáticas y el software libre comparten la filosofía de que el conocimiento debe ser libre. Sólo los resultados publicados y revisados por pares tienen el reconocimiento de la comunidad científica. Educados en esta forma de pensar, es lógico que utilicemos herramientas que estén basadas en los mismos principios.

*Granada a 10 de septiembre de 2008*

# Primeros pasos

## 1

### 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):

```
wxMaxima 0.7.5 http://wxmaxima.sourceforge.net
Maxima 5.15.0 http://maxima.sourceforge.net
Using Lisp GNU Common Lisp (GCL) GCL 2.6.8 (aka GCL)
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.
```

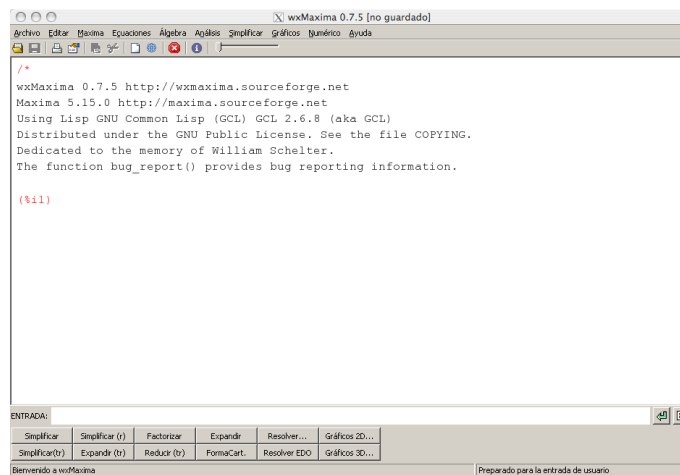


Figura 1.1 Ventana inicial de *wxMaxima*

Ya iremos comentando con mayor profundidad los distintos menús y opciones que se nos presentan pero antes de ir más lejos, ¿podemos escribir algo? En la parte inferior de la ventana tienes una línea en blanco etiquetada “Entrada”. Sitúa ahí el cursor y escribe  $2+3$ . Luego pulsa la tecla **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.

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

- 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).
- Las entradas terminan en punto y coma. `wxMaxima` lo añade si tú te has olvidado de escribirlo. Justamente lo que nos había pasado.

Puedes volver a editar cualquiera de las entradas anteriores. Para ello utiliza las flechas  $\uparrow$  y  $\downarrow$  para moverte a una entrada anterior o posterior.

## 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

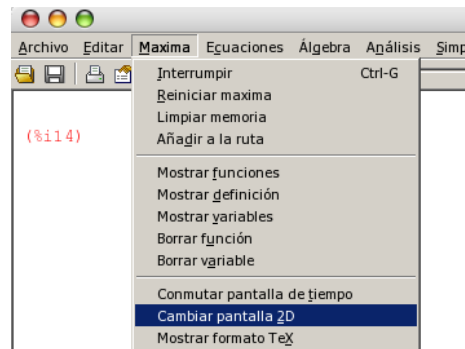
```
(%i6) 3^57;
```



```
(%o6) 1570042899082081611640534563
```

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;
13220708194808066368904552597
(%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;
132207081948080663689045525975214436596542203275214816766492036
822682859734670489954077831385060806196390977769687258235595095
458210061891186534272525795367402762022519832080387801477422896
484127439040011758861804112894781562309443806156617305408667449
(%o9) 050617812548034440554705439703889581746536825491613622083026856
377858229022841639830788789691855640408489893760937324217184635
993869551676501894058810906042608967143886410281435038564874716
5832010614366132173102768902855220001
```

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) 132207081948080663689045525975[418 digits]614366132173102768902
855220001
```

**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) 32
        81
```

## 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
```

Raíces

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) √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) √5
```

float

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.

<code>%pi</code>	el número $\pi$
<code>%e</code>	el número $e$
<code>%i</code>	la unidad imaginaria
<code>%phi</code>	la razón áurea, $\frac{1+\sqrt{5}}{2}$

Podemos operar con ellas como con cualquier otro número.

<code>(%i18)</code>	<code>(2+3*%i)*(5+3*%i);</code>
<code>(%o18)</code>	<code>(3*%i+2)*(3*%i+5)</code>

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?

<code>%</code>	último resultado
<code>%i número</code>	entrada <i>número</i>
<code>%o número</code>	resultado <i>número</i>

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

<code>(%i19)</code>	<code>%o15</code>
<code>(%o19)</code>	$\sqrt{5}$

además podemos usar esa información como cualquier otro dato.

<code>(%i20)</code>	<code>%o4+%o5;</code>
<code>(%o20)</code>	$\frac{146}{7}$

## 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  $\pi$  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) 0.833333333333333
```

`numer` 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) 0.833333333333333
(%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`.

<code>float(número)</code>	expresión decimal de <i>número</i>
<code>número, numer</code>	expresión decimal de <i>número</i>
<code>bfloat(número)</code>	expresión decimal larga de <i>número</i>

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  $\pi$ :

```
(%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 el orden `bfloat` para que *Maxima* nos muestre todos los decimales pedidos (y cambiar la pantalla a `ascii`):

```
(%i30) bfloat(%pi);
(%o30) 3.1415926535897932384626433832[43 digits]62862089986280348
25342117068b0
(%i31) set_display('ascii)$
(%i32) bfloat(%pi);
(%o32) 3.141592653589793238462643383279502884197169399375105820
974944592307816406286208998628034825342117068b0
```

Como se puede ver en la Figura 1.2, también se puede utilizar el menú **N**umérico→**A** real o **N**umérico→**A** real grande(**b**igfloat) para obtener la expresión decimal buscada.

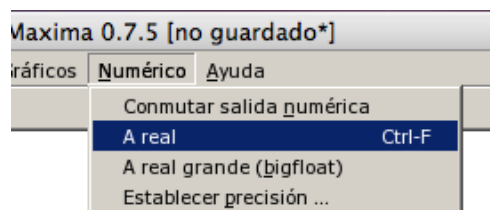


Figura 1.2 Salida numérica

### 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
```

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

## Potencias, raíces y exponenciales

Hemos visto que podemos escribir potencias utilizando  $\wedge$  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) 21/3 54/3
```

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

```
(%i36) %e^2;
(%o36) %e2
```

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

```
(%i37) exp(2);
(%o37) %e2
(%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) 2.995732273553991
```

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) 2.995732273553991
```

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) 2.995732273553991
```

¿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) 6
```

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 as  $\sin(x)$ ,  $\cos(x)$  y  $\tan(x)$ , así como las funciones recíprocas secante,  $\sec(x)$ , cosecante,  $\csc(x)$ , y cotangente,  $\cot(x)$ .

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

Asimismo, 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
```

¿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 $n$
$\text{entier}(x)$	parte entera de $x$
$\text{abs}(x)$	valor absoluto o módulo de $x$
$\text{random}(x)$	devuelve un número aleatorio
$\text{signum}(x)$	signo de $x$
$\text{max}(x_1, x_2, \dots)$	máximo de $x_1, x_2, \dots$
$\text{min}(x_1, x_2, \dots)$	mínimo de $x_1, x_2, \dots$



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
```

<code>is(<i>expresión</i>)</code>	decide si la expresión es cierta o falsa
<code>assume(<i>expresión</i>)</code>	supone que la expresión es cierta
<code>and</code>	y
<code>or</code>	o
<code>notequal</code>	distinto

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) unknown
```

=	igual
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
```

Vale, no parece muy listo. Ahora bien, hay que tener en cuenta que  $x$  podría ser un número complejo. ¿Te parece tan mala la respuesta ahora? Si nosotros disponemos de información adicional, siempre podemos “ayudar”. Por ejemplo, si sabemos que  $x$  es positivo la situación cambia:

```
(%i57) assume(x>0);
(%o57) [x>0]
(%i58) is((x+1)^2>0);
(%o58) true
```

## 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

```
(%i59) a:2
(%o59) 2
(%i60) a^2
(%o60) 4
```

<code>var: expr</code>	la variable <code>var</code> vale <code>expr</code>
<code>kill(a1, a2, ...)</code>	elimina los valores
<code>remove(var1, var2, ...)</code>	borra los valores de las variables
<code>values</code>	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.

```
(%i61) largo:10;
(%o61) 10
(%i62) ancho:7;
(%o62) 7
(%i63) largo*ancho;
(%o63) 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.

```
(%i64) solucion:solve(x^2-1=0,x);
(%o64) [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  $x$

```
(%i65) x:3;
(%o65) 3
```

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

```
(%i66) 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:

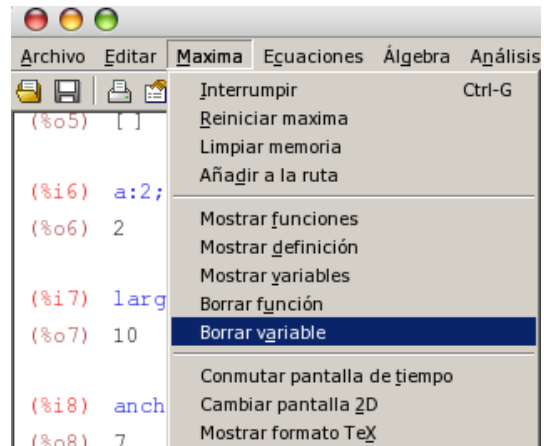
```
(%i67) diff(sin('x),'x);
(%o67) cos(x)
```

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`.

values

```
(%i68) values;
(%o68) [a,largo,ancho,x]
```



remvalue

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

```
(%i69) remvalue(a,x);
(%o69) [a,x]
```

o todas.

```
(%i70) remvalue(all);
(%o70) [largo,ancho]
```

kill

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.

```
(%i71) ancho:10$
(%i72) kill(ancho);
(%o72) done
(%i73) remvalue(ancho);
(%o73) [false]
```

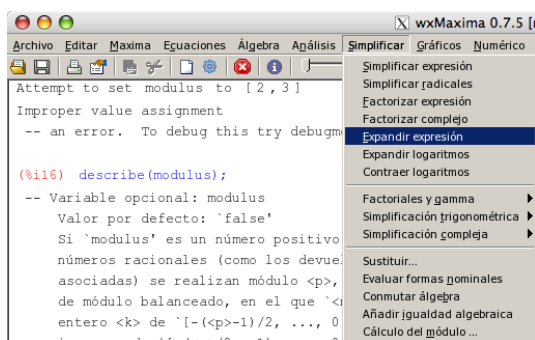
Una de las pequeñas diferencias entre `kill` y `remvalue` es que la primera no comprueba si la variable, o lo que sea, estaba previamente definida y siempre responde `done`. Existe también la posibilidad de borrar `todo`:

```
(%i74) 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) %ey+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 en la barra de botones de la parte inferior de la ventana 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 que 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?

<code>expand(expr)</code>	realiza productos y potencias
<code>partfrac(frac, var)</code>	descompone en fracciones simples
<code>num(frac)</code>	numerador
<code>denom(frac)</code>	denominador

**expand** La orden `expand` desarrollo productos, potencias,

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

y cocientes.

```
(%i9) expand(p/q);
(%o9)  $\frac{x^2}{x^2-6x+9} + \frac{x}{x^2-6x+9} - \frac{2}{x^2-6x+9}$ 
```

**partfrac** 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);
```

$$(\%o10) \quad \frac{7}{x-3} + \frac{10}{x-3^2} + 1$$

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

`num`  
`denom`

(%i11) `denom(p/q);`

(%o11)  $(x-1)(x+2)$

(%i12) `num(p/q);`

(%o12)  $(x-3)^2$

## 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.

<code>expand(expr, n, m)</code>	desarrolla potencias con grado entre $-m$ y $n$
<code>logexpand</code>	variable que controla el desarrollo de logaritmos
<code>radexpand</code>	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 a11,

```
(%i16) radexpand:a11$ logexpand:a11$
```

```
(%i17) log(a*b);
```

```
(%o17) log(a)+log(b)
```

```
(%i18) sqrt(x*y)
```

```
(%o18)  $\sqrt{x} \cdot \sqrt{y}$ 
```

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 a11.

## Factorización

<code>factor(expr)</code>	escribe la expresión como producto de factores más sencillos
---------------------------	---

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

```
(%i19) factor(100);
```

```
(%o19) 22 52
```

como con expresiones polinómicas como las anteriores

```
(%i20) factor(x^2);
```

```
(%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-3x2)
```

```
(%i22) expand(%);
```

```
(%o22) -xy2+4x2y-3x3
```

```
(%i23) factor(%);
```

```
(%o23) -x(y-3x)(y-x)
```



## Evaluación de valores en expresiones

`ev(expr, arg1, arg2, ...)` evalúa 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)  $\frac{x^2+x-2}{x^2-6*x+9}$

(%i29) `partfrac(p/q, x);`

(%o29)  $\frac{7}{x-3} + \frac{10}{x-3^2} + 1$

<code>radcan(expr)</code>	simplifica expresiones con radicales
<code>ratsimp(expr)</code>	simplifica expresiones racionales
<code>fullratsimp(expr)</code>	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 los órdenes `trigexpand`, `trigsimp` y `trigreduce`.

<code>trigexpand(expresion)</code>	desarrolla funciones trigonométricas e hipérbolicas
<code>trigsimp(expresion)</code>	simplifica funciones trigonométricas e hiperbólicas
<code>trigreduce(expresion)</code>	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) 8 * ( -frac(cos(7*x) + cos(x)) {8} - 3 * (frac(cos(5*x)} {2} + frac(cos(3*x)} {2})) + frac(cos(3*x)} {8} + 3 * frac(cos(x)} {8} )
```

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) frac(cos(2x)+1} {2}+cos(2x)
(%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)+frac(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.

<code>describe(expr)</code>	ayuda sobre <i>expr</i>
<code>example(expr)</code>	ejemplo de <i>expr</i>
<code>apropos(expr)</code>	comandos relacionados con <i>expr</i>
<code>??expr</code>	comandos que contienen <i>expr</i>
<code>demo(expr)</code>	demostración de <i>expr</i>

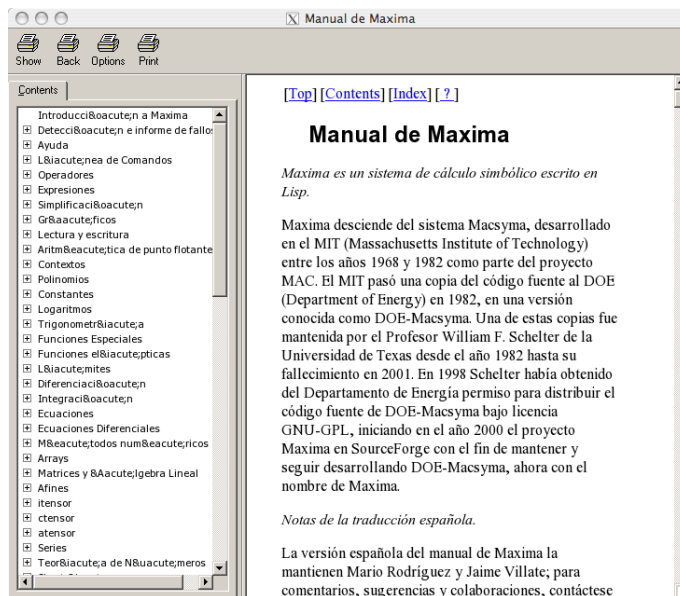


Figura 1.3 Ayuda de *wxMaxima*

En el caso de que conozcamos el nombre del comando sobre el que estamos buscando ayuda, la orden `describe` 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
```

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:

apropos

```
(%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`.

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
```

Con demo obtenemos ejemplos de órdenes pero de forma interactiva. Eso sí, sólo están disponibles sobre algunos concretos.

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

- Los 100 primeros decimales del número  $e$ ,
- el logaritmo en base 3 de 16423203268260658146231467800709255289.
- el arcocoseno hiperbólico de 1,
- el seno y el coseno de  $i$ , y
- el logaritmo de -2.

### Ejercicio 1.2

- ¿Qué número es mayor  $1000000^{999999}$  o  $999999^{1000000}$ ?
- Ordena de mayor a menor los números  $\pi$ ,  $\frac{73231844868435875}{37631844868435563}$  y  $\cosh(3)/3$ .

### Ejercicio 1.3

Descompone la fracción  $\frac{x^2-4}{x^3+x^4-2x^3-2x^2+x+1}$  en fracciones simples.

### Ejercicio 1.4

Escribe  $\sin(5x) \cos(3x)$  en función de  $\sin(x)$  y  $\cos(x)$ .

### Ejercicio 1.5

Comprueba si las funciones hiperbólicas y las correspondientes “arco”-versiones son inversas.

# Gráficos

## 2

El objetivo de este capítulo es aprender a dibujar gráficas de funciones en dos y tres dimensiones. Lo haremos, tanto para gráficas en coordenadas cartesianas, esto es, gráficas de funciones  $y = f(x)$ , 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

<code>funcion(var1,var2,..):=(expr1,expr2,...)</code>	definición de función
<code>define (func,expr)</code>	la función vale <i>expr</i>
<code>fundef(func)</code>	devuelve la definición de la función
<code>functions</code>	lista de funciones definidas por el usuario
<code>remfunction(func1,func2,...)</code>	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/√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

**define**

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`.

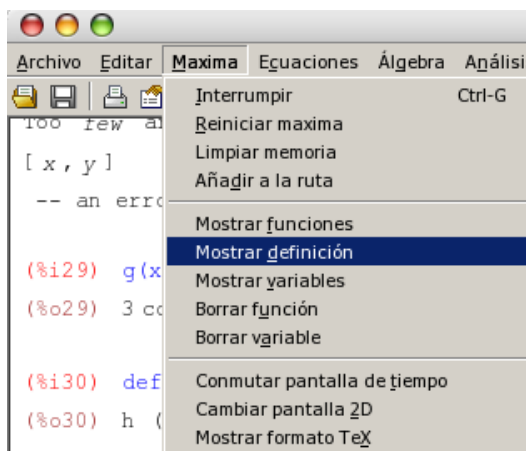


Figura 2.1 Mostrar y borrar funciones



```
(%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:

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

Este segundo método tiene el inconveniente de que las funciones definidas de esta manera no nos sirven para derivarlas o integrarlas, aunque sí podremos dibujar su gráfica. Por ejemplo, la función

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

la podemos definir como

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

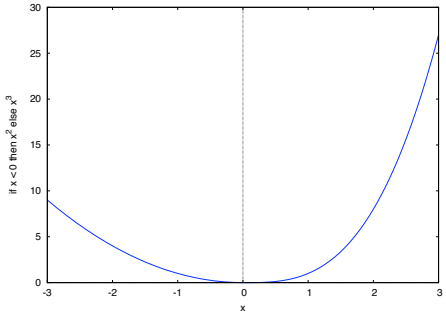
y podemos evaluarla en un punto

```
(%i14) f(-2);
(%o14) 4
```

```
(%i15) f(2);
mi (%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)  $\int_{-3}^3 \text{if } x < 0 \text{ then } x^2 \text{ else } x^3 dx$ 
```

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)  $\frac{117}{4}$ 
```

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

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

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  $x$ .

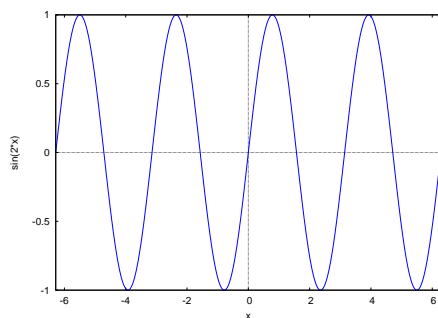
```
plot2d(f(x), [x, a, b])  gráfica de  $f(x)$  en  $[a, b]$ 
```

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

Por ejemplo:

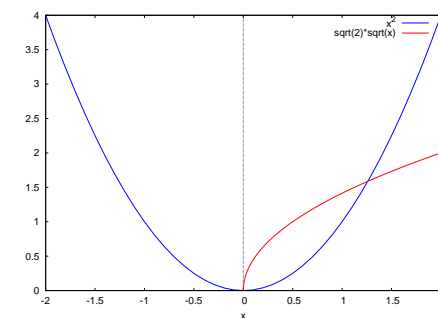
```
(%i23) plot2d(sin(2*x), [x, -2*pi, 2*pi]);
```

```
(%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.

Al comando `plot2d` se puede acceder también a través del menú **Gráficos**—**Gráficos 2D** o, directamente, a través del botón **Gráficos 2D**. Cuando lo hacemos de cualquiera de estas dos formas aparece una ventana de diálogo en la que aparecen varios datos a rellenar:

- 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 el doble de grande que en el eje X ("`set size ratio 1; set zeroaxis;`"); dibuje una malla ("`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.
- Evidentemente, estas no son todas las posibles opciones. La cantidad de posibilidades que tiene *Gnuplot* es inmensa.

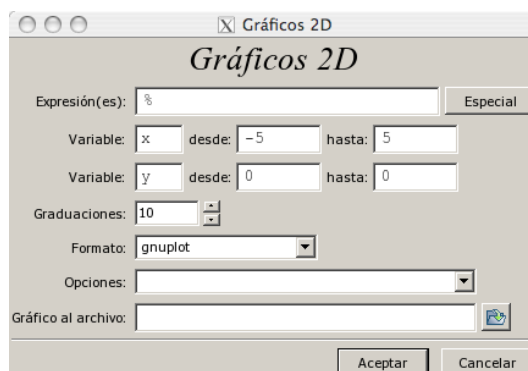
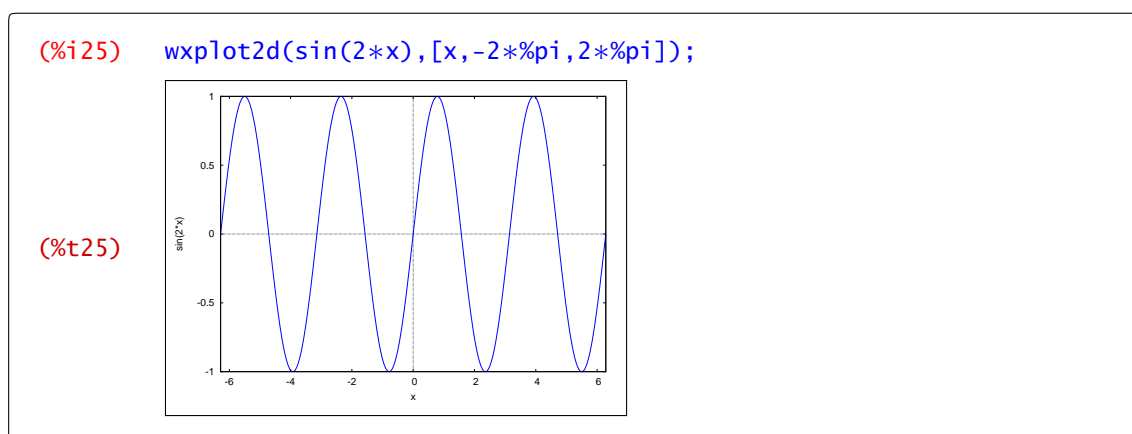


Figura 2.2 Gráficos en 2D



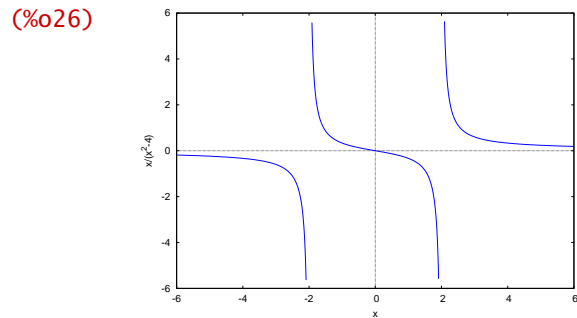
**Observación 2.1.** El prefijo “wx” añadido a `plot2d` o a cualquiera del resto de las órdenes que veremos en este capítulo (`plot3d`, `draw2d`, `draw3d`) hace que *wxMaxima* pase automáticamente a mostrar los gráficos en la misma ventana y no en una ventana separada. Es lo mismo que seleccionar en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



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

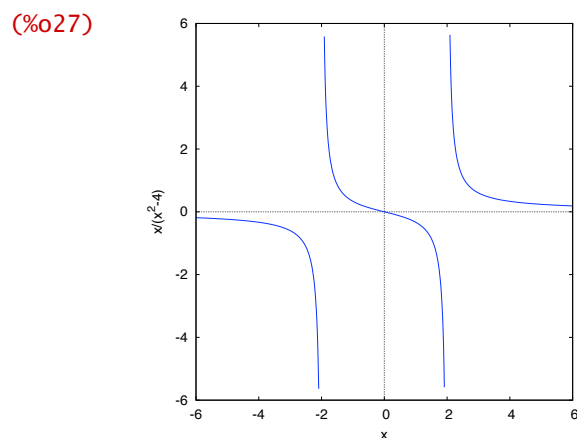
A continuación, veamos algunos ejemplos de las opciones que hemos comentado. Podemos añadir ejes,

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



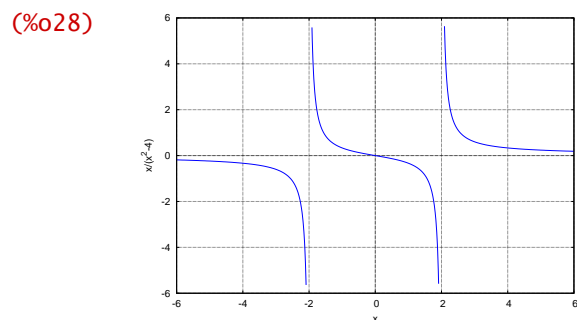
podemos cambiar la proporción entre ejes,

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



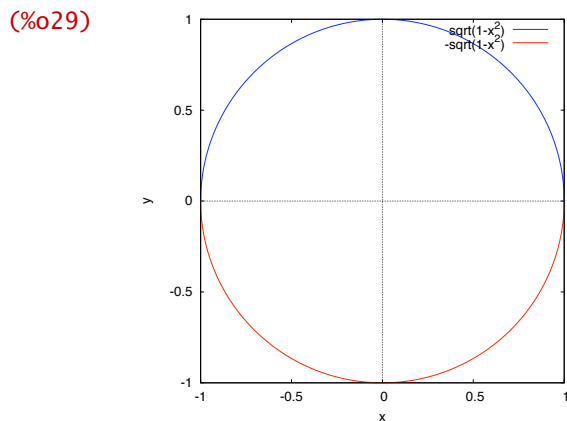
o añadir una malla que nos facilite la lectura de los valores de la función.

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



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;"]$
```

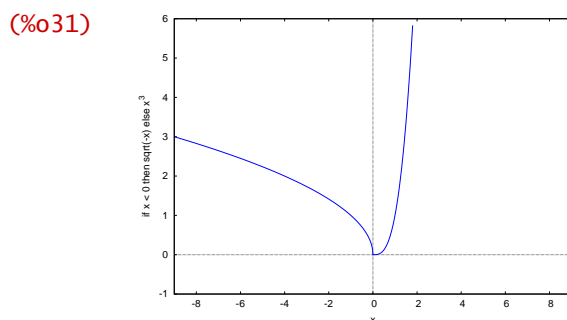


También podemos dibujar gráficas de funciones a trozos. Antes, tenemos que recordar cómo se definen estas funciones. Lo hacemos con un ejemplo. Consideremos la función  $f: \mathbb{R} \rightarrow \mathbb{R}$  definida como

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

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 sqrt(-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 y paramétricas

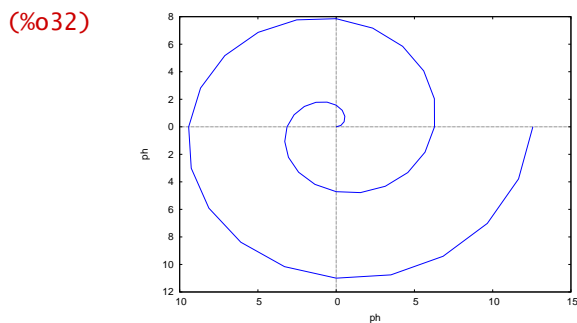
En este apartado nos dedicaremos a un tipo especial de curvas, aquellas que vienen definidas de forma paramétrica, si bien antes dedicamos unos ejemplos en los que representaremos curvas

de forma polar. Es decir, curvas planas en las que la variable independiente es el ángulo  $\theta$  y que aquí escribiremos como  $ph$ .

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  $X$ ).

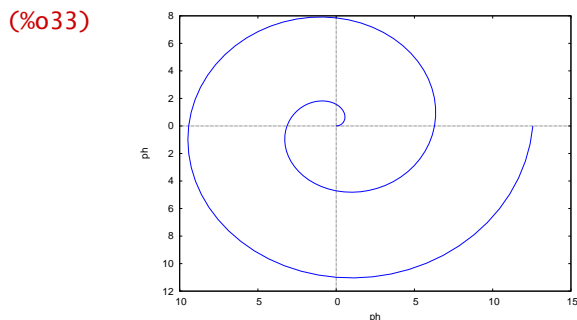
Al representar una curva en coordenadas polares estamos escribiendo la longitud del vector como una función que depende del ángulo. En otras palabras, para cada ángulo fijo decimos cuál es el módulo del vector. Quizá el ejemplo más sencillo de curva que se puede representar en coordenadas polares es una circunferencia centrada en el origen. ¿Cuál es la función en este caso? La función constantemente igual al radio. Veamos algún ejemplo un poco más elaborado.

```
(%i32) plot2d(ph,[ph,0,4*%pi],
             [gnuplot_preamble,"set polar;set zeroaxis;"])$
```



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 botón **Graduaciones** dentro de la ventana de **Gráficos 2D**.

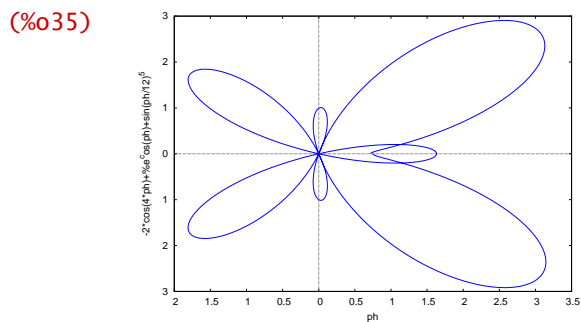
```
(%i33) plot2d(ph,[ph,0,4*%pi],
             [gnuplot_preamble,"set polar;set zeroaxis;"],[nticks,30])$
```



Ahora dibujaremos una mariposa. Para ello presentamos la función:  $r(\theta) = e^{\cos(\theta)} - 2 \cos(4\theta) + \sin\left(\frac{\theta}{12}\right)^5$ .

```
(%i34) r(ph):=exp(cos(ph))-2*cos(4*ph)+sin(ph/12)**5$
```

```
(%i35) plot2d(r(ph), [ph,0,2*%pi],
             [gnuplot_preamble, "set polar; set zeroaxis;"])$
```



## 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])
```

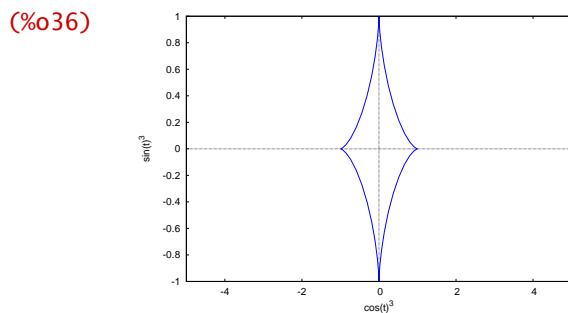
 gráfica de la curva  $(x(t), y(t))$  en  $[a, b]$ 

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:

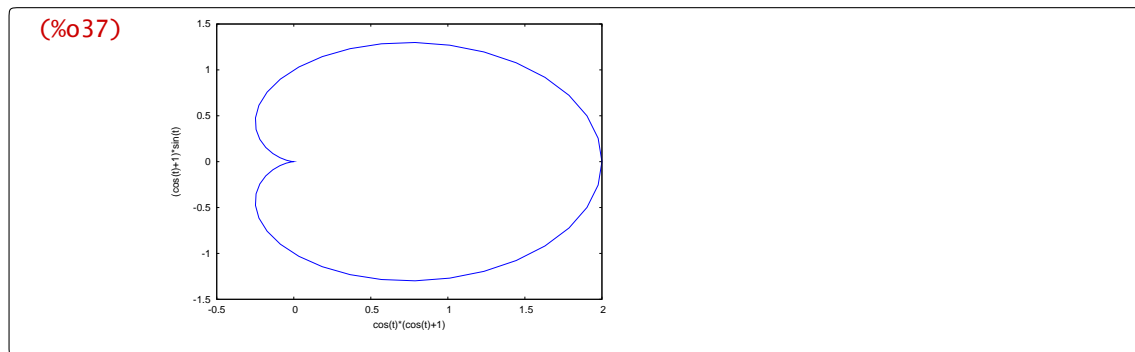
```
(%i36) plot2d([parametric, cos(ph)^3, sin(ph)^3], [ph,0,2*%pi], [nticks, 50])
```



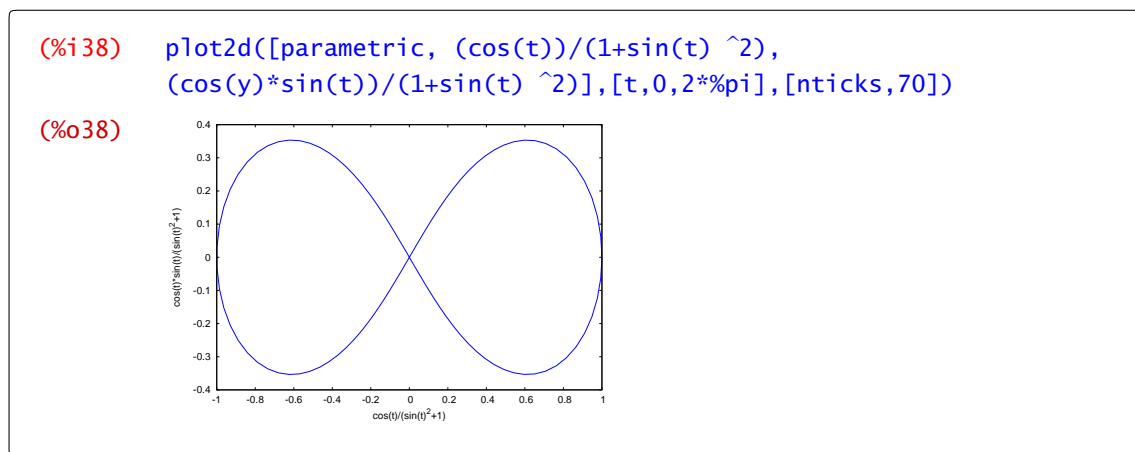
**Cardioid:** Es la curva trazada por un punto fijo de un círculo de radio  $r$  que rueda sin deslizar alrededor de otro círculo fijo del mismo radio. Sus ecuaciones paramétricas son.

```
(%i37) plot2d([parametric, (1+cos(ph))*cos(ph), (1+cos(ph))*sin(ph)],
             [ph,0,2*%pi], [nticks, 50])
```

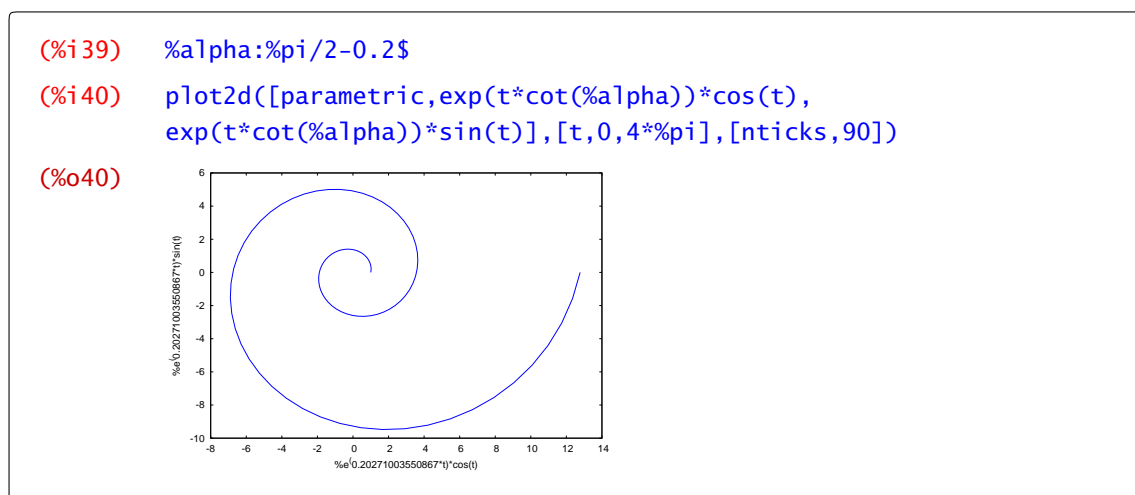




**Lemniscata de Bernoulli:** Es el lugar geométrico de los puntos  $P$  del plano, cuyo producto de distancias a dos puntos fijos  $F_1$  y  $F_2$ , llamados focos, verifica la igualdad  $|P - F_1| |P - F_2| = \frac{1}{4} |F_1 - F_2|^2$ . En coordenadas cartesianas esta curva viene dada por la ecuación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



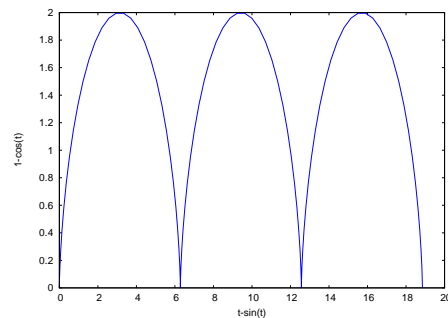
**Espiral equiangular:** También llamada espiral logarítmica. Es aquella espiral en la que el radio vector corta a la curva en un ángulo constante  $\alpha$ . Sus ecuaciones paramétricas son:



**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:

```
(%i41) plot2d([parametric,t-sin(t),1-cos(t)],[t,0,6*pi],[nticks,90])
```

```
(%o41)
```

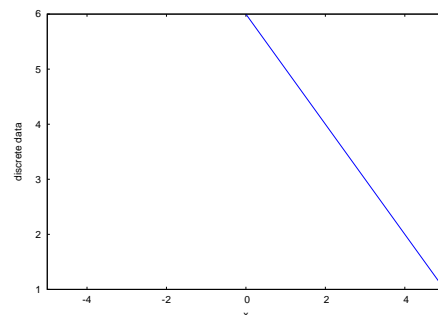


### 2.2.3 Gráficas de curvas poligonales

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:

```
(%i42) plot2d([discrete,[0,5],[6,1]],  
[x,-5,5]);
```

```
(%o42)
```

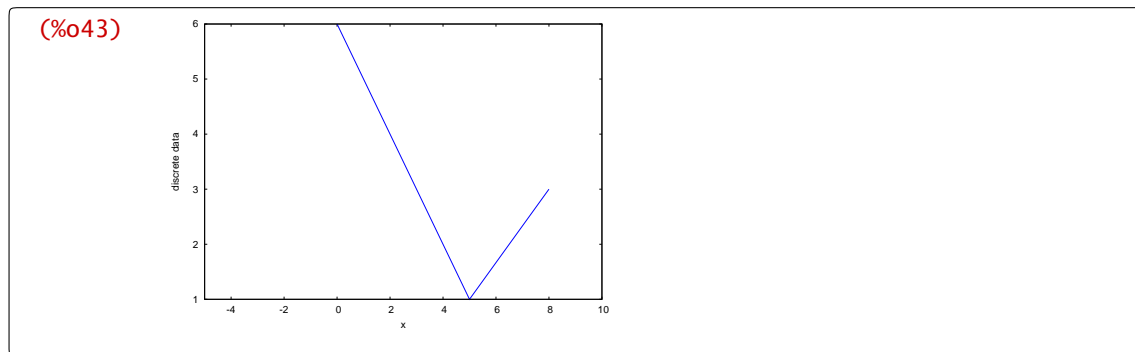


```
plot2d([discrete,[x1,x2,...],[y1,y2,...]],opc) poligonal que une los puntos  
(x1,y1), (x2,y2),...
```

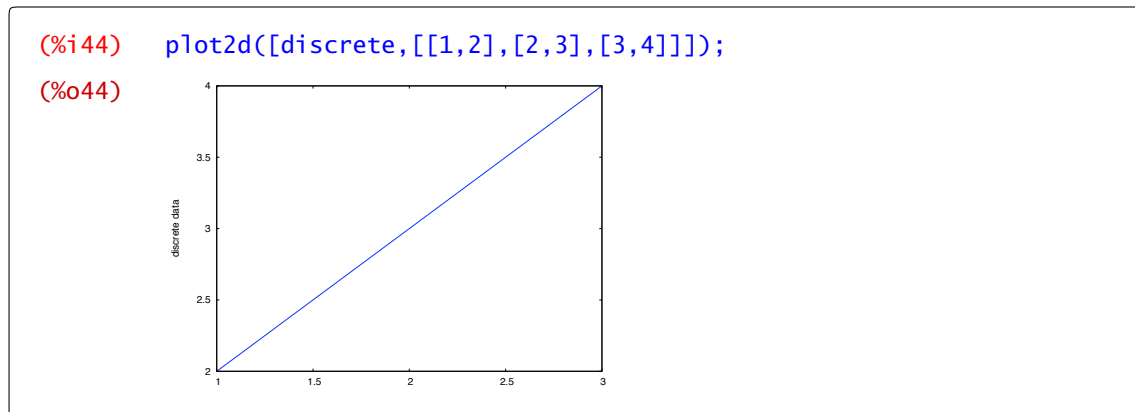
```
plot2d([discrete,[[x1,y1],[x2,y2],...]],opc) poligonal que une los puntos  
(x1,y1), (x2,y2),...
```

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

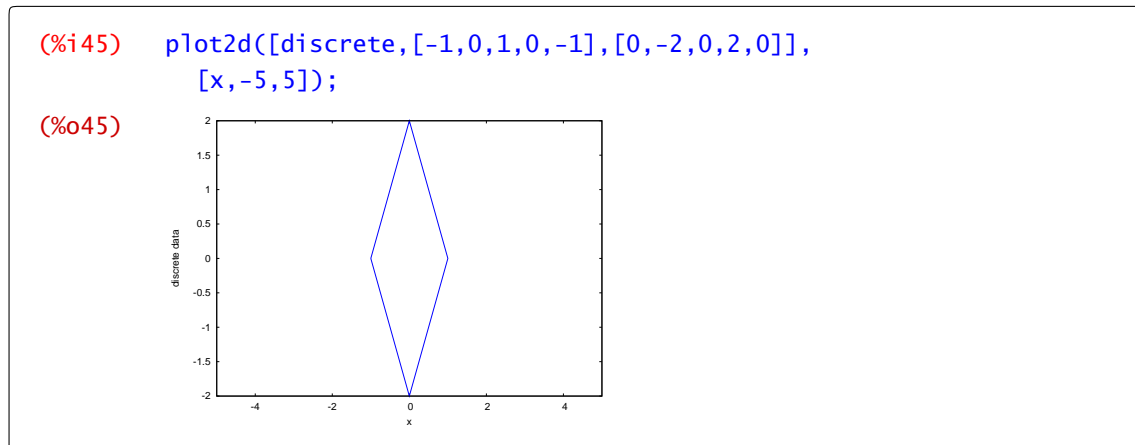
```
(%i43) plot2d([discrete,[0,5,8],[6,1,3]],  
[x,-5,10]);
```



o, agrupar los puntos en una lista,

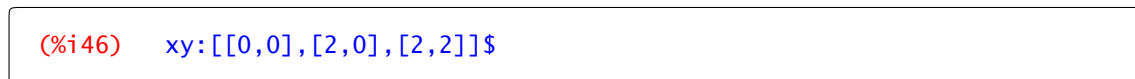


Si el primer y el último punto coinciden se obtiene lo esperable: una línea poligonal cerrada. Por ejemplo, el rombo de vértices  $(-1, 0)$ ,  $(0, -2)$ ,  $(1, 0)$  y  $(0, 2)$ :



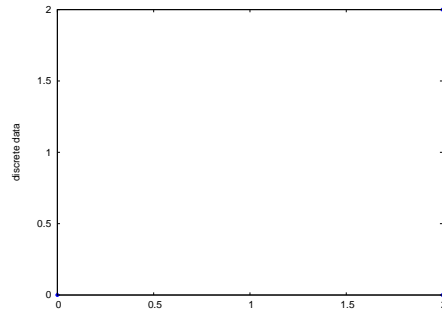
## Puntos y segmentos

¿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) plot2d([discrete,xy],[style,points]);
```

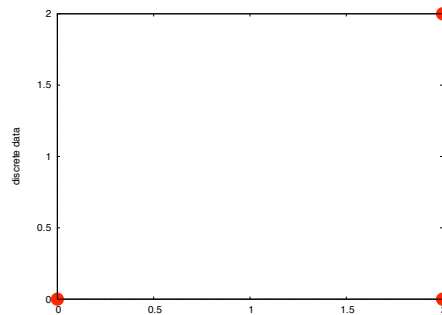
```
(%o47)
```



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,  $\alpha$ ,  $\beta$ ]`) que hacen referencia al ancho de la línea ( $\alpha$ ) y al color ( $\beta$ ).

```
(%i48) plot2d([discrete,xy],[style,[points,10,2]]);
```

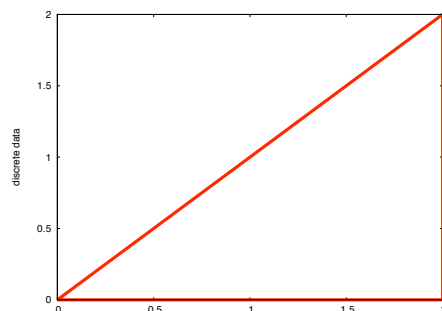
```
(%o48)
```



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

```
(%i49) plot2d([discrete,[0,2,2,0],[0,0,2,0]],[style,[lines,15,2]]);
```

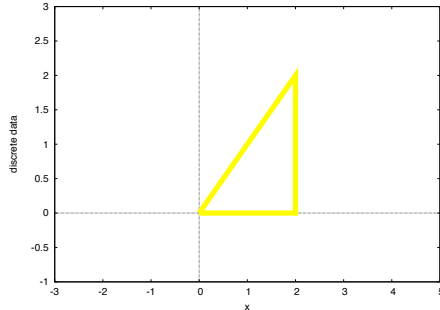
```
(%o49)
```



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.

```
(%i50) plot2d([discrete, [0,2,2,0],[0,0,2,0]],
[x,-3,5],[y,-1,3],
[style,[lines,25,5]]);
```

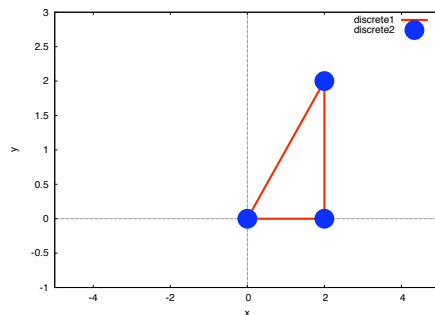
```
(%o50)
```



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

```
(%i51) 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]]);
```

```
(%o51)
```

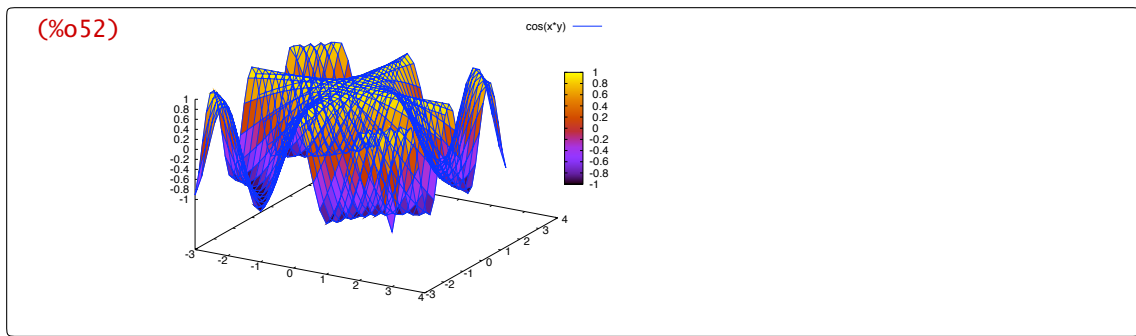


## 2.3 Gráficos en 3D

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  $[a, b] \times [c, d]$ .

<code>plot3d(f(x,y), [x,a,b], [y,c,d])</code>	gráfica de $f(x,y)$ en $[a,b] \times [c,d]$
<code>plot3d([f1(x,y), f2(x,y), ...], [x,a,b], [y,c,d])</code>	gráfica de una lista de funciones en $[a,b] \times [c,d]$

```
(%i52) plot3d(cos(x*y), [x,-3,3], [y,-3,3]);
```



**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.

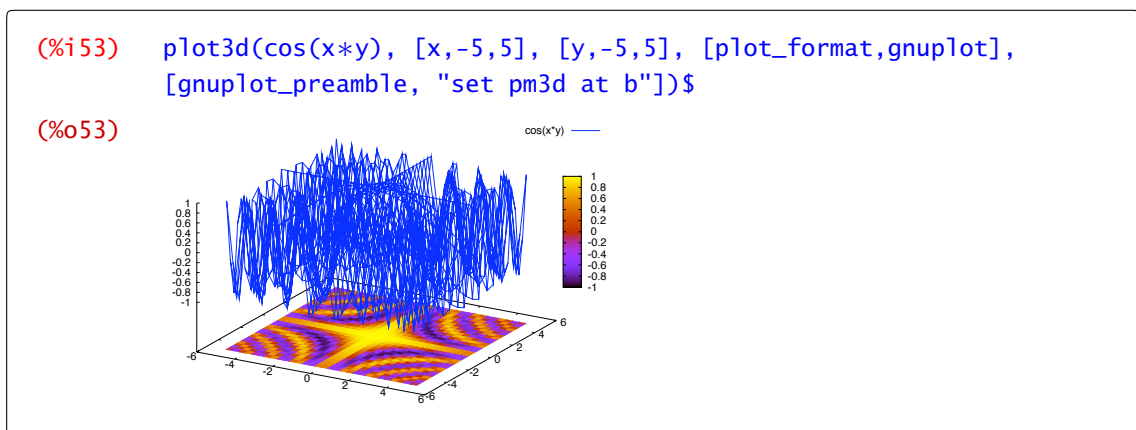
Al comando `plot3d` se accede a través del menú **Gráficos** → **Gráficos 3D** o del botón **Gráficos 3D**.



Figura 2.3 Gráficos 3D

Después de esto aparece la ventana de la Figura 2.3 con varios campos para rellenar:

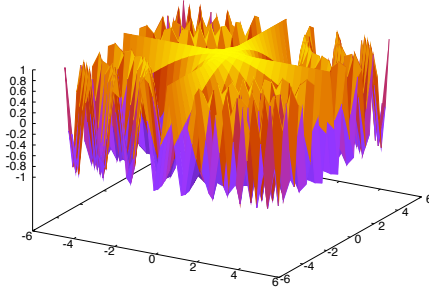
- Expresión. La función o funciones que vayamos a dibujar.
  - Variable. Hay dos campos para indicar el dominio de las dos variables.
  - Cuadrícula. Indica cuántas valores se toman de cada variable para representar la función. Cuanto mayor sea, más suave será la representación a costa de aumentar la cantidad de cálculos.
  - 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”.
  - Opciones. Las comentamos a continuación.
  - Gráfico al archivo. Permite elegir un fichero donde se guardará la gráfica.
- 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):



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):

```
(%i54) plot3d(cos(x*y), [x,-5,5], [y,-5,5], [plot_format,gnuplot],
[gnuplot_preamble, "set pm3d at s; unset surf; unset colorbox"]$
```

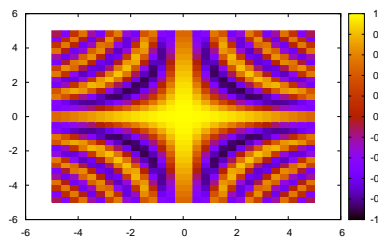
```
(%o54)
```



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

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

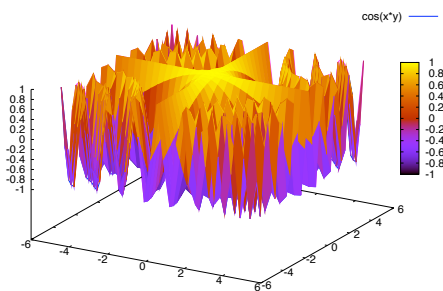
```
(%o55)
```



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.

```
(%i56) plot3d(cos(x*y), [x,-5,5], [y,-5,5], [plot_format,gnuplot],
[gnuplot_preamble, "set hidden3d"]$
```

```
(%o56)
```

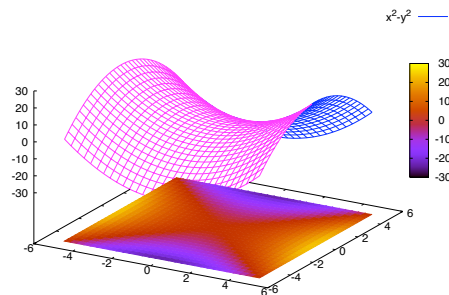


En el dibujo anterior (en el papel) es posible que no aprecie bien. A simple vista parece el mismo dibujo que teníamos dos salidas antes. Observa bien: hay una pequeña diferencia. El uso de pm3d hace que se coloree el dibujo, pero cuando decimos que no se muestra la parte no visible de la figura nos 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.

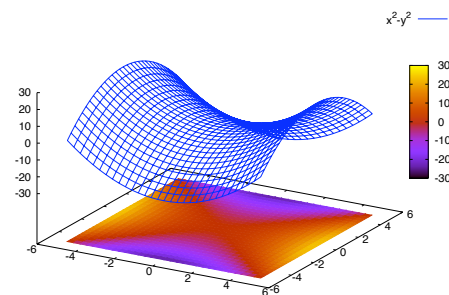
```
(%i57) plot3d(x^2-y^2, [x,-5,5], [y,-5,5], [plot_format,gnuplot],
           [gnuplot_preamble, "set pm3d at b; set hidden3d"])$
```

```
(%o57)
```



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

```
(%o58)
```



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.

```
(%i59) load(draw)$
```

<code>gr2d(opciones, objeto gráfico,...)</code>	gráfico dos dimensional
<code>gr3d(opciones, objeto gráfico,...)</code>	gráfico tres dimensional
<code>draw(opciones, objeto gráfico,...)</code>	dibuja un gráfico
<code>draw2d(opciones, objeto gráfico,...)</code>	dibuja gráfico dos dimensional
<code>draw3d(opciones, objeto gráfico,...)</code>	dibuja gráfico tres dimensional



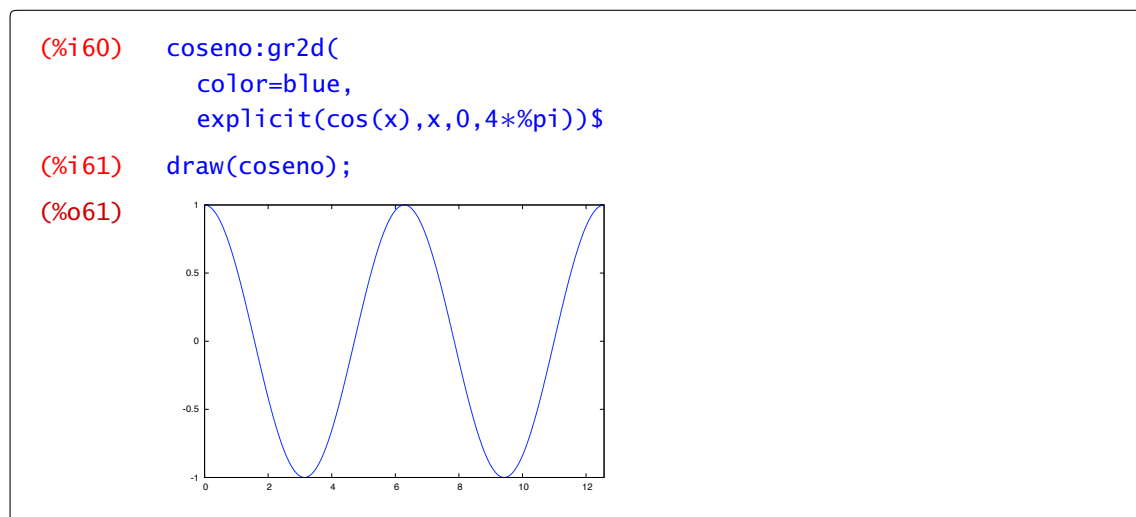
Una vez cargado el módulo draw, podemos utilizar las órdenes draw2d y draw3d para dibujar gráficos en 2 y 3 dimensiones o draw. Un gráfico está compuesto por varias opciones y el objeto gráfico que queremos dibujar. Por ejemplo, en dos dimensiones tendríamos algo así:

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

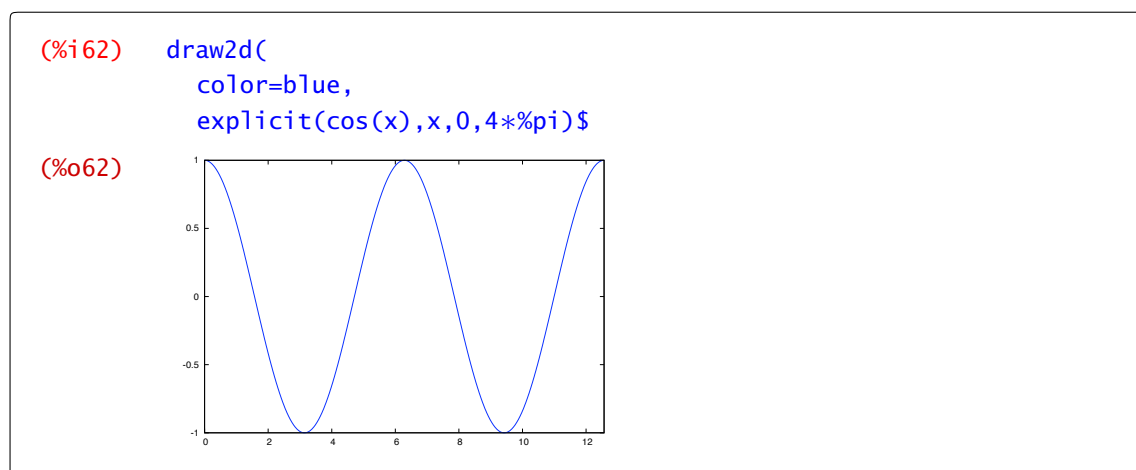
Las opciones son numerosas y permiten controlar prácticamente cualquier aspecto imaginable. Aquí comentaremos algunas de ellas pero la ayuda del programa es insustituible. En segundo lugar aparece el objeto gráfico. En este caso “explicit(cos(t),t,0,2\*%pi)”. Estos pueden ser de varios tipos aunque los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

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

draw  
draw2d

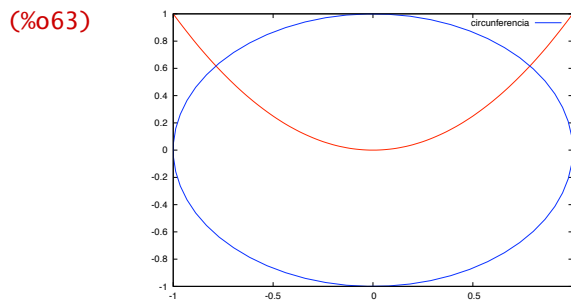


da el mismo resultado que



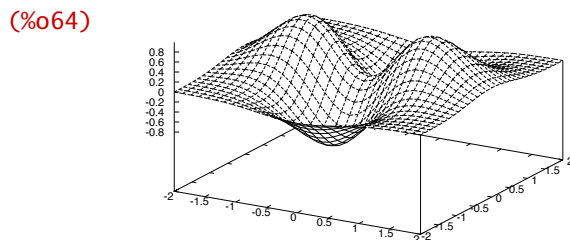
También podemos representar más de un objeto en un mismo gráfico. Simplemente escribimos uno tras otro separados por comas. En el siguiente ejemplo estamos mezclando una función dada explícitamente y una curva en coordenadas paramétricas.

```
(%i63) draw2d(
      color=red,
      explicit(x^2,x,-1,1),
      color=blue,nticks=60,
      parametric(cos(t),sin(t),t,0,2*%pi));
```



**draw3d** En tres dimensiones, la construcción es similar

```
(%i64) draw3d(surface_hide=true,
      explicit((x^2-y^2)*exp(1-x^2-y^2),x,-2,2,y,-2,2));
```



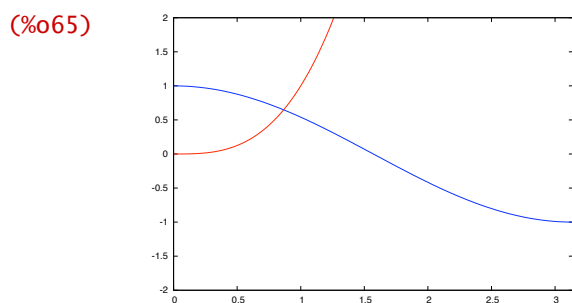
Vamos a comentar brevemente alguno de los objetos y de las opciones del módulo draw. Comenzamos con las opciones e iremos poniendo ejemplos con cada uno de los objetos.

### 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).

**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.

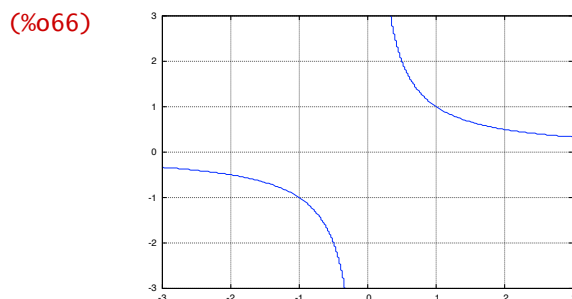
```
(%i65) draw2d(color=blue,
             explicit(cos(x),x,0,4*%pi),
             color=red,
             explicit(x^3,x,-5,5),
             xrange=[0,%pi],yrange=[-2,2])$
```



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.

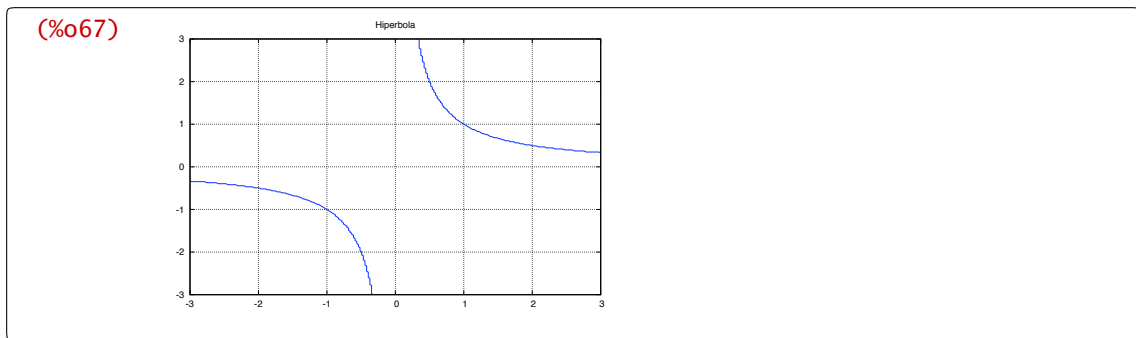
```
(%i66) draw2d(
        color=blue,nticks=100,
        implicit(x*y=1,x,-3,3,y,-3,3),
        grid=true)$
```



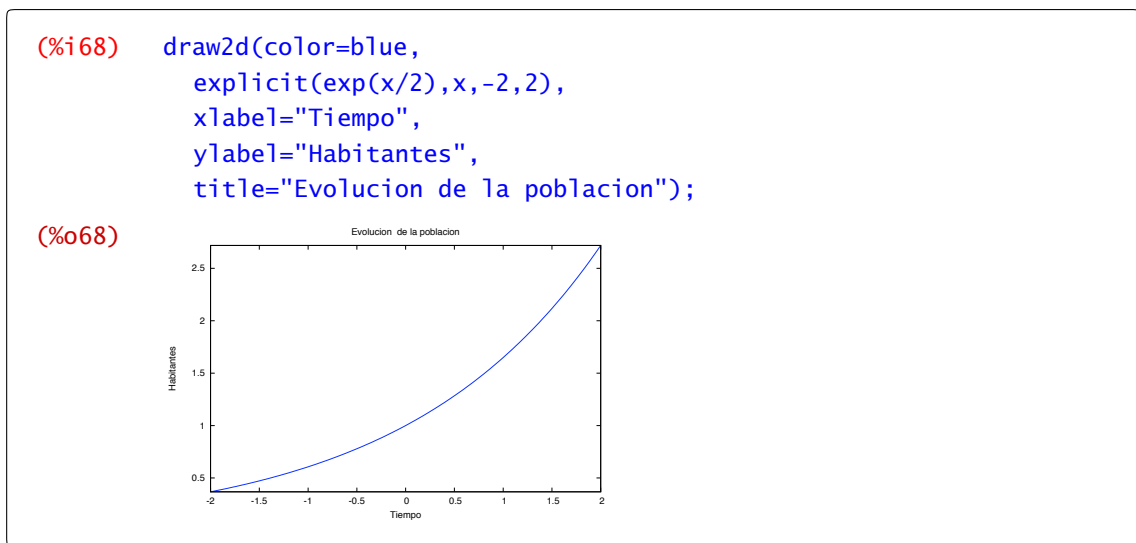
Acabamos de dibujar la hipérbola definida implícitamente por la ecuación  $xy = 1$ . La opción `grid` nos ayuda a hacernos una idea de los valores que estamos representando.

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

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



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



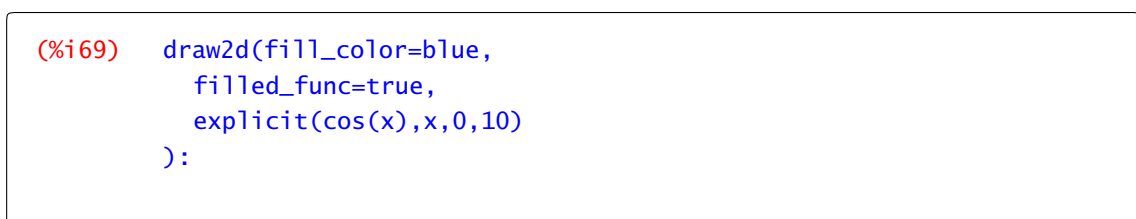
**axis, yaxis, zaxis:** si vale `true` se dibuja el correspondiente eje. Es una opción global.

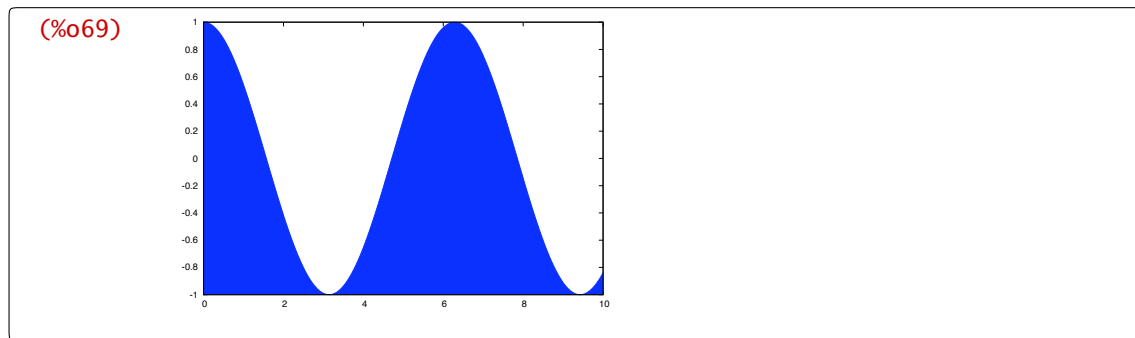
**enhanced3d:** si vale `true` se colorean las superficies en gráficos tridimensionales.

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

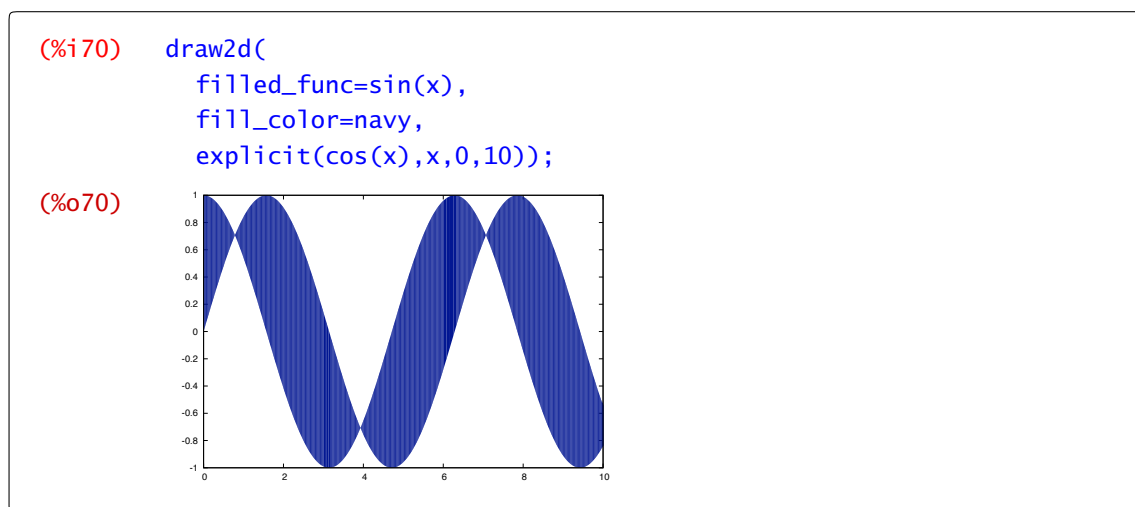
**point\_type:** indica cómo se van a dibujar los puntos. El valor para esta opción puede ser un nombre o un número: `none` (-1), `dot` (0), `plus` (1), `multiply` (2), `asterisk` (3), `square` (4), `filled_square` (5), `circle` (6), `filled_circle` (7), `up_triangle` (8), `filled_up_triangle` (9), `down_triangle` (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`

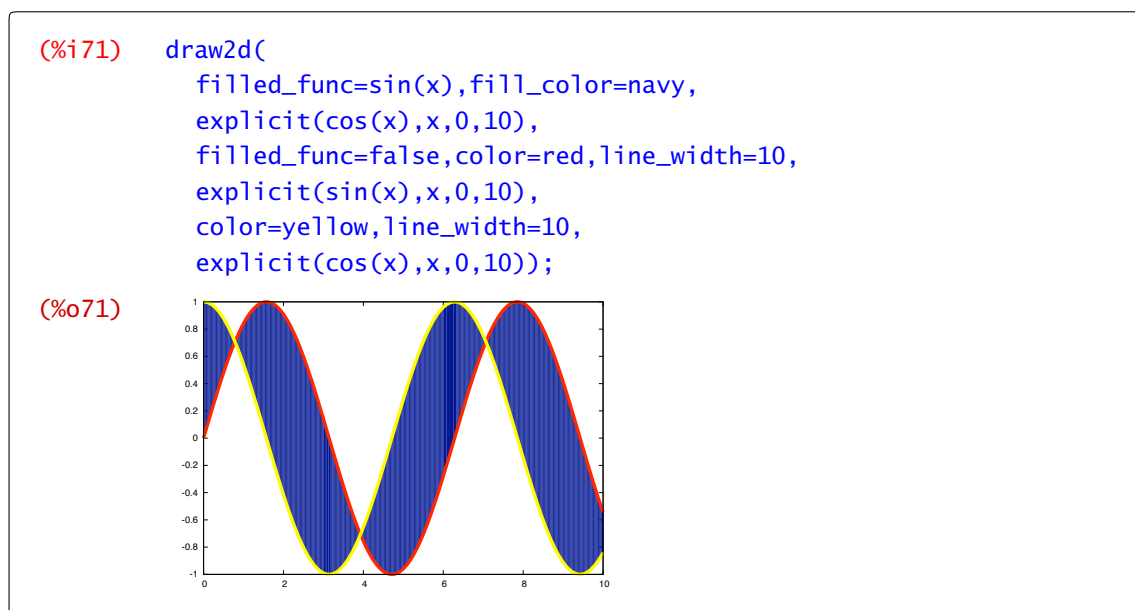




en cambio, si `filled_func` es una función, entonces se colorea el espacio entre dicha función y la gráfica que estemos dibujando



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:



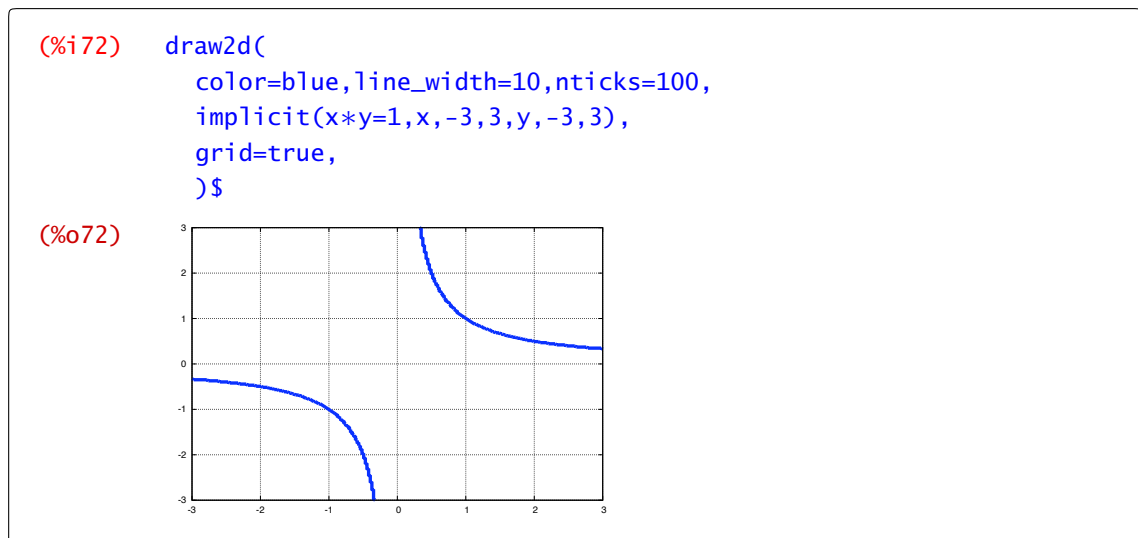
**fill\_color:** ver el apartado anterior `filled_func`.

**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.

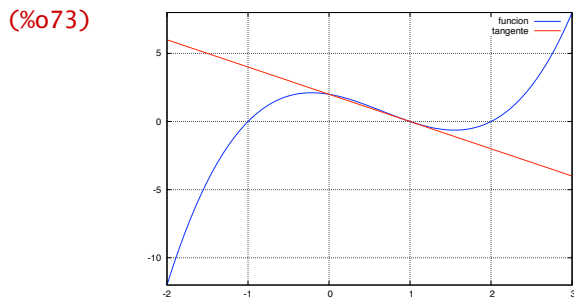
**line\_width:** grosor con el que se dibujan las líneas. Su valor por defecto es 1.



**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.

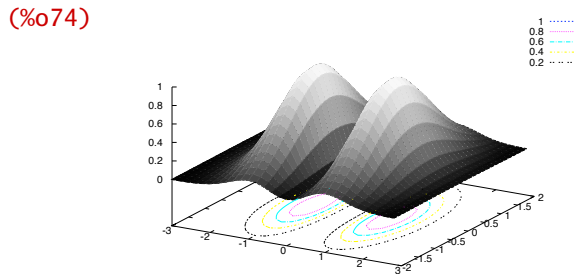
```
(%i73) draw2d(color=blue,key="función",explicit(f(x),x,-2,3),
color=red,key="tangente",explicit(tangente(x,1),x,-2,3),
grid=true);
```



**surface\_hide:** cuando vale true no se dibuja la parte no visible de las superficies.

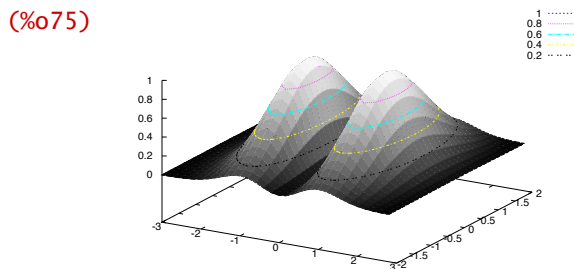
**contour** nos permite dibujar o no las curvas de nivel de una superficie. Por defecto no se muestran (none) pero también podemos dibujarlas sobre el plano XY con base

```
(%i74) 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));
```



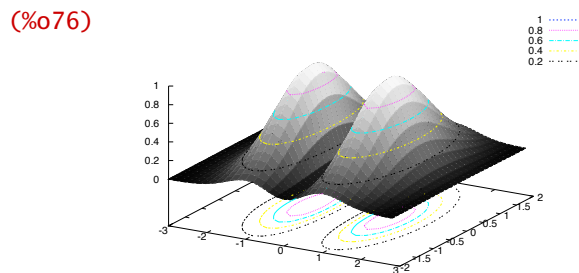
sobre la propia superficie con surface

```
(%i75) 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));
```



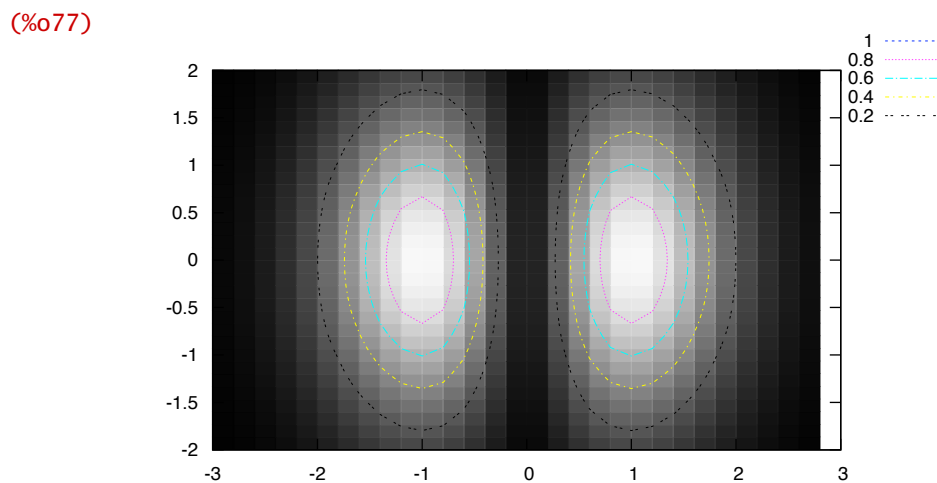
las dos posibilidades anteriores se pueden usar a la vez con both

```
(%i76) 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));
```



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

```
(%i77) 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));
```



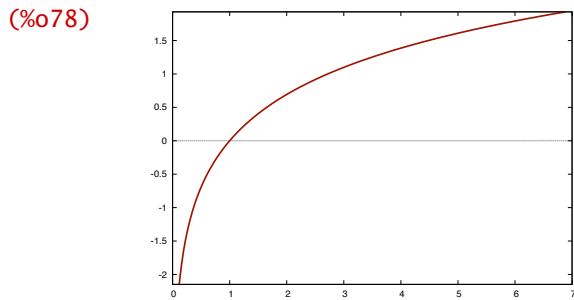
## 2.4.2 Objetos

**explicit:** nos permite dibujar una función de una o dos variables. Para funciones de una variable

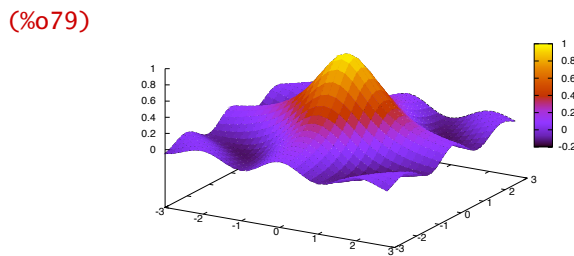


usaremos `explicit(f(x),x,a,b)` para dibujar  $f(x)$  en  $[a,b]$ . Con funciones de dos variables escribiremos `explicit(f(x,y),x,a,b,y,c,d)`.

```
(%i78) draw2d(
      color=dark-red,line_width=5,
      xaxis=true,yaxis=true,
      explicit(log(x),x,0,7) );
```



```
(%i79) draw3d(enhanced3d=true,
      surface_hide=true,
      explicit((1+x^2+y^2)^(-1)*cos(x*y),x,-3,3,y,-3,3));
```



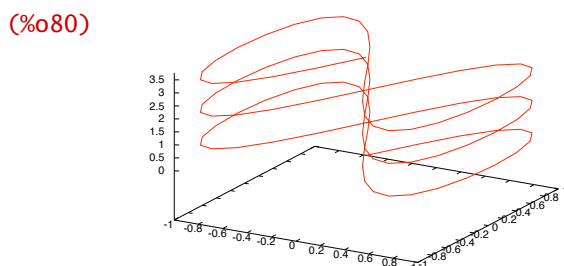
**parametric:** nos permite dibujar una curva en dos o en tres dimensiones. En coordenadas paramétricas, tenemos que dar las dos o tres coordenadas de la forma

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

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

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

```
(%i80) draw3d(color=red,
      nticks=100,
      parametric(sin(t),sin(2*t),t/5,t,0,6*pi));
```

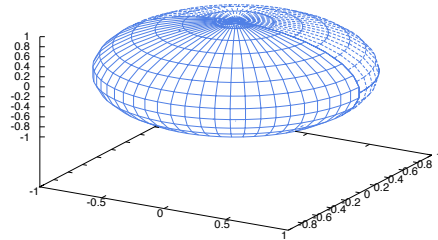


**parametric\_surface:** nos permite dibujar una superficie en paramétricas. Tenemos pues que dar las tres coordenadas así que escribimos

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

```
(%i81) 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));
```

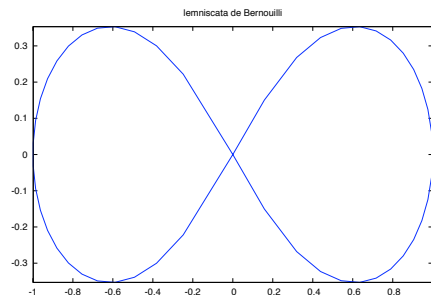
```
(%o81)
```



**polar:** dibuja el módulo del vector en función del ángulo  $t$

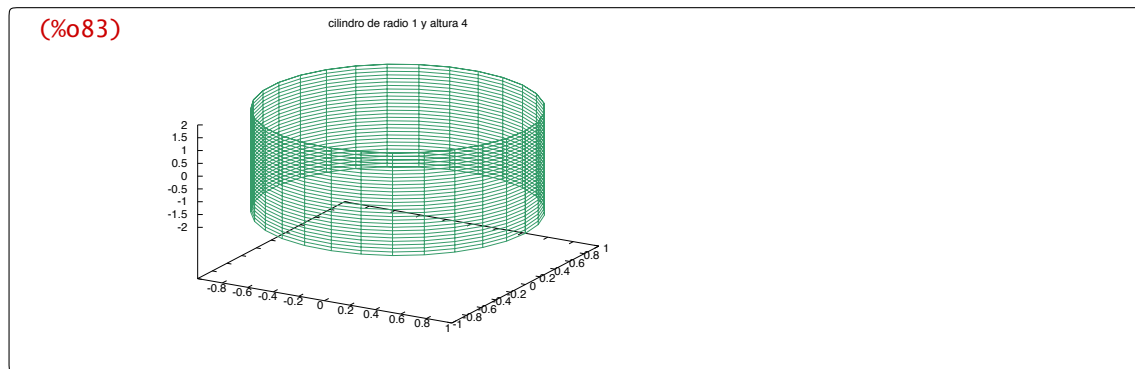
```
(%i82) draw2d(
      color=blue,
      nticks=100,
      title="lemniscata de Bernoulli",
      polar(sqrt(cos(2*t)), t, 0, 2*%pi));
```

```
(%o82)
```

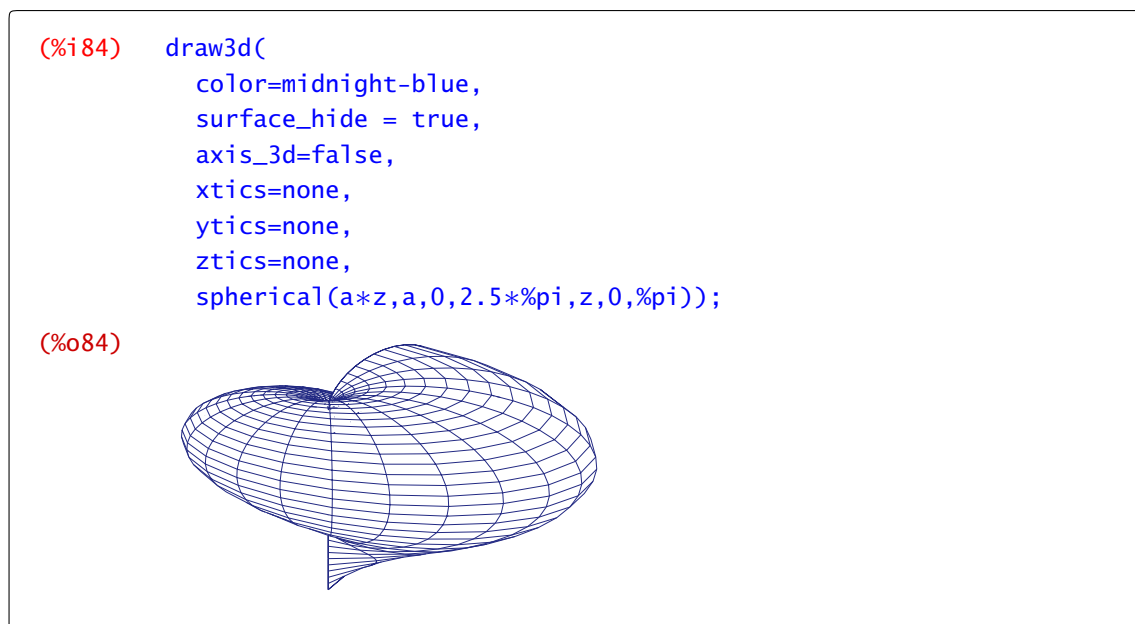


**cylindrical:** para dar un vector del espacio en coordenadas cilíndricas lo que hacemos es cambiar a coordenadas polares en las dos primeras variables (aunque tendría perfecto sentido hacerlo con cualquier par de ellas). La figura cuya representación es más sencilla en coordenadas cilíndricas te la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

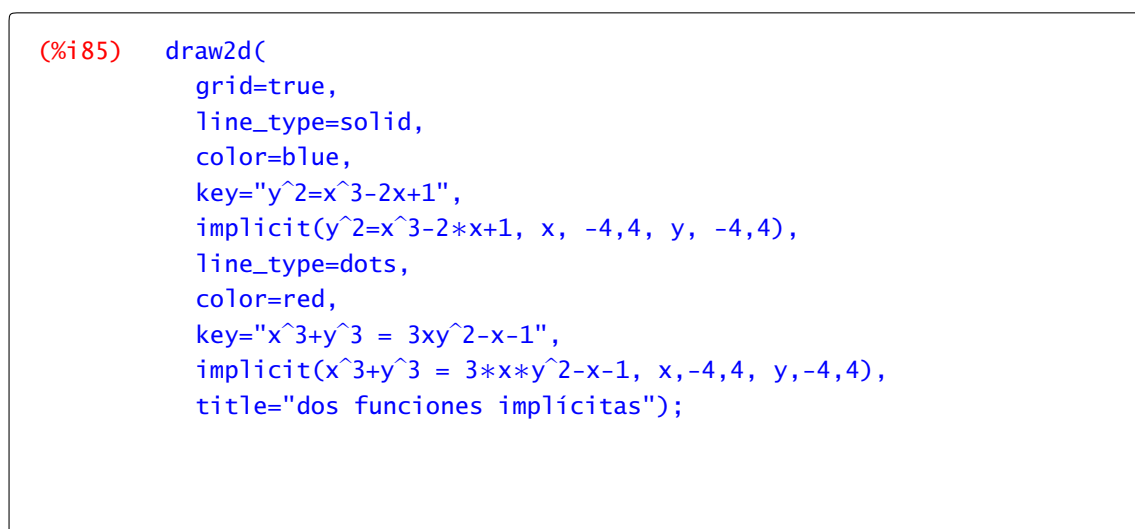
```
(%i83) draw3d(
      color=sea-green,
      title="cilindro de radio 1 y altura 4",
      cylindrical(1, z, -2, 2, t, 0, 2*%pi));
```

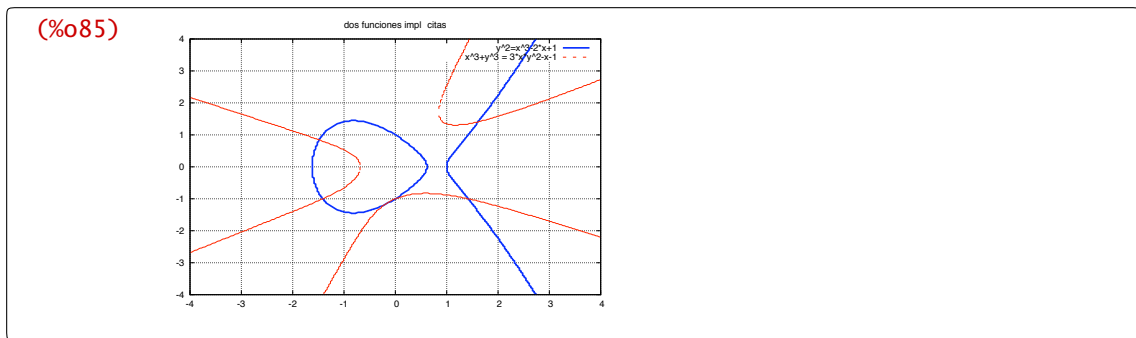


**spherical**: en coordenadas esféricas la construcción es muy parecida. Representamos el módulo y damos dónde varían los dos ángulos.

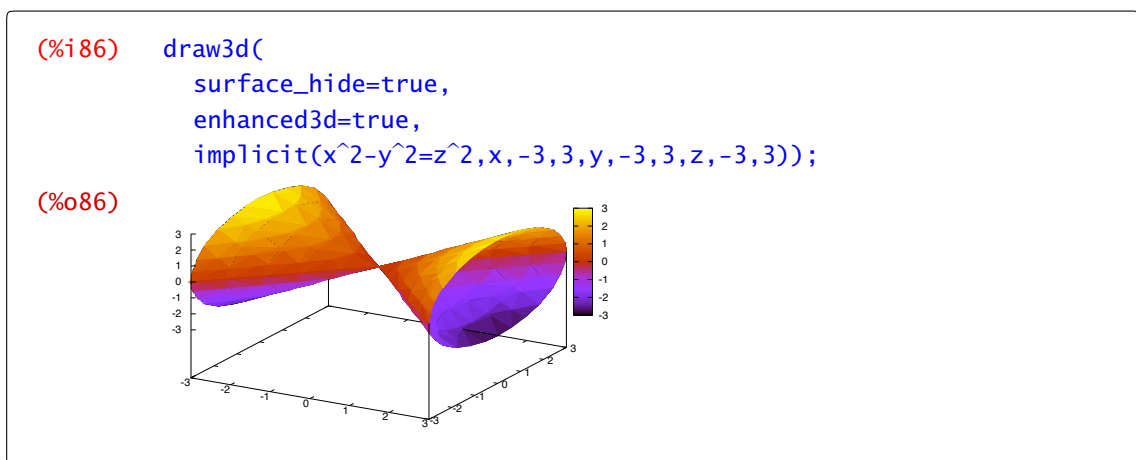


**implicit**: nos permite dibujar el lugar de los puntos que verifican una ecuación en el plano

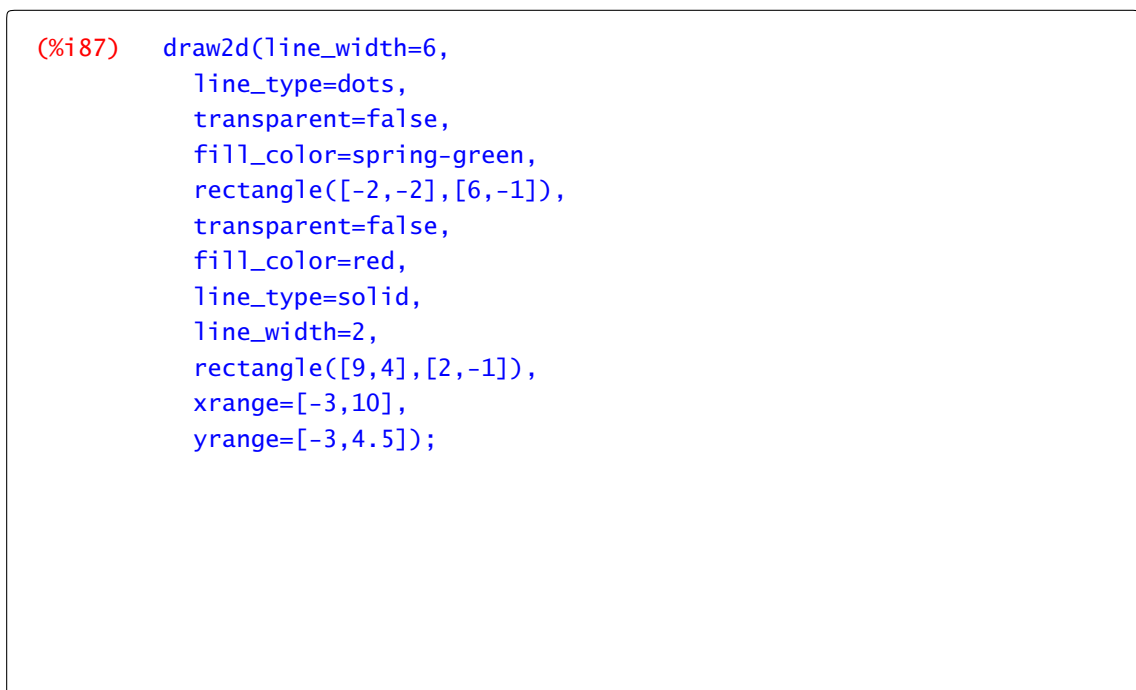


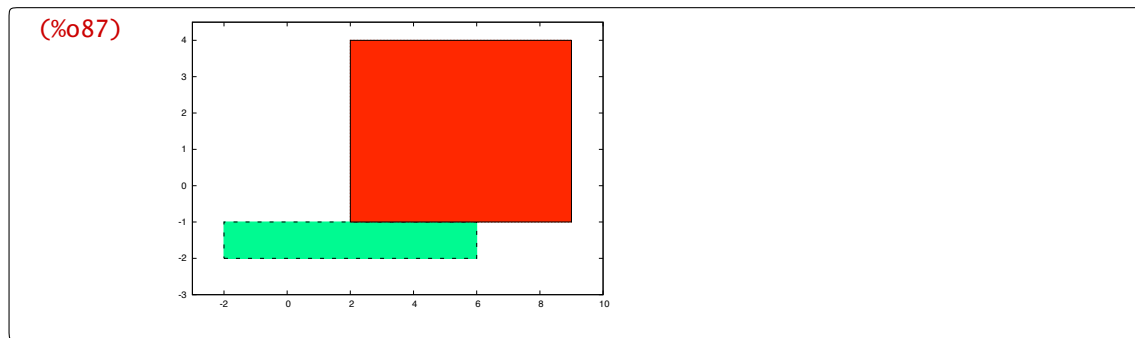


o en tres dimensiones. Además de la ecuación debemos indicar los intervalos dónde pueden tomar valores las variables.



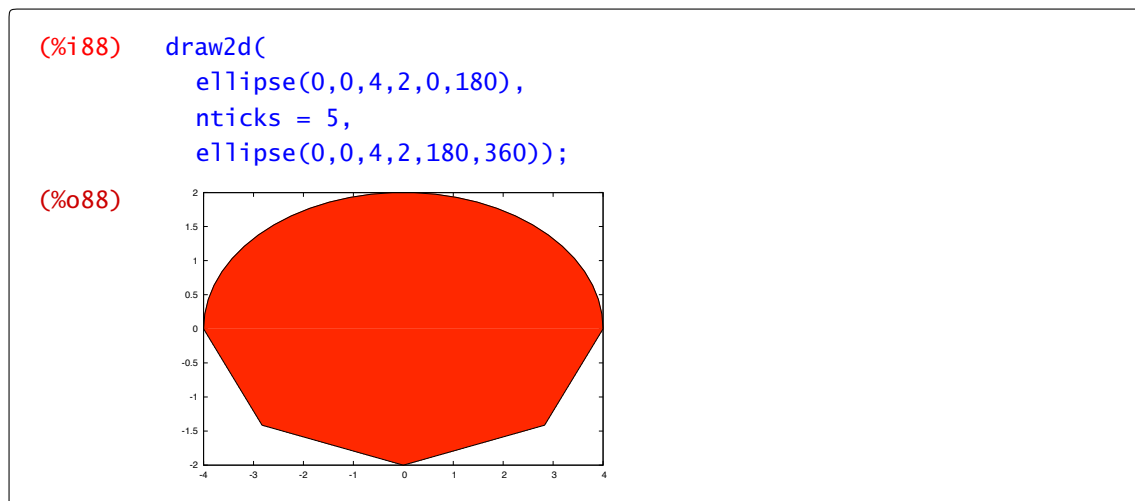
**rectangle:** para dibujar un rectángulo sólo tenemos que indicar el vértice inferior izquierdo y su opuesto.





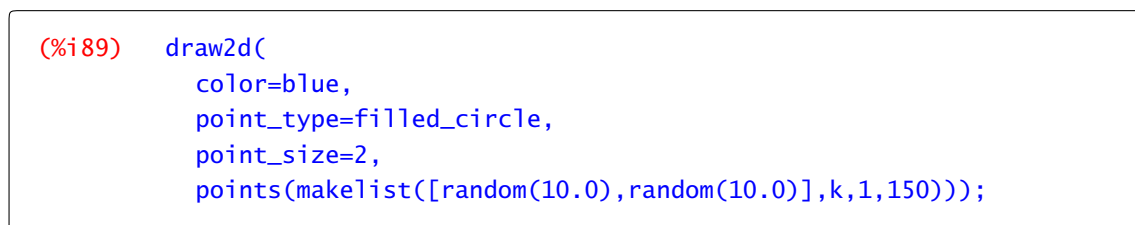
**ellipse:** la orden `ellipse` permite dibujar elipses indicando 3 pares de números: los dos primeros son las coordenadas del centro, los dos segundos indican la longitud de los semiejes y los últimos son los ángulos inicial y final.

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.

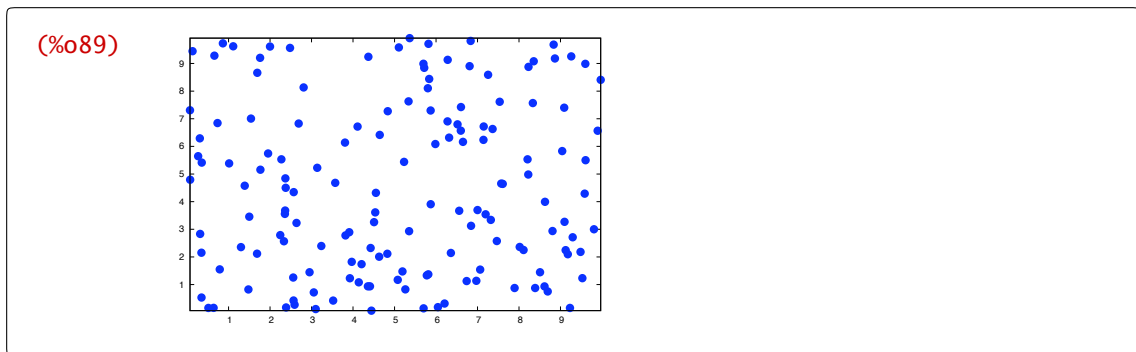


La parte superior de la elipse se ha dibujado utilizando 30 puntos y la inferior únicamente 5.

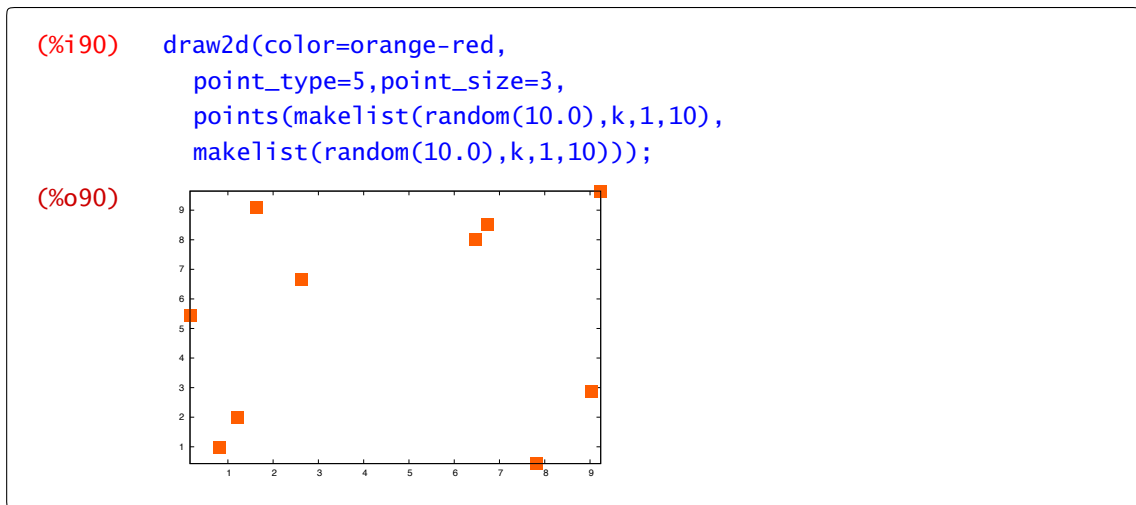
**points:** para representar una lista de puntos en el plano o en el espacio tenemos dos posibilidades. Podemos dar los vectores de la forma<sup>1</sup> `[[x1,y1],[x2,y2],...]`, como por ejemplo



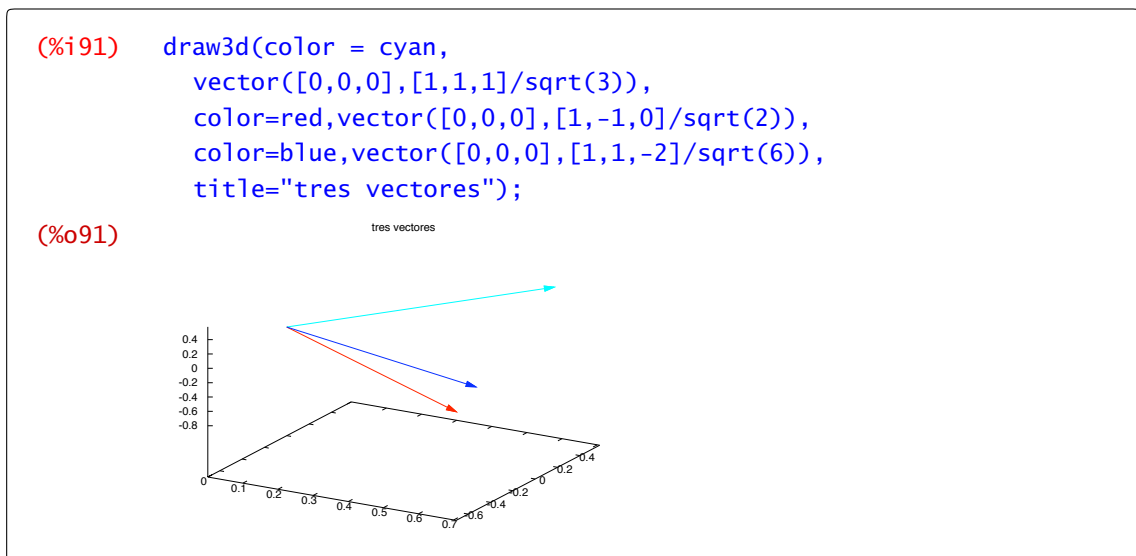
<sup>1</sup> En el ejemplo usaremos la orden `makeList` que genera una lista de acuerdo a la regla que aparece como primera entrada con tantos elementos como indique el contador que le sigue. En el próximo capítulo lo comentaremos con más detalle.



o podemos agrupar por coordenadas de la forma  $[[x_1, x_2, x_3, \dots], [y_1, y_2, y_3, \dots]]$  como aquí.



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



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.

<code>with_slider</code>	animación de <code>plot2d</code>
<code>with_slider_draw</code>	animación de <code>draw2d</code>
<code>with_slider_draw3d</code>	animación de <code>draw3d</code>

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. Utilizamos la orden `range(1,20)` que nos da todos los números naturales comprendidos entre 1 y 20.

`with_slider`

```
(%i92) with_slider(n,range(1,20),sin(x+n),[x,-2*%pi,2*%pi],[y,-1.1,1.1]);
```

En la Figura 2.4 tienes la representación de algunos valores

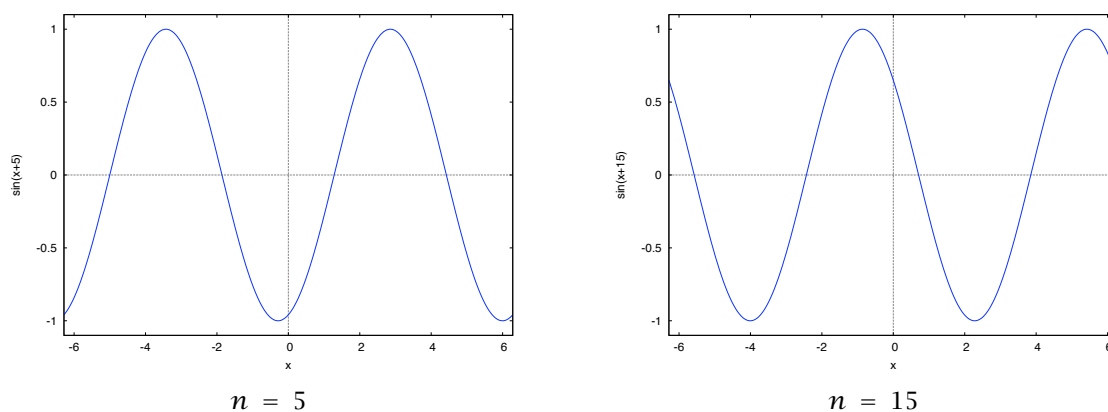
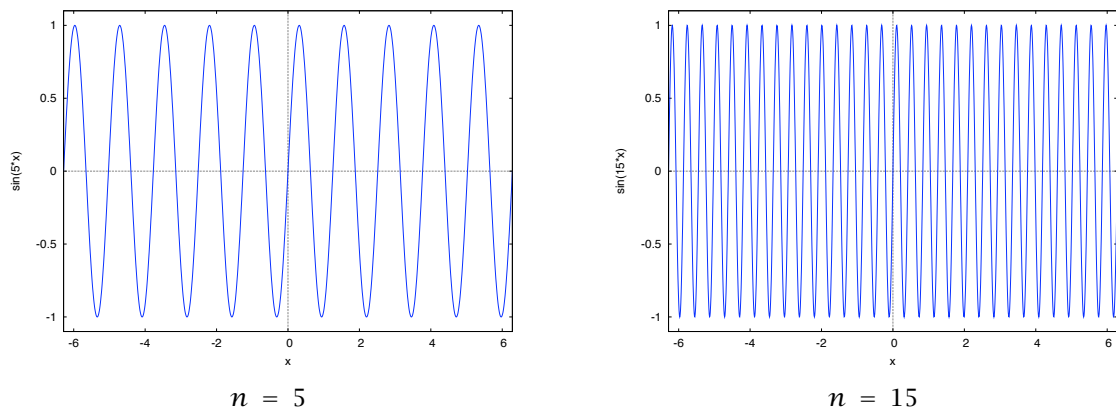


Figura 2.4  $\text{sen}(x + n)$

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.

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

Puedes ver en la Figura 2.5 puedes ver cómo aumenta la frecuencia con  $n$ .

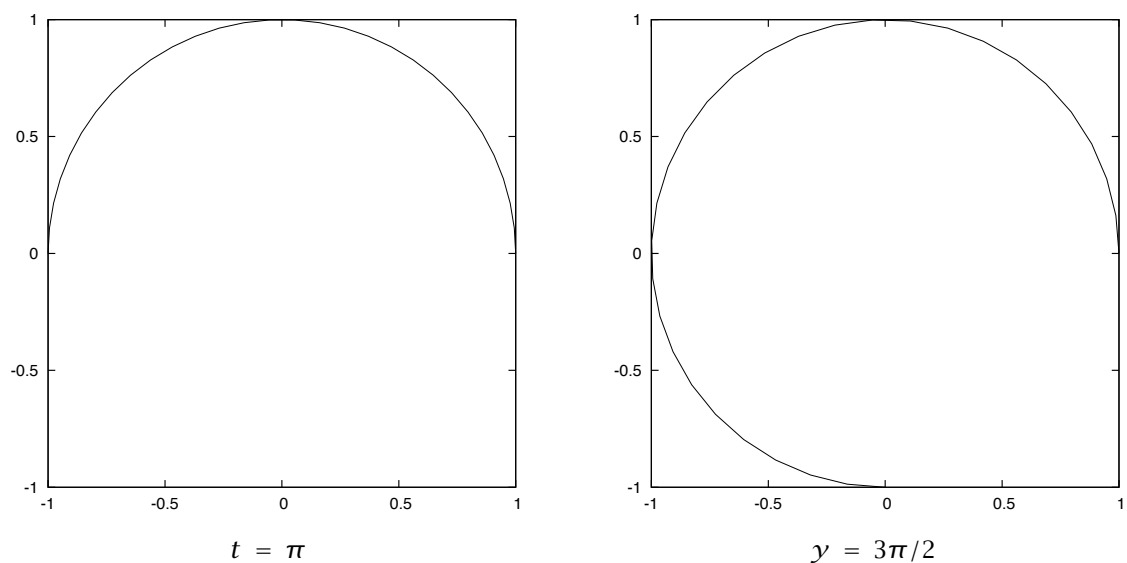


**Figura 2.5**  $\text{sen}(nx)$

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, luego 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 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

```
(%i94) with_slider_draw(
      t, 2*%pi*range(1,20)/20,
      parametric(cos(x), sin(x), x, 0, t),
      xrange=[-1,1],
      yrange=[-1,1]
      user_preamble="set size ratio 1")$
```

En la Figura 2.6 tenemos representados algunos pasos intermedios



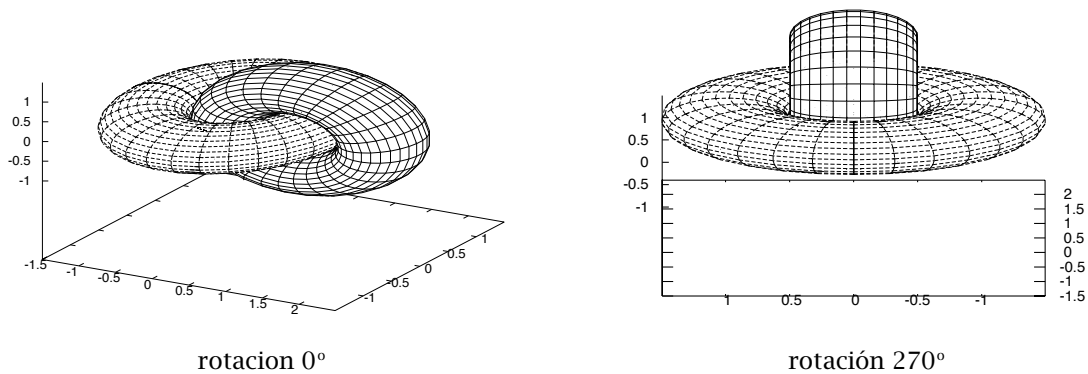
**Figura 2.6** Construcción de una circunferencia en paramétricas



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 superficie.

```
(%i95) with_slider_draw3d(
      k,range(1,10)*36,
      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 2.7 algunos pasos intermedios



**Figura 2.7** Giro alrededor de una superficie en paramétricas

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

```
(%i96) with_slider_draw3d(
      k,range(-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 2.8.

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.

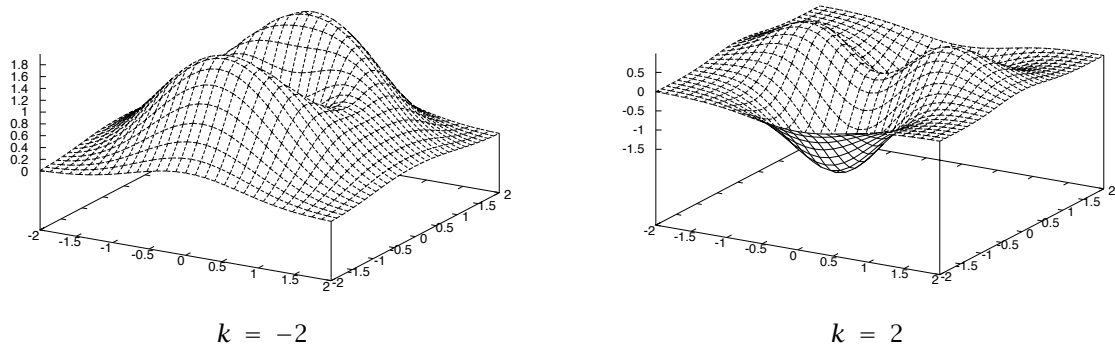


Figura 2.8 Función  $f(x, y) = (x^2 - ky^2)e^{1-x^2-y^2}$

```
(%i97) with_slider_draw(t, 2*pi*range(1, 40)/39,
        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 2.9

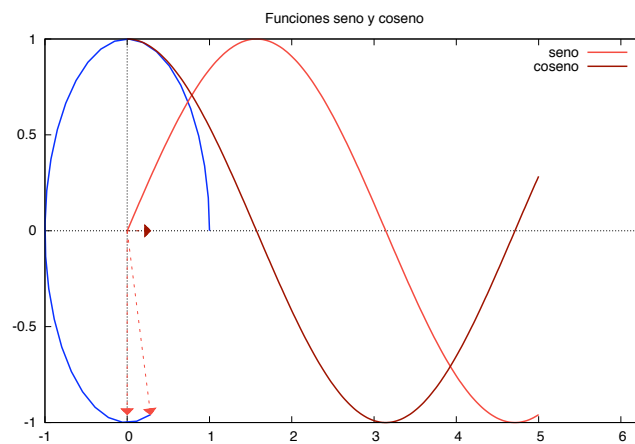


Figura 2.9 Las funciones seno y coseno

## 2.6 Ejercicios

### Ejercicio 2.1

Representa la gráfica de la función  $f(x) = \cos^2(x) - x \operatorname{sen}(x)$  en  $[-\pi, \pi]$  y sobre ella representa 8 puntos elegidos aleatoriamente.

### Ejercicio 2.2

Representa la gráfica de la función  $f: \mathbb{R}_0^+ \rightarrow \mathbb{R}$  definida como

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

### Ejercicio 2.3

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.

### Ejercicio 2.4

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

- $\gamma(\theta) = 2 + \cos(5\theta), \forall \theta \in [0, 2\pi]$ .
- $\gamma(\theta) = 8 \operatorname{sen}(5/2\theta), \forall \theta \in [0, 2\pi]$ .
- $\gamma(t) = (\operatorname{sen}(t), \operatorname{sen}(2t), t/5), t \in [0, 5]$ ,
- $\gamma(t) = 2(\cos(3t), \operatorname{sen}(5t)), t \in [0, 2\pi]$ .
- $\gamma(u, v) = (\operatorname{sen}(u), \operatorname{sen}(v), v)$  con  $u \in [-\pi, \pi], v \in [0, 5]$ .
- $\gamma(u, v) = (u \cos(v) \operatorname{sen}(u), u \cos(u) \cos(v), -u \operatorname{sen}(v))$ , con  $u, v \in [0, 2\pi]$ .

### Ejercicio 2.5

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 2.6

Realiza una animación gráfica para representar la curva

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

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

### Ejercicio 2.7

Realiza una animación gráfica que represente la cicloide.



# Listas y matrices

## 3

### 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,"ho1a"];
(%o3) [0,1,-3,a,ho1a]
```

first, second, ..., tenth	primera, segunda,...,décima entrada de una lista
lista[i]	entrada <i>i</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
```

`last` o podemos referirnos directamente al último término.

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

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

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

nos da, por ejemplo el primer elemento de la lista. 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);
```

```
(%i12) a
```

## Vectores

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

```
(%i13) v1:[1,0,-1];v2:[-2,1,3];
```

```
(%o13) [1,0,-1]
```

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

```
(%i15) v1+v2;
```

```
(%o15) [-1,1,2]
```

o multiplicarlos.

```
(%i16) v1*v2;
```

```
(%o16) [-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 “.”

```
(%i17) v1.v2;
```

```
(%o17) -5
```

Podemos ordenar los elementos de la lista

**sort**

```
(%i18) sort(v1);
```

```
(%o18) [-1,0,1]
```

o saber cuántos elementos tiene la lista

**length**

```
(%i19) length(v1);
```

```
(%o19) 3
```

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

**flatten**

```
(%i20) flatten([[1,2],1,[3,a,1]])
```

```
(%o20) [1,2,1,3,a,1]
```

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

```
(%i21) unique(%)
(%o21) [1,2,3,a]
```

### 3.1.1 Construir y operar con listas

<code>makelist</code>	genera lista
<code>range</code>	construir lista de enteros
<code>apply</code>	aplicar un operador a una lista
<code>map</code>	aplicar una función a una lista
<code>listp(expr)</code>	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  $\sin(x)$ ,  $\sin(2x)$ , ...,  $\sin(20x)$ . 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:

`makelist`

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

`range` Hay otro tipo particular de listas que se suelen utilizar como contadores. La orden `range` nos da los números enteros entre dos fijos:

```
(%i23) range(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]
```

¿Se te ocurre alguna forma de ir de 0.5 en 0.5 o de 2 en 2? Lo más sencillo es multiplicar. Por ejemplo los números del 0 al 100 de dos en dos son los siguientes:

```
(%i24) 2*range(0,50);
```



```
(%i24) [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42,
(%o24) 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  $\sin(1)$ ,  $\sin(2)$ , ...,  $\sin(10)$ , hacemos lo siguiente

`map`

```
(%i25) map(sin, range(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

`apply`

```
(%i27) apply("+", range(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("+", range(1, 100))/100;
(%o28)  $\frac{101}{2}$ 
```

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

```
(%i29) apply("*", range(1, 100))^(1/100);
(%o29)  $17^{\frac{1}{20}} 19^{\frac{1}{20}} 23^{\frac{1}{25}} 37^{\frac{1}{50}} 41^{\frac{1}{50}} 43^{\frac{1}{50}} 47^{\frac{1}{50}} 2401^{\frac{1}{25}} 15625^{\frac{1}{25}} 531441^{\frac{1}{25}}$ 
63871474118205504453038352[20digits]8482099766363898994157944832  $\frac{1}{100}$ 
```

```
(%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√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

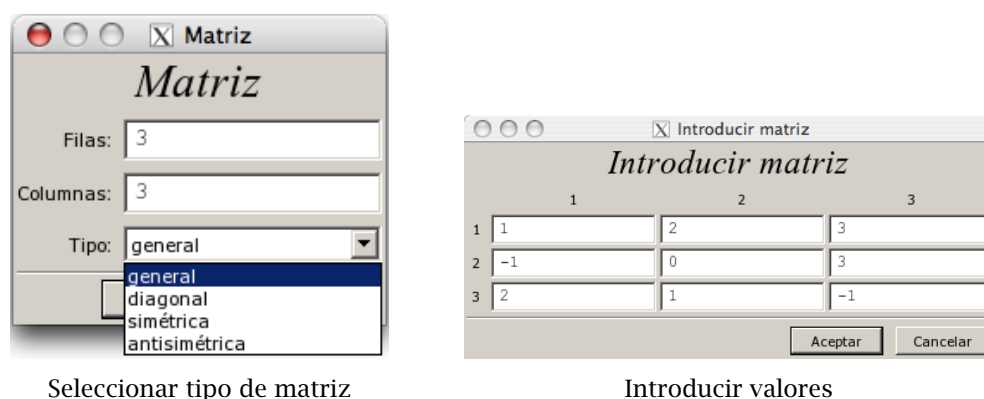
Las matrices se escriben de forma parecida a las listas y, de hecho, sólo tenemos que agrupar las filas de la matriz escritas como listas bajo la orden `matrix`. Vamos a definir un par de matrices y un par de vectores que van a servir en los ejemplos en lo que sigue.

```
(%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.

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



Seleccionar tipo de matriz

Introducir valores

Figura 3.1 Introducir matriz

<code>matrix(filas1, filas2, ...)</code>	matriz
<code>matrix_size(matriz)</code>	número de filas y columnas
<code>matrixp(expresión)</code>	devuelve true si <i>expresión</i> es una matriz

**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 matrices?



matrixp

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

Aunque pueda parecer lo contrario, no son matrices. 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?

```
(%i39) [1,2,3];
(%o39) [1,2,3]
(%i40) matrix([1,2,3]);
(%o40) [ 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 ese caso.

### 3.2.1 Operaciones elementales con matrices

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

```
(%i41) A+B;
```

$$\begin{aligned} (\%o41) & \begin{bmatrix} 0 & 3 & 4 \\ 0 & 0 & 3 \\ -1 & 8 & 1 \end{bmatrix} \\ (\%i42) & \text{A-B;} \\ (\%o42) & \begin{bmatrix} 2 & 1 & 2 \\ -2 & 0 & 3 \\ 5 & -6 & -3 \end{bmatrix} \end{aligned}$$

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.

$$\begin{aligned} (\%i43) & \text{A.B;} \\ (\%o43) & \begin{bmatrix} -8 & 22 & 7 \\ -8 & 20 & 5 \\ 2 & -5 & 0 \end{bmatrix} \\ (\%i44) & \text{A*B;} \\ (\%o44) & \begin{bmatrix} -1 & 2 & 3 \\ -1 & 0 & 0 \\ -6 & 7 & -2 \end{bmatrix} \end{aligned}$$

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

$$\begin{aligned} (\%i45) & \text{A}^{\wedge 2} \\ (\%o45) & \begin{bmatrix} 5 & 5 & 6 \\ 5 & 1 & -6 \\ -1 & 3 & 10 \end{bmatrix} \end{aligned}$$

y " $\wedge n$ " eleva cada entrada de la matriz a  $n$ .

$$\begin{aligned} (\%i46) & \text{A}^{\wedge 2} \\ (\%o46) & \begin{bmatrix} 1 & 4 & 9 \\ 1 & 0 & 9 \\ 4 & 1 & 1 \end{bmatrix} \end{aligned}$$

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

$$\begin{aligned} (\%i47) & \text{A.a;} \\ (\%o47) & \begin{bmatrix} 8 \\ 2 \\ 3 \end{bmatrix} \end{aligned}$$

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

```
(%i48) a.A
(%o48) [1,3,8]
```

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

```
(%i49) 2*A;
(%o49) [ 2  4  6
        -2  0  6
         4  2 -2]
```

### 3.2.2 Otras operaciones usuales

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

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

`transpose`

```
(%i50) transpose(A);
(%o50) [ 1 -1  2
        2  0  1
        3  3 -1]
```

calcular el determinante,

`determinant`

```
(%i51) determinant(A);
(%o51) 4
```

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

`inverse`

```
(%i52) invert(A);
```

```
(%o52)  [ -3/4  5/4  3/2 ]
         [ 5/4 -7/4 -3/2 ]
         [ -1/4 3/4  1/2 ]
```

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`

```
(%i53)  m:matrix([1,3,0,-1],[3,-1,0,6],[5,-3,1,1])$
(%i54)  rank(m);
(%o54)  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:

```
(%i55)  triangularize(m);
(%o55)  [ 1  3  0  -1 ]
         [ 0 -10 0  9 ]
         [ 0  0 -10 102 ]
```

Cualquiera de estos métodos es más rápido 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

```
(%i56)  minor(A,2,1);
(%o56)  [ 2  3 ]
         [ 1 -1 ]
```

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:

```
(%i57)  submatrix(2,m,1,4);
(%o57)  [ 3  0 ]
         [-3 1 ]
```

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

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

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:

```
(%i58) nullspace(matrix([1,2,4],[-1,0,2]));
```

```
(%o58) span( $\begin{bmatrix} -4 \\ 6 \\ -2 \end{bmatrix}$ )
```

### 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.

<code>diagmatrix(n,x)</code>	matriz diagonal $n \times n$ con $x$ en la diagonal
<code>entermatrix(m,n)</code>	definir matriz $m \times n$
<code>genmatrix</code>	genera una matriz mediante una regla
<code>matrix[i,j]</code>	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:

entermatrix

```
(%i59) 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.
```

```
(%o59)  $\begin{bmatrix} 1 & 2 & 4 \\ -1 & 0 & 2 \end{bmatrix}$ 
```

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.

diagmatrix

```
(%i60) diagmatrix(4,1);
```

```
(%o60)  $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ 
```

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

```
(%i61) a[i,j]:=i*j;
(%o61) aij:=ij
```

`genmatrix` y luego utilizamos `genmatrix` para construir la matriz ( $3 \times 3$  en este caso):

```
(%i62) genmatrix(a,3,3);
(%o62)  $\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$ 
```



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?

```
(%i63) matrixp(a);
(%o63) 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:

```
(%i64) c:genmatrix(a,4,5);
(%o64)  $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 6 & 8 & 10 \\ 3 & 6 & 9 & 12 & 15 \\ 4 & 8 & 12 & 16 & 20 \end{bmatrix}$ 
```

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):

```
(%i65) c[2,3];
(%o65) 6
```

### 3.2.4 Valores propios

<code>charpoly(matriz, variable)</code>	polinomio característico
<code>eigenvalues(matriz)</code>	valores propios de la matriz
<code>eigenvectors(matriz)</code>	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.

`charpoly`

```
(%i66) S:matrix([-11/15,-2/15,-4/3],[-17/15,16/15,-1/3],[-8/15,4/15,5/3]);
```

```
(%o66) 
$$\begin{bmatrix} -\frac{11}{15} & -\frac{2}{15} & -\frac{4}{3} \\ -\frac{17}{15} & \frac{16}{15} & -\frac{1}{3} \\ -\frac{8}{15} & \frac{4}{15} & \frac{5}{3} \end{bmatrix}$$

```

```
(%i67) charpoly(S,x);
```

```
(%o67) 
$$\left(\left(\frac{16}{15} - x\right)\left(\frac{5}{3} - x\right) + \frac{4}{45}\right)\left(-x - \frac{11}{15}\right) + \frac{2\left(-\frac{17\left(\frac{5}{3} - x\right) - 8}{15} - \frac{8}{45}\right) - 4\left(\frac{8\left(\frac{16}{15} - x\right) - 68}{225}\right)}{15}$$

```

```
(%i68) expand(%);
```

```
(%o68) -x^3+2x^2+x-2
```

Por tanto, sus valores propios son

```
(%i69) solve(% ,x);
```

```
(%o69) [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.

`eigenvalues`

```
(%i70) eigenvalues(S);
```

```
(%o70) [[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`:

`eigenvectors`

```
(%i71) eigenvectors(S);
```

```
(%o71) [[[2, -1, 1], [1, 1, 1]], [1, -1/2, -2], [1, 4/7, 1/7], [1, 7, -2]]
```

La respuesta es, en este 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

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

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

- 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  $3 \times 3$ , 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:

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

**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=range(2, 21)`, `listados=range(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  $k = 1000$ .



# Números complejos. Resolución de ecuaciones

## 4

*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. Antes de comentar cómo podemos resolver ecuaciones, comencemos explicando cómo operar con números complejos.

### 4.1 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)   $\sqrt{2}i$ 
```

Correcto. Ya hemos comentado antes 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)  2i+5
```

<code>rectform(<i>expresión</i>)</code>	<i>expresión</i> en forma cartesiana o binómica
<code>realpart(<i>expresión</i>)</code>	parte real de <i>expresión</i>
<code>imagpart(<i>expresión</i>)</code>	parte imaginaria de <i>expresión</i>
<code>polarform(<i>expresión</i>)</code>	forma polar de <i>expresión</i>
<code>abs(<i>expresión</i>)</code>	módulo o valor absoluto de <i>expresión</i>
<code>carg(<i>expresión</i>)</code>	argumento de <i>expresión</i>
<code>conjugate(<i>expresión</i>)</code>	conjugado de <i>expresión</i>
<code>demoivre(<i>expresión</i>)</code>	expresa el número complejo utilizando senos y cosenos
<code>exponentialize(<i>expresión</i>)</code>	expresa el número complejo utilizando exponenciales

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)  $\frac{3i+2}{1-i}$ 
(%i4) rectform((2+3*i)/(1-i));
(%o4)  $\frac{5i}{2} - \frac{1}{2}$ 
```

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)  $\sqrt{10}e^{i \operatorname{atan}(3)}$ 
```

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)  $\operatorname{atan}(3)$ 
(%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)  $\frac{\sqrt{2}i}{2} + \frac{\sqrt{2}}{2}$ 
```

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)  $\frac{\%i \pi}{2}$ 
(%i14) rectform(log(-3));
(%o14)  $\log(3)+\%i \pi$ 
```

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)  $\frac{(\%e+\%e^{-1})(\%e^{\%i}+\%e^{-\%i})}{4} - \frac{(\%e-\%e^{-1})(\%e^{\%i}-\%e^{-\%i})}{4}$ 
```

y `demoivre` utiliza senos y cosenos en la salida en lugar de las exponenciales:

```
(%i18) demoivre(%);
(%o18)  $\frac{(\%e+\%e^{-1}) \cos(1)}{2} - \frac{(\%e-\%e^{-1}) i \sin(1)}{2}$ 
```

## 4.2 Ecuaciones y operaciones con ecuaciones

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

$expresión1=expresión2$	ecuación
<code>lhs (expresión1=expresión2)</code>	expresión1
<code>rhs (expresión1=expresión2)</code>	expresión2

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

```
(%i19) 3*x^2+2*x+x^3-x^2=4*x^2;
(%o19)  $x^3+2x^2+2x=4x^2$ 
```

además podemos asignarle un nombre para poder referirnos a ella

```
(%i20) eq:3*x^2+2*x+x^3-a*x^2=4*x^2;
```

```
(%o20) x^3-ax^2+3x^2+2x=4x^2
```

y operar como con cualquier otra expresión

```
(%i21) eq-4*x^2;
```

```
(%o21) 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.

```
(%i22) lhs(eq);
```

```
(%o22) x^3-ax^2+3x^2+2x
```

## 4.3 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.

### 4.3.1 La orden `solve`

`solve` La orden `solve` nos da todas las soluciones, ya sean reales o complejas de una ecuación o sistema de ecuaciones.

```
(%i23) solve(x^2-3*x+1=0,x);
```

```
(%o23) [x=-\frac{\sqrt{5}-3}{2}, x=\frac{\sqrt{5}+3}{2}]
```

<code>solve(ecuación, incógnita)</code>	resuelve la ecuación
<code>solve([ecuaciones], [variables])</code>	resuelve el sistema
<code>multiplicities</code>	guarda la multiplicidad de las soluciones

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

```
(%i24) eq1:x^3-a*x^2-x^2+2*x=0;
```

```
(%o24) x^3-ax^2-x^2+2x=0
```



```
(%i25) solve(eq1,x);
```

```
(%o25) [x=-\frac{\sqrt{a^2+2a-7}-a-1}{2}, x=\frac{\sqrt{a^2+2a-7}+a+1}{2}, x=0]
```

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

```
(%i26) solve(x^2+2*x=3);
```

```
(%o26) [x=-3, x=1]
```

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

```
(%i27) solve(x^2+2*x);
```

```
(%o27) [x=-2, x=0]
```

```
(%i28) solve(x^2+2*x=0);
```

```
(%o28) [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  $x^7 - 2x^6 + 2x^5 - 2x^4 + x^3$  tiene las raíces 0, 1,  $i$ ,  $-i$ ,

`multiplicities`

```
(%i29) solve(x^7-2*x^6+2*x^5-2*x^4+x^3);
```

```
(%o29) [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?

```
(%i30) multiplicities;
```

```
(%o30) [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.



```
(%i31) 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:

```
(%i32) solve(eq1,a);
```

```
(%o32) [a= $\frac{x^2-x+2}{2}$ ]
```

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

```
(%i33) solve(sin(x)*cos(x)=0,x);
```

```
'solve' is using arc-trig functions to get a solution.  
Some solutions will be lost.
```

```
(%o33) [x=0,x= $\frac{\%pi}{2}$ ]
```

¿Qué ocurre aquí? La expresión  $\sin(x) \cos(x)$  vale cero cuando el seno o el coseno se anulen. Para calcular la solución de  $\sin(x) = 0$  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”, los polinomios, 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.

```
(%i34) eq:x+3=sqrt(x+1);
```

```
(%o34) x+3=sqrt(x+1)
```

```
(%i35) solve(eq,x);
```

```
(%o35) [x= $\sqrt{x+1}-3$ ]
```

```
(%i36) solve(eq^2);
```

```
(%o36) [x= $-\frac{\sqrt{7}i+5}{2}$ , x= $\frac{\sqrt{7}i-5}{2}$ ]
```

## 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:

```
(%i37) sol:solve(x^2-4*x+3);
```

```
(%o37) [x=3,x=1]
```

La primera solución es

```
(%i38) rhs(part(sol,1));
```

```
(%o38) 3
```

y la segunda

```
(%i39) rhs(part(sol,2));
```

```
(%o39) 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`:

```
(%i40) sol:map(rhs,solve(x^2-4*x+3));
```

```
(%o40) [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

```
(%i41) solve([x^2+y^2=1,(x-2)^2+(y-1)^2==4],[x,y]);
```

```
(%o41) [[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  $x^2 + y^2 = 1$ :

```
(%i42) solve([x^2+y^2=1,x+y=5],[x,y]);
```

```
(%o42) [[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 referirse a estos. Por ejemplo,

```
(%i43) solve([x+y+z=3,x-y=z],[x,y,z]);
```

```
(%o43) [[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)

```
(%i44) solve[[x+y=0,x+y=1],[x,y]];
Inconsistent equations: [2]
-- an error. To debug this try debugmode(true);
```

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

```
(%i45) solve[(x+1)^2=x^2+2x+1,x]
(%o45) [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*?

```
(%i46) solve((x+y)^2=x^2+2*x*y+y^2,[x,y]);
Dependent equations eliminated: (1)
(%o46) [[x=%r3,y=%r2]]
```

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

### 4.3.2 Sistemas de ecuaciones lineales

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

**linsolve** 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.

```
(%i47) eq: [x+y+z+w=1,x-y+z-w=-2,x+y-w=0]$
(%i48) linsolve(eq,[x,y,z]);
(%o48) [x=4w-3/2,y=-2w-3/2,z=1-2w]
```

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

### 4.3.3 Algsys

<code>algsys([ecuaciones],[variables])</code>	resuelve la ecuación o ecuaciones
<code>realonly</code>	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. `algsys`

```
(%i49) eq:x^2-4*x+3;
(%o49) x^2-4*x+3
(%i50) algsys([eq],[x]);
(%o50) [[x=3],[x=1]]
```

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

```
(%i51) solve(eq:x^6+x+1);
(%o51) [0=x^6+x+1]
(%i52) algsys([eq],[x]);
(%o52) [[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. `realonly`

```
(%i53) eq:x^4-1=0$
(%i54) realonly;
(%o54) false
(%i55) algsys([eq],[x]);
(%o55) [[x=1],[x=-1],[x=%i],[x=-%i]]
(%i56) realonly:true$
(%i57) algsys([eq],[x]);
(%o57) [[x=1],[x=-1]]
```

### 4.3.4 Soluciones aproximadas

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.

<code>allroots(polimonio)</code>	soluciones aproximadas del polinomio
<code>realroots(polimonio)</code>	soluciones aproximadas reales del polinomio

Estos órdenes nos dan todas las soluciones y las soluciones reales de un polinomio en una variable y pueden ser útiles en polinomios de grado alto.

```
(%i58) eq:x^9+x^7-x^4+x)$
(%i59) allroots(eq);
[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,
(%o59) x=0.51104079208431%i-0.80986929589487,
x=-0.51104079208431%i-0.80986929589487,
x=1.189238256723466%i+0.29426593530541,
x=0.29426593530541-1.189238256723466%i]
(%i60) realroots(eq);
(%o60) [x=0]
```

### 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: las 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.

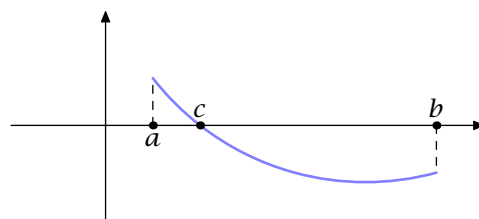
<code>find_root (f(x), x, a, b)</code>	solución de $f$ en $[a, b]$
--	-----------------------------

Teorema de los  
ceros de Bolzano

**Teorema 4.2.** Sea  $f : [a, b] \rightarrow \mathbb{R}$  una función continua verificando que  $f(a)f(b) < 0$ , entonces existe  $c \in ]a, b[$  tal que  $f(c) = 0$ .

**Ejemplo 4.3.** 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  $e^x + \ln(x) = 0$  tiene solución, estudiamos la función  $f(x) = e^x + \ln(x)$ : es continua en  $\mathbb{R}^+$  y se puede comprobar que  $f(e^{-10}) < 0$  y  $0 < f(e^{10})$ . Por tanto, la ecuación  $e^x + \ln(x) = 0$  tiene al menos una solución entre  $e^{-10}$  y  $e^{10}$ . En particular, tiene solución en  $\mathbb{R}^+$ .

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.



`find_root`

**Figura 4.1** Teorema de los ceros de Bolzano

```
(%i61) f(x):=exp(x)+log(x);
(%o61) f(x):=exp(x)+log(x)
```

Buscamos un par de puntos donde cambie de signo

```
(%i62) f(1)
(%o62) %e
(%i63) f(exp(-3));
(%o63) %e%e^-3+3
```

¿Ese número es negativo?

```
(%i64) is(f(exp(-3))<0);
(%o64) true
```

o bien,

```
(%i65) f(exp(-3)), numer;
(%o65) -1.948952728663784
```

Vale, ya que tenemos dos puntos donde cambia de signo podemos utilizar `find_root`:

```
(%i66) find_root(f(x), x, exp(-3), 1);
(%o66) 0.26987413757345
```

**Observación 4.4.** 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.



## 4.4 Hágalo usted mismo

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 las 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. 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.

#### 4.4.1 Breves conceptos de programación

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)

- `var:valor1` nos sitúa en las condiciones de comienzo del bucle.
- `cond` dirá a *Maxima* el momento de detener el proceso.
- `step valor2` expresará la forma de aumentar la condición inicial.
- `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 35:



```
(%i67) for i:1 step 1 thru 5 do print(7*i)
      7
      14
      21
      28
      35
(%o67) done
```

Se puede conseguir el mismo efecto sumando en lugar de multiplicando. Por ejemplo, los múltiplos de 5 hasta 25 son

```
(%i68) for i:1 step 5 thru 25 print(i);
      5
      10
      15
      20
      25
(%o68) done
```

**Ejemplo 4.5.** 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.

```
(%i69) suma:0$
      for i:1 thru 100 do suma:suma+i^2$
      print("la suma de los 100 primeros naturales vale ",suma);
(%o69) la suma 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`).

```
while cond do expr  bucle while
unless cond do expr  bucle unless
do expr             bucle for
return (var)       bucle for
```

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.

```
(%i70) i:0;
(%o70) 0
(%i71) while cos(i)>0 do (print(i,cos(i)),i:i+0.3);
0 1
0.30000001192093 0.95533648560273
0.60000002384186 0.82533560144755
0.90000003576279 0.62160994025671
1.200000047683716 0.36235771003359
1.500000059604645 0.070737142212368
(%o71) done
```

## 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*.

<code>if condición then expr1 else expr2</code>	condicional if-then-else
<code>if condición then expr</code>	condicional if-then

Las 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:

```
(%i72) if log(2)<0 then x:5 else 3;
(%o72) 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.

```
(%i73) for i:1 thru 10 do
      (
      if log(i)<2 then print("el logaritmo de",i,"es menor que 2")
      else (x:i,return(x))
      )$
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

(%o73)
```

**Observación 4.6.** 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.

#### 4.4.2 Método de Bisección

En este método sólo es necesario que la función  $f$  sea continua en el intervalo  $[a, b]$  y verifique  $f(a)f(b) < 0$ . En estas condiciones, el Teorema de Bolzano nos asegura la existencia de una solución de la ecuación en  $[a, b]$ . El siguiente paso consiste en tomar como nueva aproximación,  $c = \frac{a+b}{2}$  (el punto medio del segmento  $[a, b]$ ). Si  $f(c) = 0$ , hemos encontrado una solución de la ecuación y por tanto hemos terminado. Si  $f(c) \neq 0$ , consideramos como nuevo intervalo, o bien  $[a, c]$  (si  $f(a)f(c) < 0$ , o bien  $[c, b]$  si es que  $f(c)f(b) < 0$  y repetimos la estrategia en el nuevo intervalo.

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 “ $Ex$ ” a la exactitud prefijada, entonces para conseguir dicha precisión, el número “ $n$ ” de iteraciones necesarias deberá satisfacer

$$\frac{b-a}{2^n} < Ex$$

así,

$$n = E \left[ \log_2 \left( \frac{b-a}{Ex} \right) \right] + 1,$$

donde  $E[\cdot]$  denota la “parte entera” de un número (esto es, el mayor de los enteros que son menores que el número). Para obtener la “parte entera” de un número que está expresado en notación decimal, *Maxima* tiene el comando `entier`.

Para definir un algoritmo de cálculo de la sucesión de aproximaciones de este método mediante *Maxima*, vamos a resolver como ejemplo la ecuación  $x^6 + x - 5 = 0$  en el intervalo  $[0, 2]$ .

Definiremos en primer lugar la función, el intervalo y la exactitud. Seguidamente calcularemos el número “*P*” de iteraciones (o pasos) necesarias, para a continuación llevar a cabo el cálculo de dichas aproximaciones.

```
(%i74) f(x):=x^6+x-5$
log2(x):=log(x)/log(2)$
a:0$
b:2$
Ex:10^(-6)$
pasos:entier(log((b-a)/Ex))+1$
for i:1 thru pasos do
(
c:(a+b)/2,
(if f(c)=0
then return(c)
else if f(a)*f(c)<0 then b:c else a:c)
)$
print("la solucion es ",c);
(%o74) la solucion es  $\frac{20425}{16384}$ 
```

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  $10^{-10}$ .

Más tarde compararemos su efectividad con la del siguiente método.

#### 4.4.3 Método de Newton-Raphson

El método para la construcción del algoritmo que vamos a estudiar ahora es conocido con el nombre de “método de Newton-Raphson” debido a sus autores. Este método nos proporciona un algoritmo para obtener una sucesión de aproximaciones. Para asegurar la convergencia de la sucesión (hacia la solución de la ecuación), bajo ciertas condiciones, usaremos el Teorema de Newton-Raphson, cuyo enunciado daremos más adelante.

La forma de construir los términos de la sucesión de aproximaciones es bastante sencilla y responde a una idea muy intuitiva. Primero suponemos que  $f$  es derivable al menos dos veces, con primera derivada no nula en el intervalo donde trabajamos. 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:

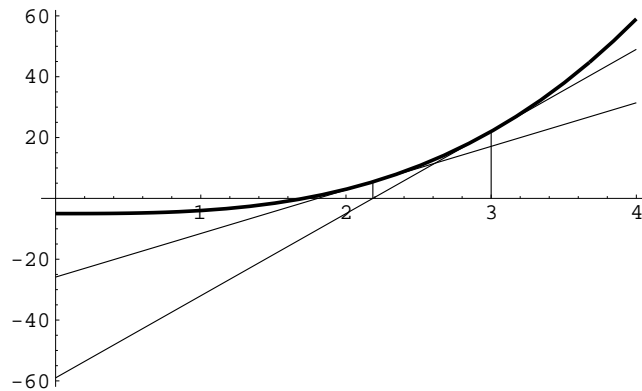
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Para comprender el algoritmo observa la Figura 4.2 donde se ve cómo se generan los valores de las aproximaciones.

**Teorema 4.7.** 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  $[a, b]$ .



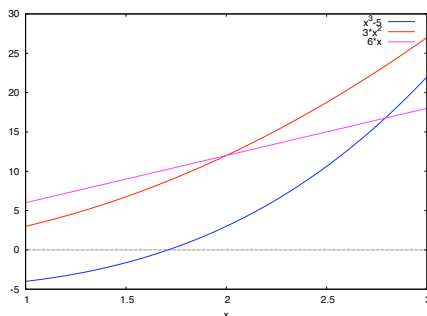
**Figura 4.2** Aproximaciones con el método de Newton-Raphson

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  $[1, 3]$ .

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.

```
(%i75) f(x):=x^3-5$
(%i76) define(df(x),diff(f(x),x))$
(%i77) define(df2(x),diff(f(x),x,2))$
(%i78) plot2d([f(x),df(x),df2(x)], [x,1,3]);
(%o78)
```



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.

```
(%i79) 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. De todas formas, no es nuestra intención y se escapa de los contenidos de este curso profundizar más en este método. En cursos superiores lo verás con más detalle.

## 4.5 Ejercicios

### Números complejos

#### Ejercicio 4.1

Efectuar las operaciones indicadas:

- $\frac{2-3i}{4-i}$
- $\frac{(2+i)(3-2i)(1+2i)}{(1-i)^2}$
- $(2i-1)^2 \left[ \frac{4}{1-i} + \frac{2-i}{1+i} \right]$
- $\frac{i^4+i^9+i^{16}}{2-i^5+i^{10}-i^{15}}$



**Ejercicio 4.9**

- 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.

**Ejercicio 4.10**

Encontrar una solución del siguiente sistema de ecuaciones cerca del origen:  $\sin(x) \cos(y) = \frac{1}{4}$ ,  $xy = 1$ .

**Ejercicio 4.11**

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 4.12**

Dado un número positivo  $x$ , se puede conseguir que la suma

$$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

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 4.13**

Reescribe el método de Newton-Raphson añadiendo una condición de salida (cuándo el error se menor que una cierta cantidad) y que compruebe que la segunda derivada está "lejos" de cero en cada paso.



# Límites y continuidad

## 5

Uno de los primeros conceptos que se presentan en un curso de Cálculo es el de continuidad. Este concepto está íntimamente ligado al concepto de límite. En clase hemos utilizado sucesiones para definir límite funcional. En este capítulo veremos cómo usar *Maxima* para resolver algunos problemas relacionados con todos estos conceptos.

### 5.1 Límites

El cálculo de límites se realiza con la orden `limit`. Con ella podemos calcular límites de funciones o de sucesiones en un número, en  $+\infty$  o en  $-\infty$ . También podemos usar el menú **Análisis**→**Calcular límite**. Ahí podemos escoger, además de a qué función le estamos calculando el límite, a qué tiende la variable incluyendo los valores “especiales” como  $\pi$ ,  $e$  o infinito. Además de esto, también podemos marcar si queremos calcular únicamente el límite por la derecha o por la izquierda.



`limit`

<code>limit (expr,x,a)</code>	$\lim_{x \rightarrow a} expr$
<code>limit (expr,x,a,plus)</code>	$\lim_{x \rightarrow a^+} expr$
<code>limit (expr,x,a,minus)</code>	$\lim_{x \rightarrow a^-} expr$
<code>inf</code>	$+\infty$
<code>minf</code>	$-\infty$
<code>und</code>	indefinido
<code>ind</code>	indefinido pero acotado

El cálculo de límites con *Maxima*, como puedes ver, es sencillo. Sabe calcular límites de cocientes de polinomios en infinito


```
(%i1) limit(n/(n+1),n,inf);
(%o1) 1
```

o en  $-\infty$ ,

```
(%i2) limit((x^2+3*x+1)/(2*x+3),x,minf);
```



```
(%i8) limit(abs(x)/x,x,0);
(%o8) und
```

**Observación 5.1.** La acotación que incluye `ind` es una información adicional que da *Maxima*. Si no sabe si es cierta la acotación o, directamente, no es cierta, entonces responde `und` pero esto no quiere decir que la función a la que le estamos calculando el límite no esté acotada: solamente quiere decir que no sabe si lo está o no. 

En este último límite lo que ocurre es que tenemos que estudiar los límites laterales

```
(%i9) limit((abs(x)/x,x,0,plus);
(%o9) 1
(%i10) limit(abs(x)/x,x,0,minus);
(%o10) -1
```

Por tanto, no existe el límite puesto que los límites laterales no coinciden. Si recuerdas la definición de función derivable, acabamos de comprobar que la función valor absoluto no es derivable en el origen.

## 5.2 Sucesiones

En clase hemos visto cómo calcular límites de sucesiones, pero ¿cómo podemos calcular esos límites con *Maxima*? Bueno, en la práctica hemos visto dos tipos de sucesiones dependiendo de cómo estaba definidas. Por un lado tenemos aquellas definidas mediante una fórmula que nos vale para todos los términos. Por ejemplo, la sucesión que tiene como término general  $x_n = \left(1 + \frac{1}{n}\right)^n$ . En este caso no hay ningún problema en definir

```
(%i11) f(n):=(1+1/n)^n;
(%o11) f(n):=(1+1/n)^n
```

y se puede calcular el límite en  $+\infty$  sin ninguna dificultad

```
(%i12) limit(f(n), n, inf);
(%o12) %e
```

La situación es diferente cuando no tenemos una fórmula para el término general como, por ejemplo, cuando la sucesión está definida por recurrencia. Veamos un ejemplo. Consideremos la sucesión que tiene como término general  $c_1 = 1$  y  $c_{n+1} = \frac{c_n}{1+c_n}$  para cualquier natural  $n$ . Podemos definirla utilizando una lista definida, como no, por recurrencia:

```
(%i13) c[1]:1;
(%o13) 1
```

```
(%i14) c[n]:=c[n-1]/(1+c[n-1]);
```

```
(%o14) c_n := \frac{c_{n-1}}{1+c_{n-1}}
```

Si somos capaces de encontrar una fórmula para el término general, podemos calcular el límite. Con lo que tenemos hasta ahora no vamos muy lejos:

```
(%i15) limit(c[n],n,inf);
```

```
Maxima encountered a Lisp error:
```

```
Error in PROGN [or a callee]: Bind stack overflow.
```

```
Automatically continuing.
```

```
To reenale the Lisp debugger set *debugger-hook* to nil.
```

Podemos demostrar por inducción que la sucesión es, en este caso, decreciente y acotada inferiormente. Una vez hecho, la sucesión es convergente y el límite  $L$  debe verificar que  $L = \frac{L}{1+L}$ . Esta ecuación sí nos la resuelve *Maxima*

```
(%i16) solve(L/(1+L)=0,L);
```

```
(%o16) [L=0]
```

con lo que tendríamos demostrado que el límite es 0.

**Observación 5.2.** Para esta sucesión en concreto, sí se puede encontrar una fórmula para el término general. De hecho existe un módulo, `solve_rec`, que resuelve justo este tipo de problemas.

```
(%i17) kill(all);
```

```
(%o0) done
```

```
(%i1) load(solve_rec)$
```

```
(%i2) solve_rec(c[n]=c[n-1]/(1+c[n-1]),c[n]);
```

```
(%o2) c_n = \frac{n+%k_1+1}{n+%k_1} - 1
```

que, simplificando, nos queda

```
(%i3) ratsimp(%)
```

```
(%o3) c_n = \frac{1}{n+%k_1}
```

si a esto le añadimos que  $c_1 = 1$  obtenemos que  $c_n = \frac{1}{1+n}$ .

De todas formas no hay que ilusionarse demasiado. Encontrar una fórmula para el término general es *difícil* y lo normal es no poder hacerlo. Es por ello que no vamos a entrar en más detalles con `solve_rec`. Lo único que podemos hacer con *Maxima* es calcular términos. Por ejemplo, `solve_rec` no es capaz de encontrar el término general de la sucesión  $x_1 = 1, x_n = \sqrt{1+x_{n-1}}, \forall n \in \mathbb{N}$ . En cambio, no tiene ninguna dificultad en calcular tanto términos como se quiera,

`solve_rec`

```
(%i4) x[1]:1;
(%o4) 1
(%i5) x[n]:=sqrt(1+x[n-1]);
(%o5)  $x_n = \sqrt{1+x_{n-1}}$ 
(%i6) x[10];
(%o6)  $\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{2+1+1+1+1+1+1+1+1+1}}}}}}}}}}}}$ 
(%i7) %,numer;
(%o7) 1.618016542231488
```

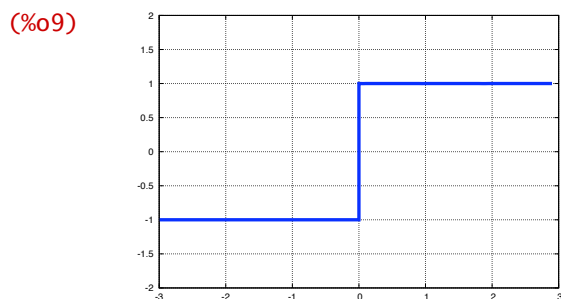
pero seremos nosotros los que tendremos que demostrar la convergencia estudiando la monotonía y la acotación de la sucesión.

### 5.3 Continuidad

El estudio de la continuidad de una función es inmediato una vez que sabemos calcular límites. Una función  $f : A \subset \mathbb{R} \rightarrow \mathbb{R}$  es continua en  $a \in A$  si  $\lim_{x \rightarrow a} f(x) = f(a)$ . Conocido el valor de la función en el punto, la única dificultad es, por tanto, saber si coincide o no con el valor del límite.

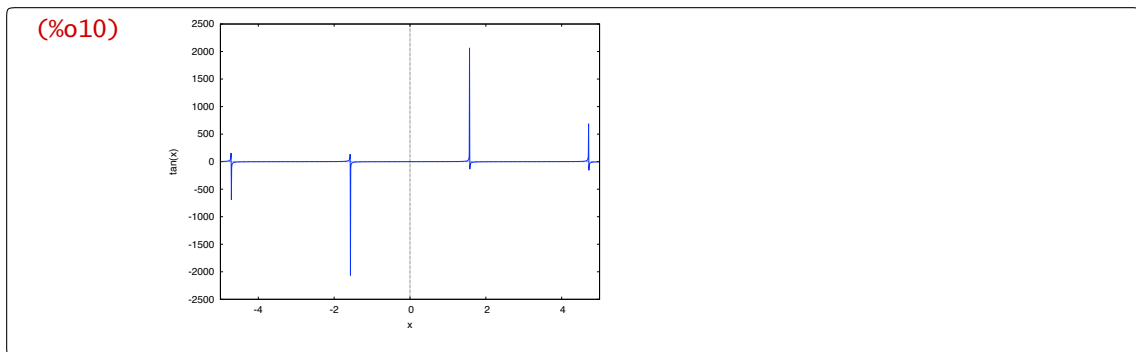
Con respecto a las funciones discontinuas, la gráfica puede darnos una idea del tipo de discontinuidad. Si la discontinuidad es evitable, es difícil apreciar un único pixel en la gráfica. Una discontinuidad de salto es fácilmente apreciable. Por ejemplo, la función signo, esto es,  $\frac{|x|}{x}$ , tiene un salto en el origen que *Maxima* une con una línea vertical.

```
(%i8) load(draw)$
(%i9) draw2d(color=blue,
  explicit(abs(x)/x,x,-3,3),
  yrange=[-2,2],
  grid=true);
```

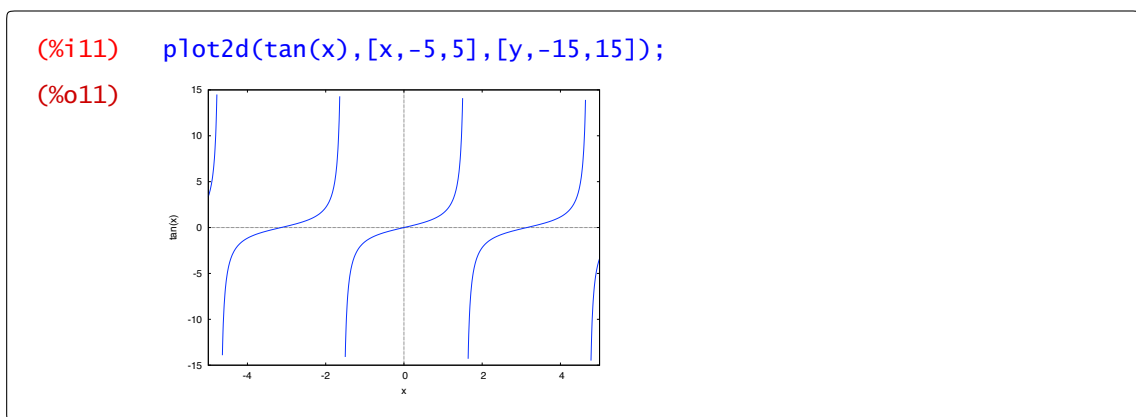


Cuando el salto es infinito o, lo que es lo mismo, cuando la función tiene una asíntota vertical, la primera dificultad que se encuentra *Maxima* es escoger un rango adecuado para representarla:

```
(%i10) plot2d(tan(x), [x,-5,5]);
```



En estos casos tenemos que ayudar nosotros a *Maxima* restringiendo el rango donde representamos la función



## 5.4 Ejercicios

### Ejercicio 5.1

Estudia la continuidad de la función  $f: \mathbb{R} \rightarrow \mathbb{R}$  definida como  $f(x) = x \cdot \ln|x|$  si  $x \neq 0$  y  $f(0) = 0$ .

### Ejercicio 5.2

Sean  $a$  y  $b$  dos números reales verificando  $b < 0 < a$ ; estudia el comportamiento en cero de la función

$$f(x) = \arctan\left(\frac{a}{x}\right) - \arctan\left(\frac{b}{x}\right), \quad \forall x \in \mathbb{R}^*.$$

### Ejercicio 5.3

Estudia la continuidad de la función  $f(x) = \arctan\left(\frac{1+x}{1-x}\right)$  con  $x \neq 1$ , así como su comportamiento en  $1$ ,  $+\infty$  y  $-\infty$ .

### Ejercicio 5.4

- Dibuja una función continua cuya imagen no sea un intervalo.
- Dibuja una función definida en un intervalo cuya imagen sea un intervalo y que no sea continua.
- Dibuja una función continua en todo  $\mathbb{R}$ , no constante y cuya imagen sea un conjunto (obligatoriamente un intervalo) acotado.

- d) Dibuja una función continua en  $[0, 1[$  tal que  $f([0, 1[)$  no sea acotado.
- e) Dibuja una función continua definida en un intervalo abierto acotado y cuya imagen sea un intervalo cerrado y acotado.

**Ejercicio 5.5**

Consideremos la función  $f : [0, 1] \rightarrow \mathbb{R}$  definida como  $f(x) = \frac{1}{2} (\cos(x) + \sin(x))$ .

- a) Utiliza que  $f([0, 1]) \subset [0, 1]$  para probar que existe  $x \in [0, 1]$  tal que  $f(x) = x$  (sin utilizar *Maxima*). A dicho punto se le suele llamar un *punto fijo* de la función  $f$ .
- b) Se puede demostrar que la sucesión  $x_1 = 1, x_{n+1} = f(x_n)$ , para cualquier natural  $n$  tiende a un punto fijo. Utiliza un bucle para encontrar un punto fijo con una exactitud menor que  $10^{-5}$ .





# Derivación

## 6

En este capítulo vamos a aprender a calcular y evaluar derivadas de cualquier orden de una función; representar gráficamente rectas tangentes y normales a la gráfica de una función; calcular extremos de funciones reales de una variable y, por último, calcular polinomios de Taylor y representarlos gráficamente para aproximar una función.

### 6.1 Cálculo de derivadas

Para calcular la derivada de una función real de variable real, una vez definida, por ejemplo, como  $f(x)$ , se utiliza el comando `diff` que toma como argumentos la función a derivar, la variable con respecto a la cual hacerlo y, opcionalmente, el orden de derivación. `diff`

<code>diff(expr, variable)</code>	derivada de $expr$
<code>diff(expr, variable, n)</code>	derivada $n$ -ésima de $expr$

A este comando también podemos acceder a través del menú **Análisis**→**Derivar** o, también, a través del botón **Derivar**<sup>2</sup> en la parte inferior de *wxMaxima*. Haciéndolo de cualquiera de estas dos formas, aparece una ventana de diálogo en la que aparecen varios datos a rellenar; a saber:

- Expresión. Por defecto, *wxMaxima* rellena este espacio con % para referirse a la salida anterior. Si no es la que nos interesa, la escribimos directamente nosotros.
- respecto la variable. Se refiere a la variable respecto a la cual vamos a derivar.
- veces. Se refiere al orden de derivación.

Comencemos con un ejemplo,

```
(%i1) diff(tan(x), x);
(%o1) sec(x)2
```

La orden `diff` considera como constantes cualquier otra variable que aparezca en la expresión a derivar, salvo que explícitamente manifestemos que están relacionadas.

```
(%i2) diff(x*y*sin(x+y), x);
(%o2) ysin(y+x)+xycos(y+x)
```

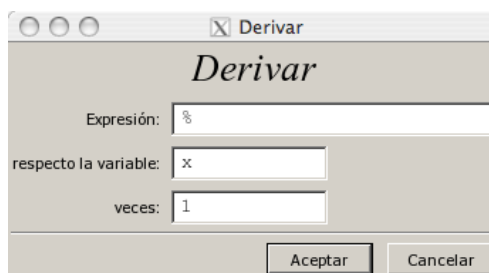


Figura 6.1 Introducir derivada

<sup>2</sup> El botón derivar sólo está disponible si en las preferencias hemos seleccionado el panel con botones completo

También podemos trabajar con funciones que previamente hayamos definido.

```
(%i3) f(x):= x^4+sin(x^2)
(%o3) f(x):=x^4+sin(x^2)
(%i4) diff(f(x),x)
(%o4) 2x cos(x^2)+4x^3
```

La tercera entrada de la orden `diff` nos permite calcular derivadas de orden superior. Por ejemplo, la cuarta derivada de  $f$  sería la siguiente.

```
(%i5) diff(f(x),x,4)
(%o5) 16x^4sin(x^2) - 12 sin(x^2) - 48x^2cos(x^2) + 24
```

### 6.1.1 Reutilizar la derivada

Las derivadas sucesivas de una función nos dan mucha información sobre la función original y con frecuencia nos hace falta utilizarlas de nuevo, ya sea para calcular puntos críticos, evaluar para estudiar monotonía o extremos relativos, etc. Es por ello que es cómodo escribir la derivada como una función. Hay varias formas en las que podemos hacerlo. Podemos, por ejemplo, utilizar la orden `define`

```
(%i6) define(g(x),diff(f(x),x));
(%o6) g(x):=2xcos(x^2)+4x^3
```

o podemos aprovechar las dobles comillas

```
(%i7) df(x):='(%o4);
(%o7) df(x):=2xcos(x^2)+4x^3
(%i8) df(1);
(%o8) 2cos(1)+4
```

**⚠ Observación 6.1.** Te recuerdo que la comilla que utilizamos para asignar a la función  $df(x)$  la derivada primera de  $f$  es la que aparece en la tecla  $\text{'}\text{'}$ , es decir, son dos apóstrofes,  $\text{'}\text{'}$ , y no hay que confundirla con las dobles comillas de la tecla  $\text{''}$ .

También podemos evaluar la derivada en un determinado punto sin necesidad de definir una nueva función,

```
(%i9) ''(diff(f(x),x),x=1;
(%o9) 2cos(1)+4
```


aunque esto deja de ser práctico cuando tenemos que calcular el valor en varios puntos.

## El operador comilla y dobles comillas

Hasta ahora no hemos utilizado demasiado, en realidad prácticamente nada, los operadores comilla y dobles comillas. Estos operadores tienen un comportamiento muy distinto: una comilla simple hace que no se evalúe, en cambio las dobles comillas obligan a una evaluación de la expresión que le sigue. Observa cuál es la diferencia cuando aplicamos ambos operadores a una misma expresión:

```
(%i10) 'diff(f(x),x)='diff(f(x),x);
(%o10)  $\frac{d}{dx}(\sin(x^2)+x^4)=2x\cos(x^2)+4x^3$ 
```

En la parte de la izquierda tenemos la derivada sin evaluar, la expresión que hemos escrito tal cual. En la derecha tenemos la derivada calculada de la función  $f$ .

**Observación 6.2.** El uso de las dobles comillas para definir la derivada de una función puede dar lugar a error. Observa la siguiente secuencia de comandos: definimos la función coseno y “su derivada”, 

```
(%i11) remfunction(all)$
(%i12) f(x):=cos(x);
(%o12) f(x):=cos(x)
(%i13) g(x):=diff(f(x),x);
(%o13) g(x):=diff(f(x),x);
(%i14) h(x):='diff(f(x),x);
(%o14) h(x):=diff(f(x),x);
```

bueno, no parece que haya mucha diferencia entre usar o no las comillas. Vamos a ver cuánto valen las funciones  $g$  y  $h$ :

```
(%i15) g(x);
(%o15) -sin(x)
(%i16) h(x);
(%o16) -sin(x)
```

Parece que no hay grandes diferencias. De hecho no se ve ninguna. Vamos a ver cuánto valen en algún punto.

```
(%i17) g(1);
Non-variable 2nd argument to diff:
1
#0: g(x=1)
-- an error. To debug this try debugmode(true);
```

Por fin, un error. Habíamos dicho que necesitábamos las comillas para que se evaluara la derivada. Sólo lo ha hecho cuando lo hemos definido pero, sin las dobles comillas, no vuelve a hacerlo y, por tanto, no sabe evaluar en 1. En la práctica `g(x) := -sin(x)` es una simple cadena de texto y no una función. Vale. Entonces, vamos con la función `h`:

```
(%i18) h(1);
Non-variable 2nd argument to diff:
1
#0: g(x=1)
-- an error. To debug this try debugmode(true);
```

¿Qué ha pasado aquí? ¿Pero si hemos puesto las comillas dobles! ¿Qué está mal? ¿Hemos escrito mal las comillas? Repásalo y verás que no. El problema es un poco más sutil: las dobles comillas afectan a lo que tienen directamente a su derecha. En la definición de la función `h` no hemos escrito entre paréntesis la derivada y las dobles comillas no afectan a todo; sólo afectan al operador `diff` pero no a la `x`. Es por eso que no la considera una variable y tampoco se puede evaluar `h` en un punto. Llegados a este punto entenderás porqué hemos recomendado que utilices el comando `define` en lugar de las comillas.

## 6.2 Rectas secante y tangente a una función

La definición de derivada de una función real de variable real en un punto  $a$  es, como conoces bien,

$$\lim_{x \rightarrow a} \frac{f(x) - f(a)}{x - a},$$

límite que denotamos  $f'(a)$ .

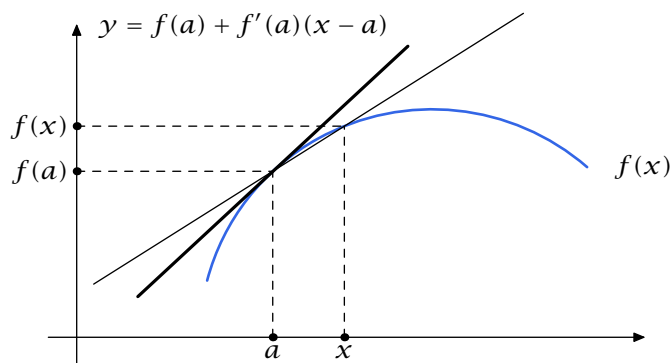


Figura 6.2 Recta tangente

En otras palabras, calculamos la recta que pasa por el punto  $(a, f(a))$  y el punto  $(x, f(x))$ , hacemos tender  $x$  a  $a$  y, en el límite, la recta que obtenemos es la recta tangente. La pendiente de dicha recta es la derivada de  $f$  en  $a$ .

Vamos a aprovechar la orden `with_slider` para representar gráficamente este proceso en un ejemplo. Consideremos la función  $f(x) = x^3 - 2x^2 - x + 2$  y su derivada, a la que notaremos  $df$ ,

```
(%i19) f(x):=x^3-2*x^2-x+2;
```

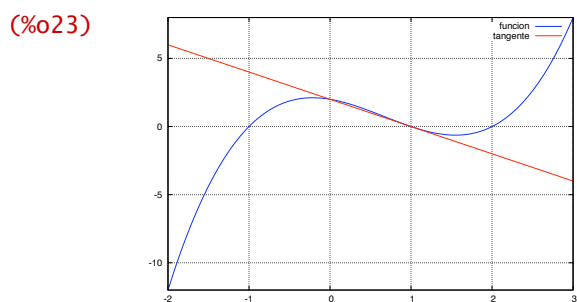
```
(%o19) f(x):=x3-2x2-x+2
(%i20) define(df(x),diff(f(x),x));
(%o20) df(x):=3x2-4x-1
```

La recta tangente a una función  $f$  en un punto  $a$  es la recta  $y = f(a) + f'(a)(x - a)$ . La definimos.

```
(%i21) tangente(x,a):=f(a)+df(a)*(x-a);
(%o21) tangente(x,a):=f(a)+df(a)*(x-a)
```

Ya podemos dibujar la función y su tangente en 1:

```
(%i22) load(draw)$
(%i23) draw2d(
    color=blue,key="función",explicit(f(x),x,-2,3),
    color=red,key="tangente",explicit(tangente(x,1),x,-2,3),
    grid=true);
```



Para dibujar la recta secante la primera cuestión es ¿cuál es la recta que pasa por  $(1, f(1))$  y por  $(x, f(x))$ ? Recordemos que la recta que pasa por un par de puntos  $(a, c)$ ,  $(b, d)$  que no estén verticalmente alineados es la gráfica de la función

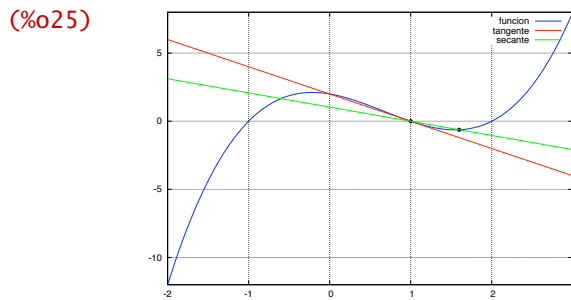
$$\text{recta}(x) = \frac{c-d}{a-b}x + \frac{ad-bc}{a-b}.$$

La definimos en *Maxima*:

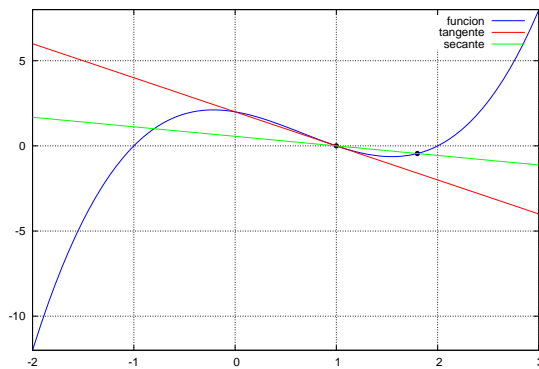
```
(%i24) recta(x,a,c,b,d):=x*(c-d)/(a-b)+(a*d-b*c)/(a-b);
(%o24) recta(x,a,c,b,d):=x(c-d)/(a-b)+ad-bc/(a-b)
```

Ahora podemos ver qué ocurre con las rectas que pasan por los puntos  $(1, f(1))$  y  $(1+h, f(1+h))$  cuando  $h$  tiende a cero. Por ejemplo para  $h = 0.6$ , tendríamos

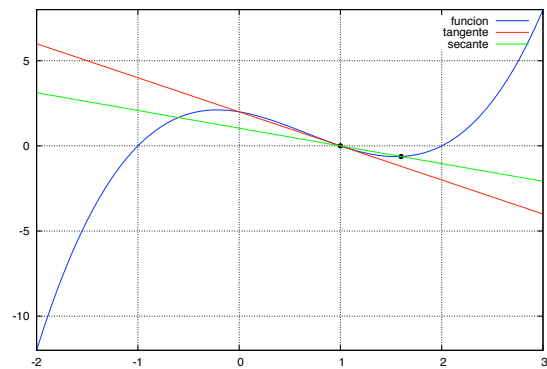
```
(%i25) draw2d(
  point_type=filled_circle,color=black,points([[1,f(1)]]),
  point_type=filled_circle,color=black,points([[1.6,f(1.6)]]),
  color=blue,key="funcion",explicit(f(x),x,-2,3),
  color=red,key="tangente",explicit(tangente(x,1),x,-2,3),
  color=green,key="secante",
  explicit(recta(x,1,f(1),1+0.6,f(1.6)),x,-2,3),
  grid=true)$
```



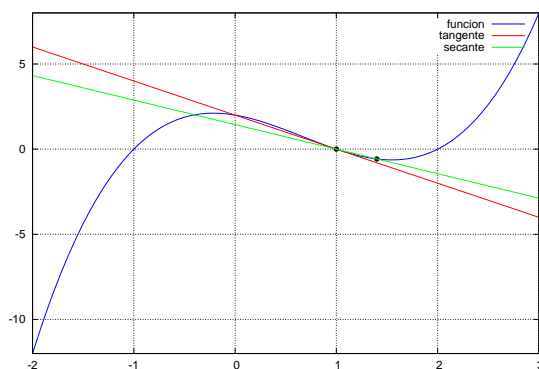
Si te fijas, en la gráfica anterior hemos añadido también el par de puntos que definen la recta secante. En la Figura 6.3 puedes ver el resultado para  $h = 0.2, 0.4, 0.6$ , y  $0.8$ .



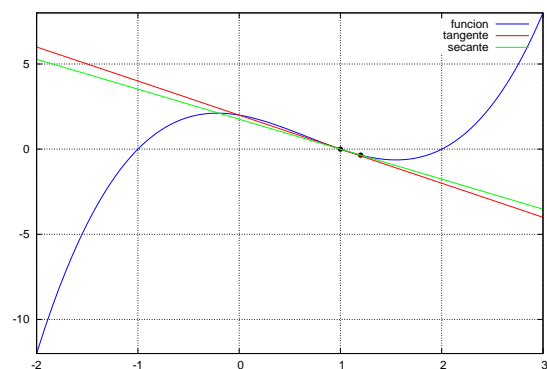
$h = 0.8$



$h = 0.6$



$h = 0.4$



$h = 0.2$

Figura 6.3 Rectas secantes y tangente

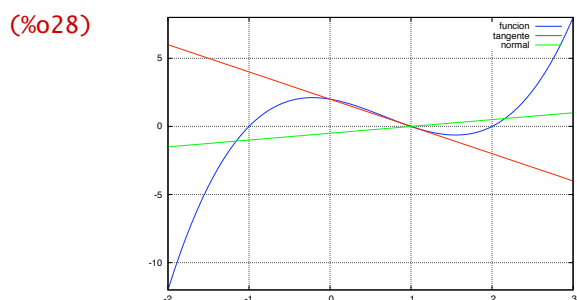
Podemos unir todo lo que hemos hecho y usar la capacidad de *wxMaxima* para representar gráficos en función de un parámetro. Dibujemos la función, la recta tangente y las secantes para  $h = 0.1, 0.2, \dots, 2$ :

```
(%i26) with_slider(
      n,0.1*reverse(range(1,20)),
      [f(x),tangente(x,1),recta(x,1,f(1),1+n,f(1+n))],
      [x,-2,3]);
```

### 6.2.1 Recta normal

La recta normal es la recta perpendicular a la recta tangente. Su pendiente es  $-1/f'(a)$  y, por tanto, tiene como ecuación  $y = f(a) - \frac{1}{f'(a)}(x - a)$ . Con todo lo que ya tenemos hecho es muy fácil dibujarla.

```
(%i27) normal(x,a):=f(a)-df(a)^(-1)*(x-a)$
(%i28) draw2d(
      point_type=filled_circle,color=black,points([[1,f(1)]]),
      color=blue,key="funcion",explicit(f(x),x,-2,3),
      color=red,key="tangente",explicit(tangente(x,1),x,-2,3),
      color=green,key="normal",explicit(normal(x,1),x,-2,3),
      grid=true)$
```



En la figura anterior, la recta normal no parece perpendicular a la recta tangente. Eso es por que no hemos tenido en cuenta la escala a la que se dibujan los ejes. Prueba a cambiar la escala para que queden perpendiculares.

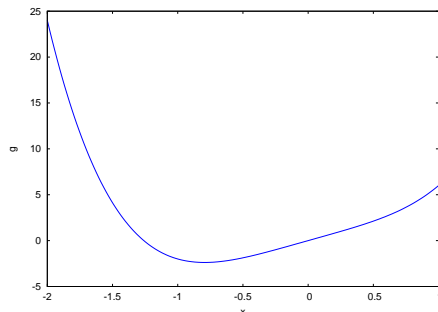
Prueba a cambiar la función  $f$ , el punto  $a$ . ¡Ahora es tu turno! Por ejemplo, haz esto mismo para una función que no sea derivable como  $f(x) = \sqrt{|x|}$  en el origen.

## 6.3 Máximos y mínimos relativos

En esta sección vamos a aprender a localizar extremos relativos de una función  $f$ . Para ello encontraremos las soluciones de la ecuación de punto crítico:  $f'(x) = 0$ . Y para resolver dicha ecuación podemos usar el comando `solve`.

**Ejemplo 6.3.** Calculemos los extremos relativos de la función  $f(x) = 2x^4 + 4x$ ,  $\forall x \in \mathbb{R}$ . Comenzamos, entonces, presentándosela al programa (no olvidéis borrar de la memoria la anterior función  $f$ ) y pintando su gráfica para hacernos una idea de dónde pueden estar sus extremos.

```
(%i29) f(x):=2*x^4+4*x$
(%i30) plot2d(f(x),[x,-4,4]);
(%o30)
```



Parece que hay un mínimo en las proximidades de -1. Para confirmarlo, calculamos los puntos críticos de  $f$ .

```
(%i31) diff(f(x),x)$
(%i32) d1f(x):='(%)$
(%i33) puntosf:solve(d1f(x),x);
(%o33) [x=(-sqrt(3)*%i-1)/(2*2^(1/3)),x=(sqrt(3)*%i+1)/(2*2^(1/3)),
x=-1/2^(1/3)]
```

Observamos que hay sólo una raíz real que es la única que nos interesa.

```
(%i34) last(puntosf);
(%o34) x=-1/2^(1/3)
```

Llamamos  $a$  al único punto crítico que hemos obtenido, y evaluamos en él la segunda derivada:

```
(%i35) a:rhs(%)$
(%i36) diff(f(x),x,2)$
(%i37) d2f(x):='(%)$
(%i38) d2f(a);
(%o38) 24/2^(2/3)
```

Por tanto la función  $f$  tiene un mínimo relativo en dicho punto  $a$ . ¿Puede haber otro extremo más?

**Ejemplo 6.4.** Vamos a calcular los extremos relativos de la función:

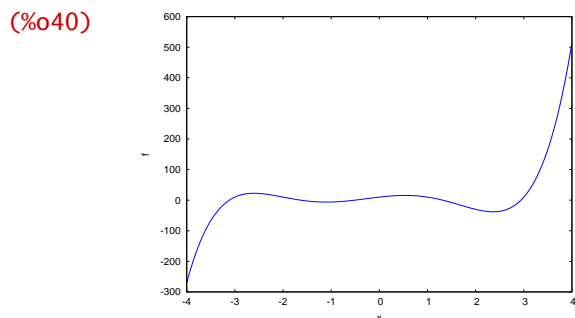
$$f(x) = x^5 + x^4 - 11x^3 - 9x^2 + 18x + 10 \text{ en el intervalo } [-4, 4]$$

Procedemos de la misma forma que en el ejemplo anterior.

```
(%i39) f(x):=x^5+x^4-11*x^3-9*x^2+18*x+10$
```



```
(%i40) plot2d(f(x), [x, -4, 4])$
```



A simple vista observamos que hay:

- un máximo relativo entre -3 y -2,
- un mínimo relativo entre -2 y -1,
- un máximo relativo entre 0 y 1, y
- un mínimo relativo entre 2 y 3.

Vamos entonces a derivar la función e intentamos calcular los ceros de  $f'$  haciendo uso del comando `solve`.

```
(%i41) define(d1f(x), diff(f(x), x));
```

```
(%o41) d1f(x) := 5x^4 + 4x^3 - 33x^2 - 18x + 18
```

```
(%i42) solve(d1f(x), x);
```

Las soluciones que nos da el programa no son nada manejables, así que vamos a resolver la ecuación  $f'(x) = 0$  de forma numérica, esto es, con el comando `find_root`. Para ello, nos apoyamos en la gráfica que hemos calculado más arriba, puesto que para resolver numéricamente esta ecuación hay que dar un intervalo en el que se puede encontrar la posible solución. Vamos, entonces, a obtener la lista de puntos críticos que tiene la función.

```
(%i43) puntosf: [find_root(d1f(x), x, -3, -2), find_root(d1f(x), x, -2, 1),
                 find_root(d1f(x), x, 0, 1), find_root(d1f(x), x, 2, 3)];
```

```
(%o43) [-2.600821117505113, -1.096856508567827, 0.53386135374308,
        2.363816272329865]
```

Ahora, para decidir si en estos puntos críticos se alcanza máximo o mínimo relativo, vamos a aplicar el test de la segunda derivada. Esto es:

```
(%i44) define(d2f(x), diff(f(x), x, 2));
```

```
(%i45) d2f(x) := 20x^3 + 12x^2 - 66x - 18
```

```
(%i46) map(d2f, puntosf);
```

```
(%o46) [-117.0277108731797, 42.43722588782084, -46.77165945967596,
        157.2021444450349]
```

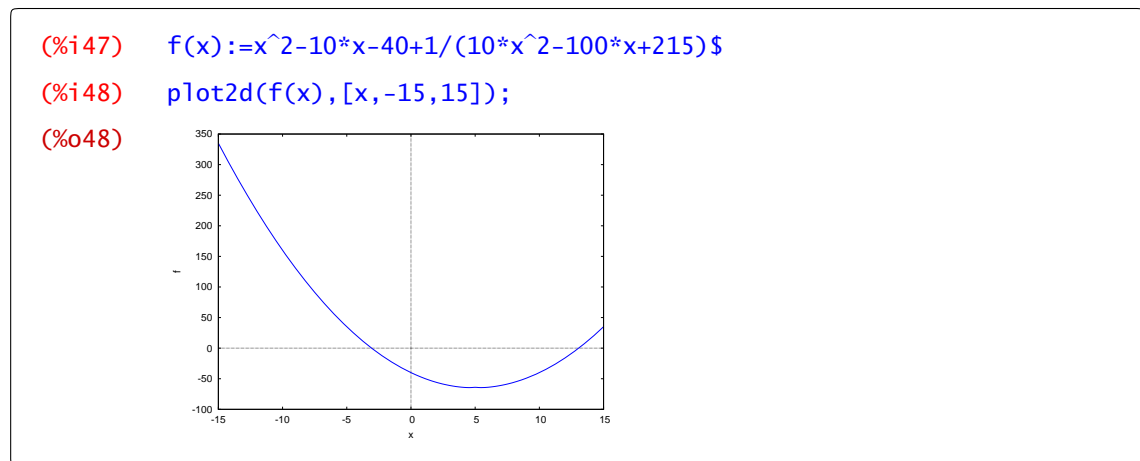
Lo que nos dice que en el primer y tercer puntos hay máximos relativos, mientras que en los otros dos tenemos mínimos relativos.

A continuación veremos un caso en el que la gráfica diseñada por el programa nos puede llevar a engaño. Descubriremos el error gracias al “test de la segunda derivada”.

**Ejemplo 6.5.** Vamos a estudiar los extremos relativos de la función

$$f(x) = x^2 - 10x - 40 + \frac{1}{10x^2 - 100x + 251}.$$

Comenzamos dibujando su gráfica en el intervalo  $[-15, 15]$

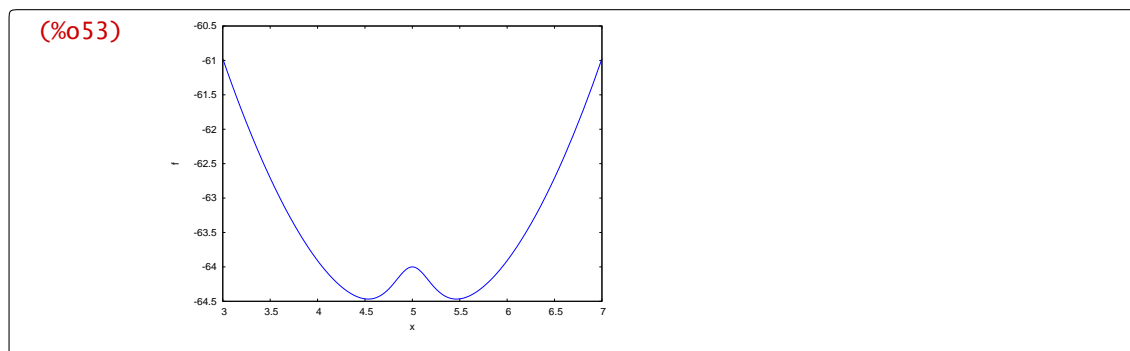


Aparentemente hay un mínimo cerca de 5. De hecho, si aplicamos el comando `solve` a la primera derivada de  $f$  obtenemos que precisamente la derivada de  $f$  se anula en  $x = 5$ . A la vista de la gráfica parece que efectivamente en  $x = 5$  hay un mínimo de la función  $f$ . Pero nada más lejos de la realidad, porque si evaluamos la derivada segunda de  $f$  en 5 obtendremos que  $f''(5) = -18 < 0$ . La segunda derivada es negativa y por tanto en  $x = 5$  tiene que haber un máximo. En efecto:

```
(%i49) diff(f(x),x)$
(%i50) '(%),x=5;
(%o50) 0
(%i51) diff(f(x),x,2)$
(%i52) '(%),x=5;
(%o52) -18
```

De hecho, si hubiéramos dibujado la gráfica en un intervalo más pequeño, sí se hubiera apreciado el máximo en el punto 5.

```
(%i53) plot2d(f(x),[x,3,7]);
```



En el siguiente ejemplo, resolvemos un problema de optimización.

**Ejemplo 6.6.** Se desea construir una ventana con forma de rectángulo coronado de un semicírculo de diámetro igual a la base del rectángulo. Pondremos cristal blanco en la parte rectangular y cristal de color en el semicírculo. Sabiendo que el cristal coloreado deja pasar la mitad de luz (por unidad de superficie) que el blanco, calcular las dimensiones para conseguir la máxima luminosidad si se ha de mantener el perímetro constante dado.

Llamemos  $x$  a la longitud de la base de la ventana y  $h$  a su altura. El perímetro es una cantidad dada  $A$ ; es decir,  $x + 2h + \frac{\pi x}{2} = A$ . Despejamos  $h$  en función de  $x$ :

(%i54) `altura:solve(x+2*h+(%pi*x)/2=A,h)`

(%o54) `[h=` $\frac{2A + (-\pi - 2)x}{4}$ `]`

La luminosidad viene dada por

$$f(x) = 2xh + \pi \frac{x^2}{8} = x(A - x - \pi \frac{x}{2}) + \pi \frac{x^2}{8} = Ax - \frac{1}{8}(8 + 3\pi)x^2$$

Definimos entonces la función  $f$  y calculamos sus puntos críticos:

(%i55) `f(x):=A*x-(1/8)*(8+3*%pi)*x^2`

(%i56) `solve(diff(f(x),x),x);`

(%o56) `[x=` $\frac{4A}{3\pi+8}$ `]`

(%i57) `%[1]`

(%o57) `x=` $\frac{4A}{3\pi+8}$

Y ahora evaluaremos la segunda derivada en dicho punto crítico. Para ello:

(%i58) `a:rhs(%)`

(%i59) `''(diff(f(x),x,2)),x=a;`

(%o59) `-` $\frac{3\pi+8}{4}$

La segunda derivada es negativa: ya tenemos el máximo que estábamos buscando.

Por último, veamos cómo podemos verificar una desigualdad haciendo uso de técnicas de derivación.

**Ejemplo 6.7.** Se nos plantea demostrar la siguiente desigualdad:  $\log(x + 1) \geq \frac{x}{x+1}$ ,  $\forall x \geq -1$ . Estudiemos entonces la siguiente función:  $f(x) = \log(x + 1) - \frac{x}{x+1}$ . Tendremos que comprobar que dicha función es siempre positiva. Para esto es suficiente comprobar que, si la función  $f$  tiene mínimo absoluto, su valor es mayor o igual que cero. En efecto:

```
(%i60) f(x):=log(x+1)-x/(x+1)$
(%i61) solve(diff(f(x),x),x);
(%o61) [x=0]
```

Por tanto sólo tiene un punto crítico, y además:

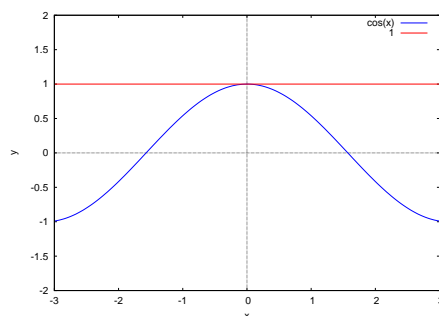
```
(%i62) ''(diff(f(x),x,2), x=0);
(%o62) 1
```

Luego, en el punto 0 alcanza un mínimo relativo que, por ser único, es el mínimo absoluto. Como se verifica que  $f(0) = 0$ , la desigualdad es cierta.

## 6.4 Polinomio de Taylor

En la sección dedicada a rectas secantes y tangentes hemos visto cómo la recta tangente a una función en un punto aproxima localmente a dicha función en ese punto. Es decir, que si sustituimos una función por su recta tangente en un punto, estamos cometiendo un error como se puede ver. En efecto, si dibujamos en una misma gráfica la función  $f(x) = \cos(x)$  y su recta tangente en cero, es decir  $t(x) = f(0) + f'(0)(x - 0) = 1$  obtenemos

```
(%i63) f(x):=cos(x);
(%o63) f(x):=cos(x)
(%i64) t(x):=1;
(%o64) t(x):=1
(%i65) plot2d([f(x),t(x)], [x,-3,3], [y,-2,2]);
(%o65)
```



En cuanto nos alejamos un poco del punto de tangencia (en este caso el 0), la función coseno y su tangente no se parecen en nada. La forma de mejorar la aproximación será aumentar el grado del polinomio que usamos, y la cuestión es, fijado un grado  $n$ , qué polinomio de grado menor o igual al fijado es el que más se parece a la función. El criterio con el que elegiremos el polinomio será hacer coincidir las sucesivas derivadas, esto es, el polinomio de Taylor de orden  $n$  de una función  $f$  en un punto  $a$ :

$$T(f, a, n)(x) = f(a) + f'(a)(x - a) + \frac{f''(a)}{2!}(x - a)^2 + \frac{f'''(a)}{3!}(x - a)^3 + \dots + \frac{f^{(n)}(a)}{n!}(x - a)^n = \sum_{k=1}^n \frac{f^{(k)}(a)}{k!}(x - a)^k$$

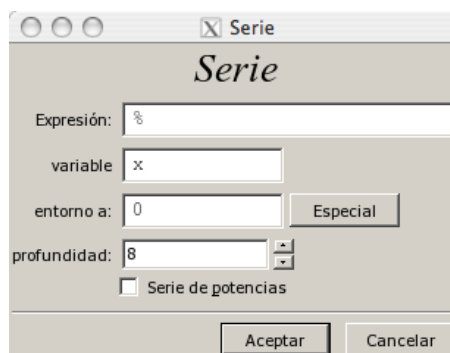


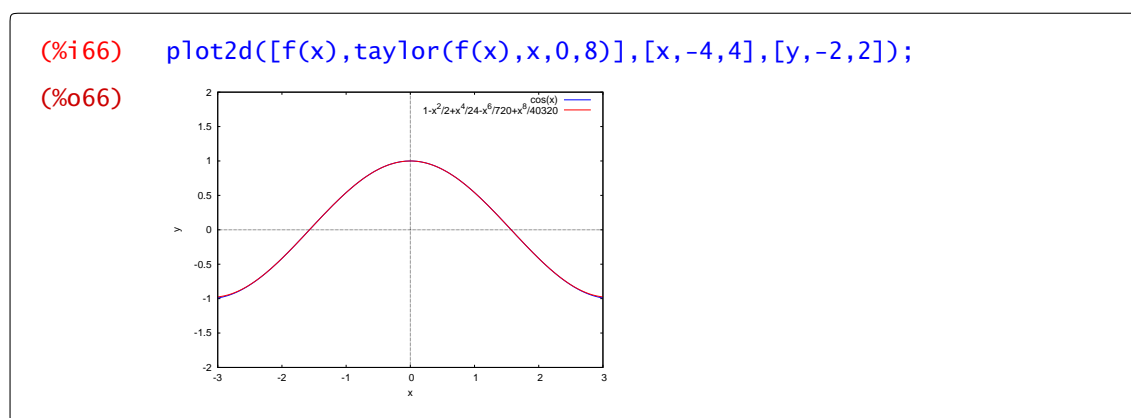
Figura 6.4 Polinomio de Taylor

El programa tiene una orden que permite calcular directamente el polinomio de Taylor centrado en un punto  $a$ . Se trata del comando `taylor`. En concreto, el comando `taylor(f(x), x, a, n)` nos da el polinomio de Taylor de la función  $f$  centrado en  $a$  y de grado  $n$ . Haciendo uso del menú podemos acceder al comando anterior desde **Análisis**–**Calcular serie**. Entonces se abre una ventana de diálogo en la que, escribiendo la expresión de la función, la variable, el punto en el que desarrollamos y el orden del polinomio de Taylor, obtenemos dicho polinomio. O bien, si marcamos la casilla de **Especial**, obtenemos el desarrollo de la función elegida en serie de potencias en torno al punto elegido.

taylor

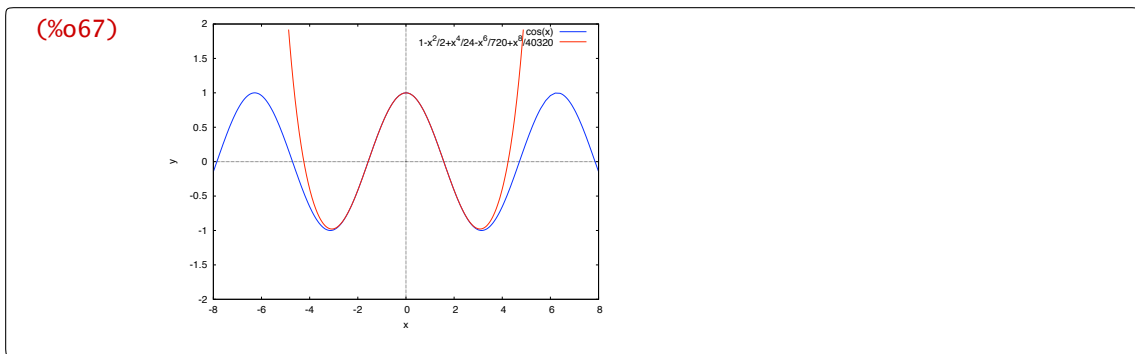
`taylor(f(x), x, a, n)` polinomio de Taylor de la función  $f$  en el punto  $a$  y de orden  $n$

Por ejemplo, vamos a dibujar las gráficas de la función  $f(x) = \cos(x)$  y de su polinomio de Taylor de orden 8 en el cero para comprobar que la aproximación ahora es más exacta.

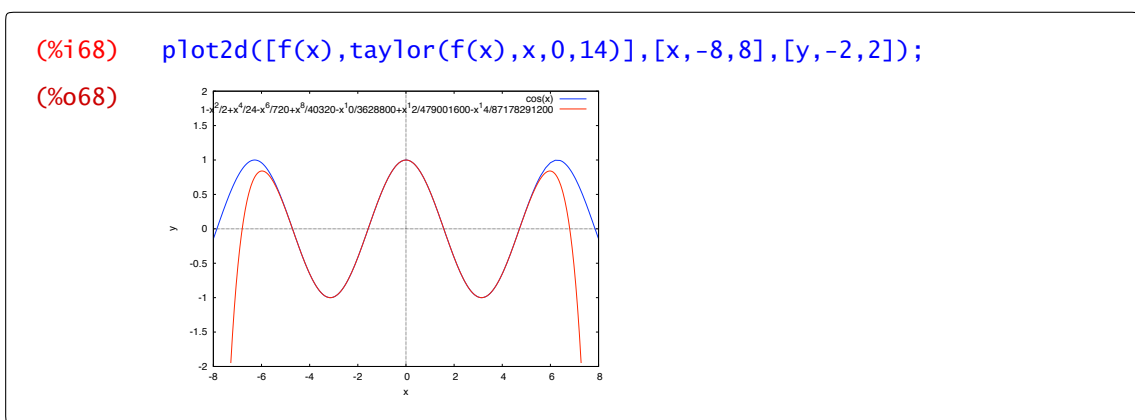


Pero si aumentamos el dominio podemos ver que el polinomio de Taylor se separa de la función cuando nos alejamos del origen.

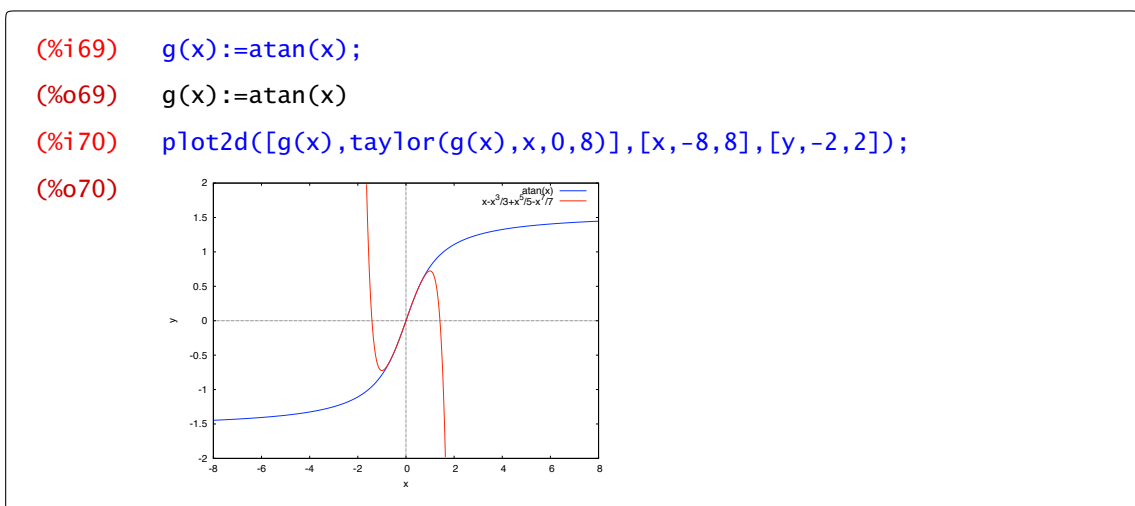
```
(%i67) plot2d([f(x), taylor(f(x), x, 0, 8)], [x, -8, 8], [y, -2, 2]);
```



Esto es lo esperable: la función coseno está acotada y el polinomio de Taylor, como todo polinomio no constante, no lo está. Eso sí, si aumentamos el grado del polinomio de Taylor vuelven a parecerse:



El hecho de que la función coseno y su polinomio de Taylor se parezcan tanto como se quiera, con sólo aumentar el grado del polinomio lo suficiente, no es algo que le ocurra a todas las funciones. Para la función arcotangente la situación no es tan buena:



sólo se parecen, al menos eso se ve en la gráfica, en el intervalo  $] - 1, 1[$  (a ojo).

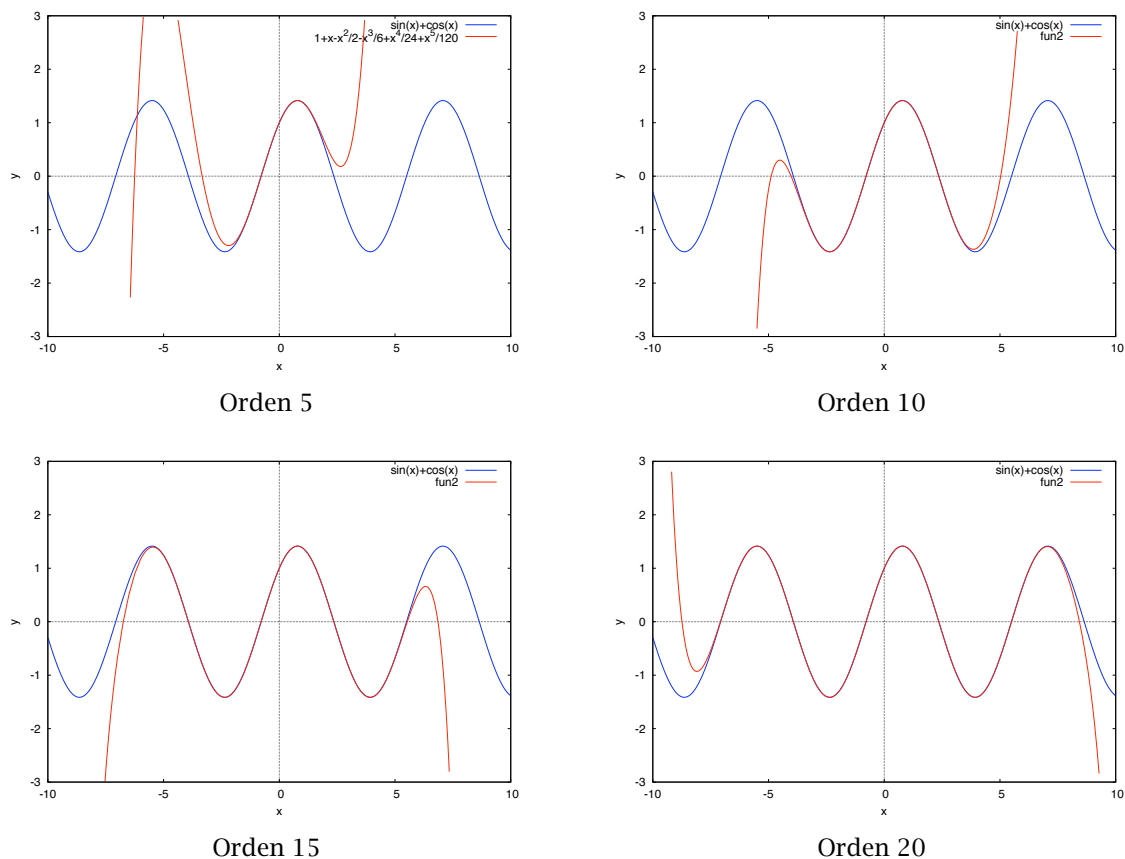
## Animaciones con polinomios

Hasta aquí hemos visto cómo comparar la gráfica de una función con la de su polinomio de Taylor. Ahora bien, en lugar de ir dibujando una función y un polinomio de Taylor, parece más interesante dibujar la función y varios polinomios para ir comprobando si se parecen o no a dicha función cuando aumenta su orden. La orden `with_slider` (que ya conoces) nos va a permitir hacer animaciones de la gráfica de una función y sus polinomios de Taylor. Por ejemplo:

```
(%i71) f(x):=sin(x)+cos(x);
(%o71) f(x):=sin(x)+cos(x)
(%i72) tay(n,x):=block([ts:taylor(f(z),z,0,n)],subst(z=x,ts));
(%o72) tay(n,x):=block([ts:taylor(f(z),z,0,n)],subst(z=x,ts))
(%i73) with_slider(n,range(1,20),[f(x),'(tay(n,x))],[x,-10,10],[y,-3,3]);
```

nos permite dibujar los primeros 20 polinomios de Taylor de la función  $f$ . En la Figura 6.5 tienes algunos pasos intermedios representados.

**Observación 6.8.** Hemos usado la orden `block` para definir una función intermedia que nos permita realizar la animación. No vamos a entrar en más detalles sobre cómo utilizarla en la definición de funciones. Puedes consultar la ayuda de *Maxima*, si tienes interés, donde encontrarás una explicación detallada de su uso.



**Figura 6.5** Función  $\sin(x) + \cos(x)$  y sus polinomios de Taylor

## 6.5 Ejercicios

### Ejercicio 6.1

- Calcula las tangentes a la hipérbola  $xy = 1$  que pasan por el punto  $(-1, 1)$ . Haz una representación gráfica.
- Calcula las tangentes a la gráfica de  $y = 2x^3 + 13x^2 + 5x + 9$  que pasan por el origen. Haz una representación gráfica.

### Ejercicio 6.2

Haz una animación gráfica en la que se represente la gráfica de la parábola  $f(x) = 6 - (x - 3)^2$  en azul, la recta tangente en el punto  $(1, f(1))$  en negro y las rectas secantes que pasan por los puntos  $(1, f(1))$  y  $(1 - h, f(1 - h))$  donde  $h$  varía de  $-3$  a  $-0.2$  con incrementos iguales a  $0.2$ , en rojo. Utiliza las opciones que consideres más apropiadas para hacer la animación.

### Ejercicio 6.3

Representa en una misma gráfica la parábola  $y = x^2/4$  y sus rectas tangentes en los puntos de abscisas  $-3 + 6k/30$  para  $1 \leq k \leq 29$ . Utiliza las opciones que consideres más apropiadas.

### Ejercicio 6.4

¿Es cierto o falso que el polinomio de Taylor de una función al cuadrado es el cuadrado del polinomio?

### Ejercicio 6.5

Halla dos números no negativos tales que la suma de uno más el doble del otro sea 12 y su producto sea máximo.

### Ejercicio 6.6

Un móvil tiene un movimiento rectilíneo con desplazamiento descrito por la función

$$s(t) = t^4 - 2t^3 - 12t^2 + 60t - 10.$$

¿En qué instante del intervalo  $[0, 3]$  alcanza su máxima velocidad?

### Ejercicio 6.7

Estudia los extremos relativos del polinomio de orden 5 centrado en el origen de la función  $f(x) = \cos(x) + e^x$ .



# Series numéricas y series de Taylor

## 7

### 7.1 Series numéricas

El estudio de la convergencia de una serie numérica no siempre es fácil y mucho menos el cálculo del valor de dicha suma. *Maxima* sabe calcular la suma de algunas series y siempre tenemos la posibilidad de aplicar algún criterio de convergencia.

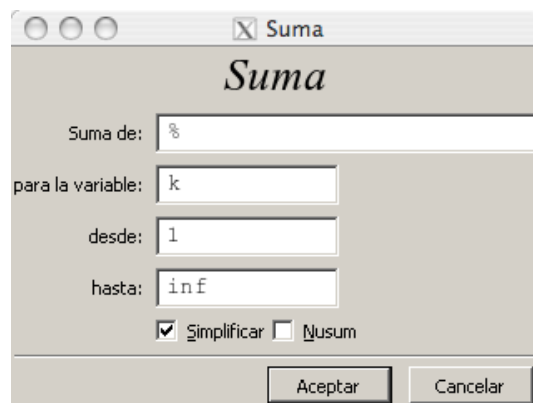
<code>sum(expr, n, a, b)</code>	$\sum_{n=a}^b expr$
<code>simplsum(suma)</code>	simplifica <i>suma</i>
<code>nusum (expr, x, a, plus)</code>	$\sum_{n=a}^b expr$

La orden genérica para calcular una suma, finita o infinita, es `sum`. Hace falta indicar qué sumamos, la variable que hace de índice y entre qué valores varía dicha variable. sum

```
(%i74) sum(i, i, 0, 100);
(%o74) 5050
```

Además hace falta indicarle que desarrolle y simplifique el sumatorio si queremos obtener un resultado más razonable. Esto se consigue con el operador `simplsum` que aparece escrito automáticamente si utilizamos el menú de *wxMaxima*.

```
(%i75) sum(i, i, 0, n);
(%o75)  $\sum_{i=0}^n i$ 
(%i76) sum(i, i, 0, n), simplsum;
(%o76)  $\frac{n^2+n}{2}$ 
```



simplsum

Figura 7.1 Calcular suma

La segunda posibilidad es usar la orden `nusum` que utiliza algunas reglas de sumación para obtener en algunos casos (series hipergeométricas principalmente) una expresión mejor del resultado. nusum

```
(%i77) nusum(i, i, 0, n);
(%o77)  $\frac{n(n+1)}{2}$ 
```

A cualquiera de estas dos órdenes se accede desde el menú **Análisis**→**Calcular suma**. Aparece la ventana de la Figura 7.1 que tienes que rellenar con la expresión a sumar, variable y extremos.

En la parte inferior de dicha ventana, puedes marcar **Simplificar** o **Nusum** para escoger entre las órdenes que hemos comentado.

El conjunto de series que sabe sumar *Maxima* no es demasiado grande, ya hemos dicho que es un problema difícil, pero tiene respuesta para algunas series clásicas. Por ejemplo,

```
(%i78) sum(1/n^2,n,1,inf),simpsum;
(%o78)  %pi^2
        6
```

o la serie armónica

```
(%i79) sum(1/n,n,1,inf),simpsum;
(%o79)  ∞
```

que sabemos que no es convergente. Si intentamos sumar una progresión geométrica cualquiera nos pregunta por la razón y la suma sin mayores problemas

```
(%i80) sum(x^n,n,1,inf),simpsum;
Is |x|-1 positive, negative or zero? negative;
(%o80)  x
        1-x
```

En cambio no suma algunas otras como

```
(%i81) sum(1/n!,n,0,inf),simpsum;
(%o81)  ∑ 1/n!
        n=0
```

Como hemos dicho, calcular la suma de una serie es un problema difícil y lo que hemos hecho en clase es estudiar cuándo son convergentes sin calcular su suma. Aplicar el criterio de la raíz o del cociente para decidir la convergencia de una serie, reduce el problema al cálculo de un límite, que en el caso anterior es trivial, en el que *Maxima* sí puede ser útil. Por ejemplo, la serie  $\sum_{n \geq 1} \frac{1}{n^n}$

```
(%i82) a[n]:=1/n^n;
(%o82)  a_n := 1/n^n
(%i83) sum(a[n],n,1,inf),simpsum;
(%o83)  ∑ 1/n^n
        n=1
```

no sabemos sumarla, pero el criterio del cociente

```
(%i84) limit(a[n+1]/a[n],n,inf);
```

```
(%o84) 0
```

nos dice que es convergente.

## 7.2 Desarrollo de Taylor

El desarrollo en serie de Taylor de una función parte de una idea sencilla: si el polinomio de Taylor de la función  $\cos(x)$  se parece cada vez más a la función, ¿por qué parar?

```
(%i85) taylor(cos(x),x,0,4);
```

(%o85)  $1 - \frac{x^2}{2} + \frac{x^4}{24} + \dots$

¿Qué es eso de un polinomio con infinitos sumandos? Para eso acabamos de ver series de números. Ya sabemos en qué sentido podemos hablar de sumas infinitas.

```
niceindices(powerseries(f(x), x, a))
```

desarrollo en serie de potencias de la función  $f$  centrado en el punto  $a$

Si en el menú **Análisis**→**Calcular serie**, marcamos **Serie de potencias** obtenemos el desarrollo en serie de potencias de la función

```
(%i86) niceindices(powerseries(cos(x), x, 0));
```

(%o86)  $\sum_{i=0}^{\infty} \frac{(-1)^i x^{2i}}{(2i)!}$

La orden `powerseries` es la encargada del cálculo de dicho desarrollo y el comando que se antepone, `niceindices`, escribe la serie de Taylor de una función con índices más apropiados. Si no la utilizamos, simplemente el resultado aparece un poco más “enmarañado”:

```
(%i87) powerseries(cos(x),x,0);
```

(%o87)  $\sum_{i16=0}^{\infty} \frac{(-1)^{i16} x^{2i16}}{(2i16)!}$

pero es esencialmente el mismo. Funciona bastante bien para las funciones con las que hemos trabajado en clase:

```
(%i88) niceindices(powerseries(atan(x), x, 0));
```

(%o88)  $\sum_{i=0}^{\infty} \frac{(-1)^i x^{2i+1}}{2i+1}$

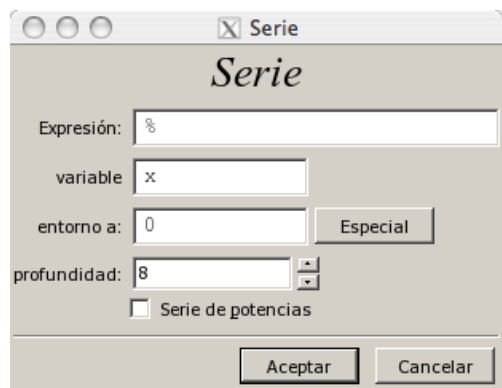


Figura 7.2 Calcular serie de Taylor

`powerseries`  
`niceindices`

Ahora podemos dar una respuesta al problema de en dónde se parecen una función y su polinomio: la primera condición es que la serie de Taylor tiene que ser convergente. El conjunto de valores  $x$  en los que es convergente, hemos visto en clase que, depende del radio de convergencia de la serie. Vamos a calcularlo:

```
(%i89) a[i]:=(-1)^i/(2*i+1);
(%o89) a_i := \frac{(-1)^i}{2i+1}
(%i90) limit(abs(a[i+1]/a[i]),i,inf);
(%o90) 1
```

Perfecto. Ya sabemos que la serie de Taylor de la función arcotangente converge para  $x \in ]-1, 1[$  y no es convergente en  $\mathbb{R} \setminus ]-1, 1[$  y el dibujo que vimos de su polinomio de Taylor tiene ahora sentido. Queda por saber qué ocurre para  $x = \pm 1$ . Eso te toca a tí.

**Observación 7.1.** Se pueden realizar algunas operaciones con series de potencias. Por ejemplo, intenta derivarlas:

```
(%i91) f(x):=niceindices(powerseries(log(1+x^2),x,0))$
(%i92) diff(f(x),x);
(%o92) -2 \sum_{i=1}^{\infty} (-1)^i x^{2i-1}
```

## 7.3 Ejercicios

### Ejercicio 7.1

Estudiar la convergencia de las series que tienen el siguiente término general.

- a)  $a_n = \frac{1}{2^n - n}$ ,  
 b)  $a_n = \frac{n!}{n^n}$ ,  
 c)  $a_n = \frac{1}{\ln(n)^2}$ ,  
 d)  $a_n = \frac{n^2 - n + 3}{3^n + 2^{-n} + n - 1}$ ,  
 e)  $a_n = \ln\left(\frac{n^3 + 1}{n^3 + 2n^2 - 3}\right)$ .

### Ejercicio 7.2

Se deja caer una pelota elástica desde una altura  $h$ . Cada vez que toca el suelo bota hasta una altura que es el 75% de la altura desde la que cayó. Hállese  $h$  sabiendo que la pelota recorre una distancia total de 21 m. hasta detenerse.

### Ejercicio 7.3

Calcula el desarrollo en serie de potencias centrado en  $a$  de las siguientes funciones, realiza una animación de la función y los primeros 20 polinomios de Taylor y, por último, estudia la convergencia de la serie de Taylor.

a)  $f(x) = \cos(x^2), a = 0,$

b)  $f(x) = \sin(x)^2, a = 0,$

c)  $f(x) = \log(1 + x^2), a = 0,$

d)  $f(x) = \log(5 + x^2), a = 0,$

e)  $f(x) = \log(5 + x^2), a = 0,$

f)  $f(x) = x^4 + 3x^2 - 2x + 3, a = 1,$

g)  $f(x) = |x|, a = 1,$

h)  $f(x) = |x|, a = 5.$



# Integración

## 8

En este capítulo vamos a aprender a calcular integrales en una variable, así como aplicar estas integrales al cálculo de áreas de recintos en el plano limitados por varias curvas, longitudes de curvas, volúmenes de cuerpos de revolución y áreas de superficies de revolución.

### 8.1 Cálculo de integrales

La principal orden de *Maxima* para calcular integrales es `integrate`. Nos va a permitir calcular integrales, tanto definidas como indefinidas, con mucha comodidad.

<code>integrate(f(x), x)</code>	primitiva de la función $f(x)$
<code>integrate(f(x), x, a, b)</code>	$\int_a^b f(x) dx$

Disponemos también de la siguiente opción en el menú para calcular integrales, sin necesidad de escribir el comando correspondiente en la ventana de entradas: **Análisis**→**Integrar**. Después sólo tenemos que rellenar los datos que nos interesen.



Figura 8.1 Cálculo de integrales

Comencemos por integrales indefinidas. *Maxima* calcula primitivas de funciones trigonométricas,

```
(%i1) integrate(x*sin(x), x);
(%o1) sin(x)-x*cos(x)
```

de funciones racionales,

```
(%i2) integrate(1/(x^4-1), x);
```

$$(\%02) \quad -\frac{\log(x+1)}{4} - \frac{\operatorname{atan}(x)}{2} + \frac{\log(x-1)}{4}$$

irracionales,

(%i3) `integrate(sqrt(x^2+1),x);`

$$(\%03) \quad \frac{\operatorname{asinh}(x)}{2} + \frac{x\sqrt{x^2+1}}{2}$$

ya sea aplicando integración por partes

(%i4) `integrate(x^3*%e^x,x);`

$$(\%04) \quad (x^3 - 3x^2 + 6x - 6) \%e^x$$

o el método que considere necesario

(%i5) `integrate(%e^(-x^2),x);`

$$(\%05) \quad \frac{\sqrt{\%pi} \operatorname{erf}(x)}{2}$$

integral que no sabíamos hacer. Bueno, como puedes ver, *Maxima* se defiende bien con integrales. Eso sí, es posible que nos aparezcan funciones (como erf). El motivo es muy sencillo: la forma de saber calcular primitivas de muchas funciones es saberse muchas funciones. *Maxima* se sabe muchas y, cuando nos aparezca alguna nueva, siempre podemos preguntar cuál es.

(%i6) `describe(erf);` -- Función: erf (<x>)

Es la función de error, cuya derivada es '2\*exp(-x^2)/sqrt(%pi)'.

There are also some inexact matches for 'erf'.

Try '?? erf' to see them.

(%o6) true

Para calcular integrales definidas sólo tenemos que añadir los extremos del intervalo de integración

(%i7) `integrate(x*sin(x),x,%pi/2,%pi);`

(%o7)  $\pi - 1$

(%i8) `integrate(1/(x^2-1),x,2,5);`

$$(\%08) \quad -\frac{\log(6)}{2} + \frac{\log(4)}{2} + \frac{\log(3)}{2}$$

En la integrales pueden aparecer parámetros y si *Maxima* tiene "dudas" acerca de su valor pregunta,



```
(%i9) integrate(x^n,x);
Is n+1 zero or nonzero? nonzero;
(%o9)  $\frac{x^{n+1}}{n+1}$ 
```

pregunta todo lo que haga falta:

```
(%i10) integrate(1/x,x,a,b);
Is b - a positive, negative, or zero? positive;
Is b positive, negative, or zero? positive;
Is a positive, negative, or zero? positive;
Is x + b positive or negative? positive;
(%o10)  $\log(b) - \log(a)$ 
```

Evidentemente hay funciones a las que *Maxima* no sabe calcular una integral indefinida. En ese caso, da como respuesta la misma integral que le hemos preguntado.

```
(%i11) integrate(exp(x^3+x),x);
(%o11)  $\int e^{x^3+x} dx$ 
```

Veremos que este problema desaparece cuando pasamos a integrales definidas.

### 8.1.1 Integración impropia

Como recordarás de las clases teóricas, la integral que, en principio, se define para funciones continuas en intervalos compactos, puede extenderse a funciones continuas definidas en intervalos de longitud infinita y a funciones que no están acotadas en un intervalo de longitud finita. Es lo que se conoce como *integración impropia*. Para el programa *Maxima*, trabajar con integrales impropias no supone ningún problema ya que las trata exactamente igual que las integrales de funciones continuas en intervalos cerrados y acotados. De hecho, no hay forma de distinguir unas de otras: en ningún momento hemos indicado al comando `integrate` que la función era acotada ni hemos dicho si el intervalo era abierto, cerrado o semiabierto. Sólo hemos indicado los extremos y ya está. Por tanto, con la misma orden `integrate` se pueden calcular integrales impropias. Por ejemplo:

```
(%i12) integrate(1/sqrt(1-x^2),x,-1,1);
(%o12)  $\pi$ 
(%i13) integrate(%e(-x^2),x,0,inf);
(%o13)  $\frac{\sqrt{\pi}}{2}$ 
```

Intenta calcular una primitiva de los integrandos anteriores. Como verás, la primitiva de la función  $f(x) = \frac{1}{\sqrt{1-x^2}}$  ya la conocías; en cambio, en la primitiva de la función  $f(x) = \exp(-x^2)$  aparece una función que seguro que no conocías, pero que ha aparecido en el apartado dedicado al cálculo de integrales. Probamos a hacer las siguientes integrales impropias:

$$\int_1^{+\infty} \frac{1}{x^2} dx, \int_1^{+\infty} \frac{1}{x} dx$$

```
(%i14) integrate(1/x^2,x,1,inf);
(%o14) 1
(%i15) integrate(1/x,x,0,inf);
Integral is divergent
-- an error. To debug this try debugmode(true);
```

Observa cómo en la salida de la última integral el programa advierte de la “no convergencia” de la integral planteada. ¿Sabéis decir el por qué? Bueno, recordemos que la definición de dicha integral es

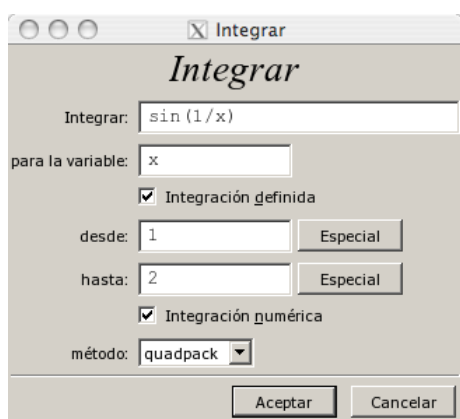
$$\int_1^{+\infty} \frac{1}{x} dx = \lim_{x \rightarrow +\infty} G(x) - \lim_{x \rightarrow 1} G(x),$$

donde  $G$  es una primitiva de  $\frac{1}{x}$ . En este caso, una primitiva es  $\ln(x)$  que no tiene límite en  $+\infty$  (es  $+\infty$ ).

### 8.1.2 Integración numérica

El cálculo de primitivas no es interesante por sí mismo, el motivo que lo hace interesante es la regla de Barrow: si conoces una primitiva de una función, entonces el cálculo de la integral en un intervalo se reduce a evaluar la primitiva en los extremos y restar.

<code>quad_quags(f(x), x, a, b)</code>	aproximación numérica de $\int_a^b f(x) dx$
<code>quad_qagi(f(x), x, a, b)</code>	aproximación numérica de $\int_a^b f(x) dx$
<code>romberg(f(x), x, a, b)</code>	aproximación numérica de $\int_a^b f(x) dx$



En la práctica, no siempre es fácil calcular una primitiva, pero sí es fácil (para un ordenador y teóricamente) aproximar el valor de la integral por las áreas de los rectángulos que aparecen en la definición de integral. Este método no es el mejor, pero versiones mejoradas permiten aproximar el valor de casi cualquier integral.

Si en el menú **Análisis** → **Integrar** marcamos “Integración definida” e “Integración numérica” se nos da la opción de escoger entre dos métodos: `quadpack` y `romberg`, cada uno referido a un tipo diferente de aproximación numérica de la integral. Vamos a calcular numéricamente la integral que hemos hecho más arriba que tenía como resultado  $\pi - 1$

```
(%i16) quad_qags(x*sin(x),x,%pi/2,%pi);
(%o16) [2.141592653589794,2.3788064330872769 10-14,21,0]
(%i17) romberg(x*sin(x),x,%pi/2,%pi);
(%o17) 2.141591640806944
```

Observamos que hay una diferencia entre las salidas de ambos comandos. Mientras que en la última aparece el valor aproximado de la integral (fíjate que es  $\pi - 1$ ), en la primera aparece como salida una lista de 4 valores:

- la aproximación de la integral
- el error absoluto estimado de la aproximación
- el número de evaluaciones del integrando
- el código de error (que puede ir desde 0 hasta 6) que, copiado de la ayuda de *Maxima*, significa

El código de error (el cuarto elemento del resultado) puede tener los siguientes valores:

- '0' si no ha habido problemas;
- '1' si se utilizaron demasiados intervalos;
- '2' si se encontró un número excesivo de errores de redondeo;
- '3' si el integrando ha tenido un comportamiento extraño frente a la integración;
- '4' fallo de convergencia;
- '5' la integral es probablemente divergente o de convergencia lenta;
- '6' si los argumentos de entrada no son válidos.

Qué duda cabe que este tipo de integración numérica es interesante para integrandos de los cuales no se conoce una primitiva. Vamos a intentar calcular una primitiva y posteriormente una integral definida de la función  $f(x) = e^{-x^2}$ .

```
(%i18) integrate(exp(-x^2),x);
(%o18)  $\frac{\sqrt{\pi} \operatorname{erf}(x)}{2}$ 
(%i19) integrate(exp(-x^2),x,0,1);
(%o19)  $\frac{\sqrt{\pi} \operatorname{erf}(1)}{2}$ 
```

Si ahora queremos, podemos calcular numéricamente este valor,

```
(%i20) float(%);
```

```
(%o20) 0.74682413281243
```

o también, podíamos haber utilizado los comandos de integración numérica directamente

```
(%i21) quad_qags(exp(-x^2),x,0,1);
(%o21) [0.74682413281243,8.2954620176770253 10-15,21,0]
(%i22) romberg(exp(-x^2),x,0,1);
(%o22) 0.7468241699099
```

#### quadpack

En realidad, quadpack no es un método concreto sino una serie de métodos para aproximar numéricamente la integral. Los dos que hemos visto antes, quad\_qags y romberg, se pueden utilizar en intervalos finitos. Comprueba tú mismo lo que ocurre cuando calculas  $\int_1^{\infty} \cos(x)/x^2 dx$  utilizando el menú **Análisis**→**Integrar**: obtendrás algo así

```
(%i23) quad_qagi(cos(x)/x^2, x, 1, inf);
***MESSAGE FROM ROUTINE DQAGI IN LIBRARY SLATEC.
***INFORMATIVE MESSAGE, PROG CONTINUES, TRACEBACK REQUESTED
* ABNORMAL RETURN
* ERROR NUMBER = 1
*
***END OF MESSAGE
(%o23) [-0.084302101159614,1.5565565173632223 10-4,5985,1]
```

*Maxima* decide cuál es el método que mejor le parece y, en este caso utiliza quad\_qagi. Evidentemente, depende del integrando el resultado puede dar uno u otro tipo de error. Por ejemplo, vamos ahora a calcular de forma numérica  $\int_0^1 \frac{\sin(1/x)}{x}$ .

```
(%i24) quad_qags((1/x)*sin(1/x),x,0,1);
***MESSAGE FROM ROUTINE DQAGS IN LIBRARY SLATEC.
***INFORMATIVE MESSAGE, PROG CONTINUES, TRACEBACK REQUESTED
* ABNORMAL RETURN
* ERROR NUMBER = 5
*
***END OF MESSAGE
(%o24) [-1.050233246377689,0.20398634967385,8379,5]
(%i25) romberg((1/x)*sin(1/x),x,0,1);
(%o25) romberg( $\frac{\sin(\frac{1}{x})}{x}$ ,x,0.0,1.0)
```

Se trata de una función que cerca del cero oscila mucho, lo que hace que el comando quad-qags nos dé aviso de error (aunque da una aproximación), mientras que el comando romberg no nos dé ninguna salida.

## 8.2 Sumas de Riemann

El proceso que hemos seguido en clase para la construcción de la integral de una función  $f : [a, b] \rightarrow \mathbb{R}$  pasa por lo que hemos llamado sumas superiores y sumas inferiores. Lo que hacíamos era dividir el intervalo  $[a, b]$  en  $n$  trozos usando una partición  $P = \{a = x_0 < x_1 < \dots < x_n = b\}$  del intervalo. Luego calculábamos el supremo y el ínfimo de  $f$  en cada uno de los intervalos  $I_i = [x_{i-1}, x_i]$  y podíamos aproximar la integral de dos formas: si escogíamos el área asociada a los rectángulos que tienen como altura el supremo obteníamos las sumas superiores. Las sumas inferiores se obtenían de forma análoga pero con el ínfimo.

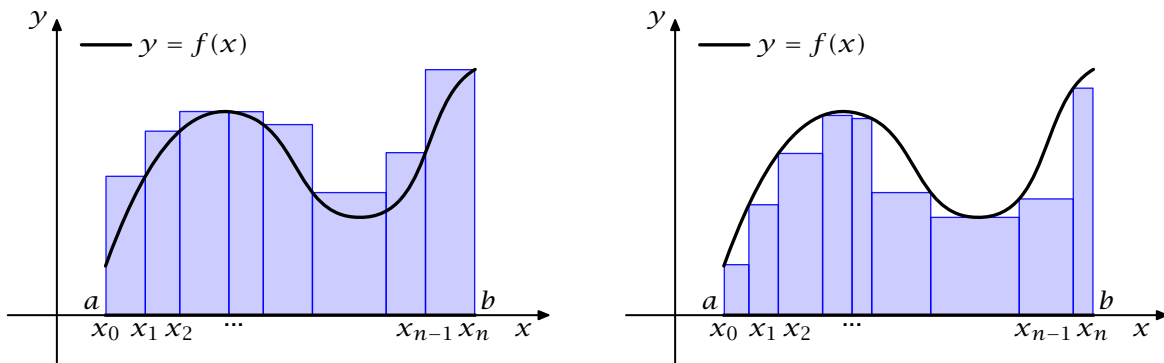


Figura 8.2 Sumas superiores e inferiores

Las dos áreas que tienes marcadas en la Figura 8.2 son

$$S(f, P) = \sum_{i=1}^n \sup (f(I_i)) \cdot \ell(I_i) \text{ y } I(f, P) = \sum_{i=1}^n \inf (f(I_i)) \cdot \ell(I_i),$$

donde  $\ell(I_i)$  denota la longitud del intervalo  $I_i$  o, lo que es lo mismo, la base de cada uno de los rectángulos que tenemos en la figura.

Si coinciden las mejores aproximaciones obtenidas de esta manera o hablando un poco a la ligera, si la menor de las sumas superiores coincide con la mayor de las sumas inferiores entonces llamamos a ese valor integral de la función.

Este método tiene varios inconvenientes si pretendemos hacerlo con un ordenador. Por ejemplo, ¿cómo escogemos la partición? y, más importante, ¿cómo calculamos el supremo y el ínfimo de la función en cada uno de los intervalos? El primer problema se puede evitar tomando la partición de una forma particular: dividimos el intervalo  $[a, b]$  en trozos iguales, en muchos trozos iguales. Esto nos permite dar una expresión concreta de la partición.

Si dividimos  $[a, b]$  en dos trozos, estarás de acuerdo que los puntos serían  $a, b$  y el punto medio  $\frac{a+b}{2}$ . ¿Cuáles serían los puntos que nos dividen  $[a, b]$  en tres trozos? Ahora no hay punto medio, ahora nos hacen falta dos puntos además de  $a$  y  $b$ . Piénsalo un poco. En lugar de fijarnos en cuáles son los extremos de los intervalos, fijémonos en cuánto miden. Efectivamente, todos miden lo mismo  $\frac{b-a}{3}$  puesto que queremos dividir el intervalo en tres trozos iguales. Ahora la pregunta es: ¿qué intervalo tiene como extremos de la izquierda  $a$  y mide  $\frac{b-a}{3}$ ? La respuesta es muy fácil:  $[a, a + \frac{b-a}{3}]$ . Con este método es sencillo seguir escribiendo intervalos: sólo tenemos que seguir sumando  $\frac{b-a}{3}$  hasta llegar al extremo de la derecha,  $b$ .

En general, si  $n$  es un natural mayor o igual que 2,

$$a, a + \frac{b-a}{n}, a + 2 \cdot \frac{b-a}{n}, \dots, a + n \cdot \frac{b-a}{n}$$

son los  $n + 1$  extremos que dividen a  $[a, b]$  en  $n$  intervalos iguales. Vamos a seguir estos pasos con *Maxima*. Escojamos una función y un intervalo:

```
(%i26) a:-2;b:5;n:10
(%o27) -2
(%o28) 5
(%o28) 10
(%i29) f(x):=(x-1)*x*(x-3);
(%o29) f(x):=(x-1)x(x-3)
```

La lista de extremos es fácil de escribir usando el comando `makelist`.

```
(%i30) makelist(a+i*(b-a)/n,i,0,n);
(%o30) [-2,-13/10,-3/5,1/10,4/5,3/2,11/5,29/10,18/5,43/10,5]
```

La segunda dificultad que hemos mencionado es la dificultad de calcular supremos e ínfimos. La solución es escoger un punto cualquiera de cada uno de los intervalos  $I_i$ : el extremo de la izquierda, el de la derecha, el punto medio o, simplemente, un punto elegido al azar. Esto tiene nombre: una suma de Riemann. Si dividimos el intervalo en muchos trocitos, no debería haber demasiada diferencia entre el supremo, el ínfimo o un punto intermedio.

Convergencia de las sumas de Riemann

**Proposición 8.1.** Sea  $f : [a, b] \rightarrow \mathbb{R}$  integrable. Sea  $P_n$  la partición

$$a, a + \frac{b-a}{n}, a + 2 \cdot \frac{b-a}{n}, \dots, a + n \cdot \frac{b-a}{n}$$

y sea  $x_i \in \left[ a + (i-1) \frac{b-a}{n}, a + i \frac{b-a}{n} \right]$ . Entonces

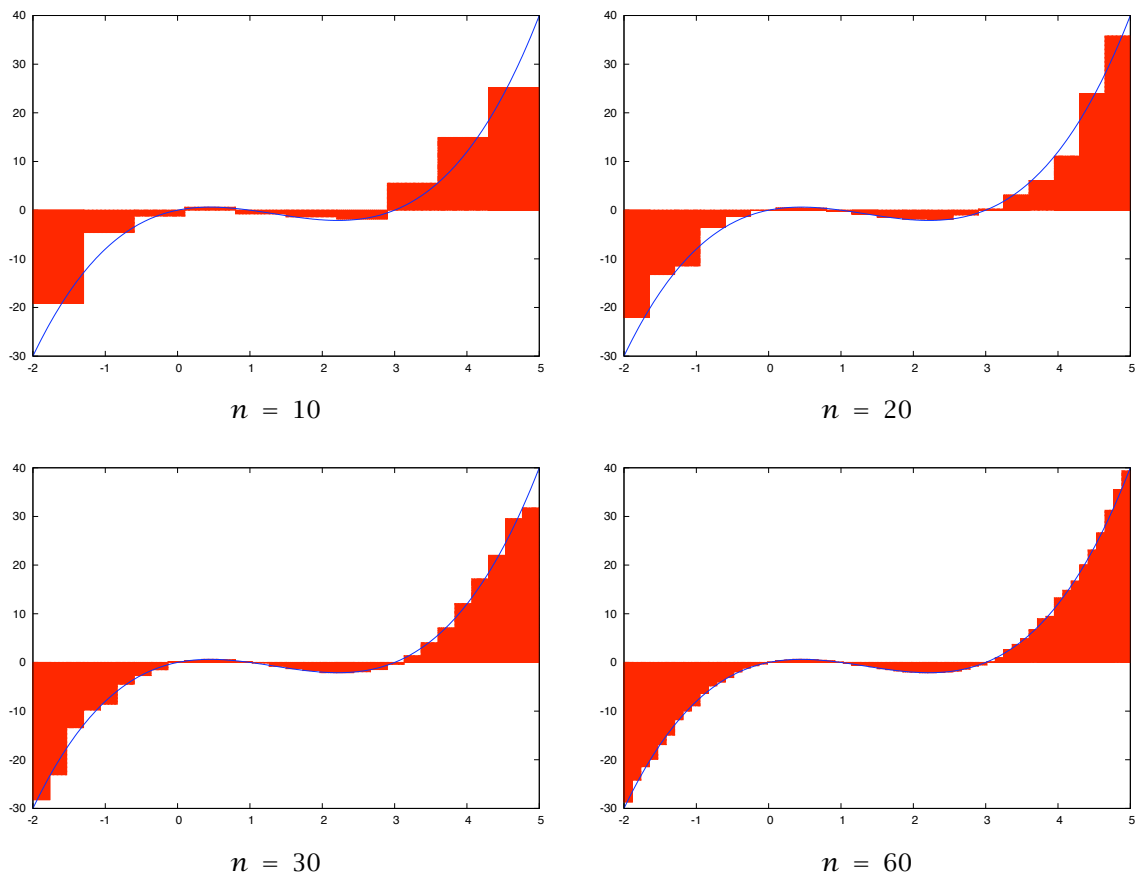
$$\lim_{n \rightarrow \infty} \sum_{i=1}^n f(x_i) \frac{b-a}{n} = \int_a^b f(x) dx.$$

En la Figura 8.3 podemos ver cómo mejora cuando aumentamos el número de subdivisiones. Buenos, sigamos adelante. Elegimos puntos al azar en cada uno de los intervalos.

```
(%i31) puntos:makelist(a+(i-1)*(b-a)/n+random(1.0)*(b-a)/n,i,1,n);
[-1.974524678870055,-0.86229173716074,0.095612020508637,
0.12911247230811,1.242837772556621,1.915206527061883,
(%o31) 2.658260436280155,3.229497471823565,3.636346487565163,
4.30162951626272]
```

Ahora tenemos que sumar las áreas de los rectángulos de las sumas de Riemann, cosa que podemos hacer de la siguiente manera usando la orden `sum`:

```
(%i32) sum((b-a)/n*(f(a+(i-1)*(b-a)/n+random(1.0)*(b-a)/n)),i,1,n);
(%o32) 5.859818716400653
```



**Figura 8.3** Convergencia de las sumas de Riemann

que si lo comparamos con el valor de dicha integral

```
(%i33) integrate(f(x),x,a,b);
(%o33)  77
        12
(%i34) %,numer
(%o34)  6.416666666666667
```

...pues no se parece demasiado. Bueno, aumentemos el número de intervalos:

```
(%i35) n:100;
(%o35)  100
(%i36) sum((b-a)/n*(f(a+(i-1)*(b-a)/n+random(1.0)*(b-a)/n)),i,1,n);
(%o36)  6.538846969978081
```

Vale, esto es otra cosa. Ten en cuenta que, debido al uso de `random`, cada vez que ejecutes la orden obtendrás un resultado diferente y, por supuesto, que dependiendo de la función puede ser necesario dividir en una cantidad alta de intervalos.

## 8.3 Aplicaciones

### 8.3.1 Cálculo de áreas planas

Te recuerdo que si  $f$  y  $g$  son funciones integrables definidas en un intervalo  $[a, b]$ , el área entre las dos funciones es

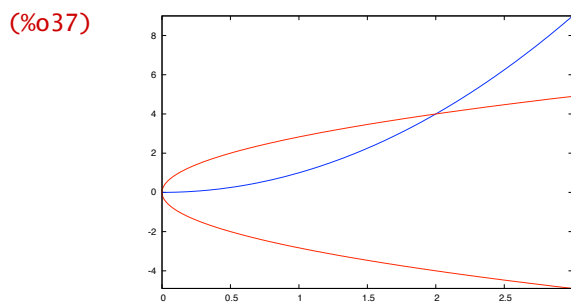
$$\int_a^b |f(x) - g(x)| dx$$

Evidentemente, calcular una integral con un valor absoluto no es fácil (¿cuál es la primitiva de la función  $|x|$ ?). Como función definida a trozos, lo que tenemos que hacer es dividir el intervalo  $[a, b]$  en subintervalos en los que sepamos cuál de las dos funciones  $f$  y  $g$  es la más grande.

**Ejemplo 8.2.** Calculemos el área entre las curvas  $y = x^2$ ,  $y^2 = 8x$ .

Podemos dibujar las dos curvas y hacernos una idea del aspecto del área que queremos calcular.

```
(%i37) draw2d(
color=blue,
explicit(x^2,x,0,3),
color=red,
explicit(sqrt(8*x),x,0,3),
explicit(-sqrt(8*x),x,0,3)
)
```



En realidad, lo primero que nos hace falta averiguar son los puntos de corte:

```
(%i38) solve([y=x^2,y^2=8*x],[x,y]);
(%o38) [[x=2,y=4],[x=-sqrt(3)*i-1,y=2*sqrt(3)*i-2],[x=sqrt(3)*i-1,y=-2*sqrt(3)*i-2],
[x=0,y=0]]
```

De todas las soluciones, nos quedamos con las soluciones reales:  $(0,0)$  y  $(2,4)$ . En el intervalo  $[0,2]$ , ¿cuál de las dos funciones es mayor? Son dos funciones continuas en un intervalo que sólo coinciden en 0 y en 2, por tanto el Teorema de los ceros de Bolzano nos dice que una de ellas es siempre mayor que la otra. Para averiguarlo sólo tenemos que evaluar en algún punto entre 0 y 2. En este caso es más fácil: se ve claramente que  $y = x^2$  está por debajo. Por tanto el área es

```
(%i39) integrate(sqrt(8*x)-x^2,x,0,2);
```



$$(\%o39) \quad \frac{8}{3}$$

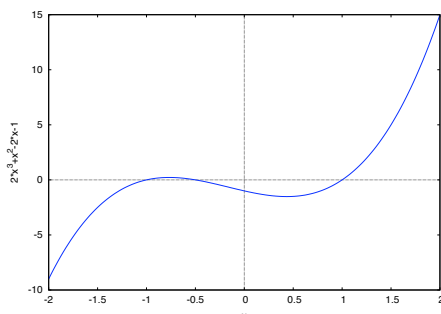
**Observación 8.3.** La fórmula que define el cálculo del área entre dos funciones tiene un valor absoluto que hace difícil calcular una primitiva directamente. Ese es el motivo por el que dividimos el intervalo en trozos: quitar ese valor absoluto. Si sólo pretendemos calcular el valor numéricamente, entonces el valor absoluto no es un impedimento y podemos calcular directamente la integral olvidándonos de puntos de corte o de qué función es mayor:

```
(%i40) quad_qags(abs(x^2-sqrt(8*x)),x,0,2);
(%o40) [2.6666666666666668,2.9605947323337525 10^-15,189,0]
```

No siempre es tan evidente qué función es mayor. En esos casos la continuidad nos permite utilizar el Teorema de los ceros de Bolzano para averiguarlo.

**Ejemplo 8.4.** Calculemos el área entre la función  $f(x) = 2x^3 + x^2 - 2x - 1$  y el eje OX. La función es un polinomio de grado 3 que puede, por tanto, tener hasta tres raíces reales. Si le echamos un vistazo a su gráfica

```
(%i41) f(x):=2x^3+x^2-2x-1$
(%i42) plot2d(f(x),[x,-2,2]);
(%o42)
```



se ve que tiene tres raíces. Pero ¿cómo sabíamos que teníamos que dibujar la función entre -2 y 2? En realidad el camino correcto es, en primer lugar, encontrar las raíces del polinomio:

```
(%i43) solve(f(x)=0,x);
(%o43) [x=-1/2,x=-1,x=1]
```

Ahora que sabemos las raíces se entiende por qué hemos dibujado la gráfica de la función en ese intervalo particular y no en otro. Si las raíces son  $-\frac{1}{2}$ ,  $-1$  y  $1$  sabemos que  $f$  no se anula en el intervalo en  $]-1, -\frac{1}{2}[$  ni en  $]-\frac{1}{2}, 1[$  pero, ¿cuál es su signo en cada uno de dichos intervalos? Aquí es donde entra en juego el Teorema de los ceros de Bolzano. Si  $f$ , una función continua, sí cambiase de signo en  $]-\frac{1}{2}, 1[$  o en  $]-1, -\frac{1}{2}[$  tendría que anularse, cosa que no ocurre. Por tanto,  $f$  siempre tiene el mismo signo en cada uno de esos intervalos. ¿Cuál? Sólo es cuestión de mirar el valor de  $f$  en un punto cualquiera. En  $]-1, -\frac{1}{2}[$

```
(%i44) f(-0.75);
(%o44) 0.21874999813735
```

la función es positiva. Y en  $]-\frac{1}{2}, 1[$

```
(%i45) f(0);
(%o45) -1
```

la función es negativa. Por tanto, la integral que queremos hacer es

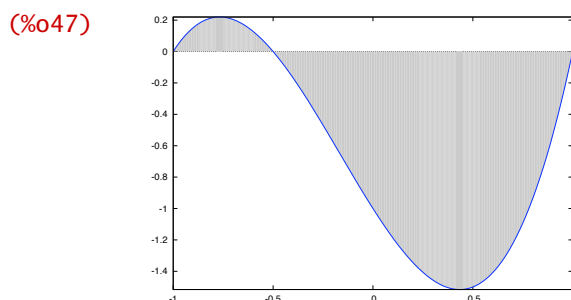
$$\int_{-1}^1 |f(x)| dx = \int_{-1}^{-\frac{1}{2}} -f(x) dx + \int_{-\frac{1}{2}}^1 f(x) dx,$$

o, lo que es lo mismo,

```
(%i46) integrate(f(x),x,-1,-1/2)+integrate(-f(x),x,-1/2,1);
(%o46) 71
      48
```

**Observación 8.5.** Por cierto, recuerda que podríamos haber usado la opción `filled_func` y `fill_color` para “maquillar” un poco el dibujo del área dibujando por un lado la función y por otro el área sombreada:

```
(%i47) draw2d(
      filled_func=0, fill_color=grey,
      explicit(f(x),x,-1,1),
      filled_func=false,
      color=blue, line_width=2,
      explicit(f(x),x,-1,1),
      xaxis=true);
```



### 8.3.2 Longitud de curvas

Si  $f$  es una función definida en el intervalo  $[a, b]$ , la longitud del arco de curva entre  $(a, f(a))$  y  $(b, f(b))$  se puede calcular mediante la fórmula

$$\text{longitud} = \int_a^b \sqrt{1 + f'(x)^2} dx$$

Por ejemplo, la semicircunferencia superior de radio 1 centrada en el origen es la gráfica de la función  $f(x) = \sqrt{1-x^2}$ , con  $x$  variando entre -1 y 1. Si aplicamos la fórmula anterior podemos calcular la longitud de una circunferencia.

```
(%i48) f(x):=sqrt(1-x^2);
(%o48) f(x):=sqrt(1-x^2)
(%i49) diff(f(x),x);
(%o49) -x/sqrt(1-x^2)
(%i50) integrate(sqrt(1+diff(f(x),x)^2),x,-1,1);
(%o50) pi
```

Observa que hemos calculado la longitud de media circunferencia, ya que la longitud de la circunferencia completa de radio 1 es  $2\pi$ .

Esto que hemos visto es un caso particular de la longitud de una curva en el plano. Ya vimos curvas cuando hablamos de dibujar en paramétricas. Una curva es una aplicación de la forma  $x \mapsto (f(x), g(x))$ ,  $a \leq x \leq b$ . La longitud de dicha curva es

$$\text{longitud} = \int_a^b \sqrt{f'(x)^2 + g'(x)^2} dx$$

Por ejemplo, la circunferencia unidad la podemos parametrizar como  $t \mapsto (\cos(t), \sin(t))$  con  $0 \leq t \leq 2\pi$ . Por tanto, la longitud de dicha circunferencia es

```
(%i51) integrate(sqrt(sin(t)^2+cos(t)^2),x,0,2*pi);
(%o51) 2pi
```

### 8.3.3 Volúmenes de revolución

Los cuerpos de revolución o sólidos de revolución son regiones de  $\mathbb{R}^3$  que se obtienen girando una región plana alrededor de una recta llamada eje de giro. En la Figura 8.4 tienes el sólido de revolución engendrado al girar alrededor del eje OX la gráfica de la función seno entre 0 y  $\pi$ .

#### Giro de una curva $y = f(x)$ respecto al eje X

Sea  $f : [a, b] \rightarrow \mathbb{R}$  una función continua. Girando la región del plano comprendida entre la curva  $y = f(x)$ , el eje de abscisas con  $x$  entre  $a$  y  $b$ , alrededor del eje OX obtenemos un sólido de revolución  $\Omega$  con volumen igual a

$$\text{Vol}(\Omega) = \pi \int_a^b f(x)^2 dx$$

Veamos algunos ejemplos.

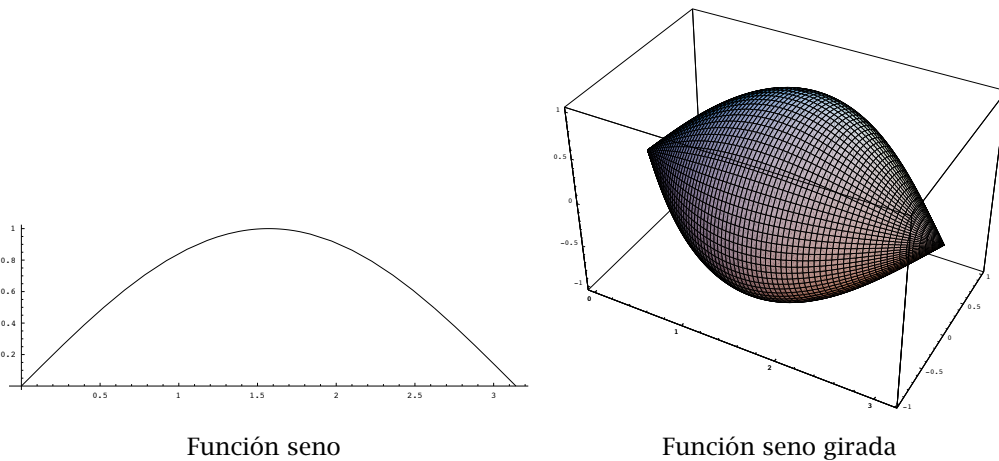


Figura 8.4

**Ejemplo 8.6.** Calculemos el volumen de una esfera de radio  $R$  viéndola como superficie de revolución. La curva que consideramos es  $f(x) = \sqrt{R^2 - x^2}$  con  $x \in [-R, R]$  y, por tanto, el volumen lo calculamos aplicando la fórmula anterior :

```
(%i52) %pi*integrate(R^2-x^2,x,-R,R);
(%o52) 4*pi*R^3/3
```

**Ejemplo 8.7.** Vamos ahora a calcular el volumen de un cono circular recto. Un cono circular recto de altura  $h$  y radio de la base  $R$  se obtiene girando la recta  $y = Rx/h$  entre  $0$  y  $h$ . Su volumen es igual a

```
(%i53) %pi*integrate(R^2*x^2/h^2,x,0,h);
(%o53) pi*h*r^2/3
```

**Giro de una curva  $y = f(x)$  respecto al eje  $Y$**

Consideremos la gráfica de una función positiva  $y = f(x)$  en un intervalo  $[a, b]$ . Por ejemplo la función  $1 - x^2$  en  $[0, 1]$ .

Si giramos respecto al eje  $OY$  obtenemos la Figura 8.5.

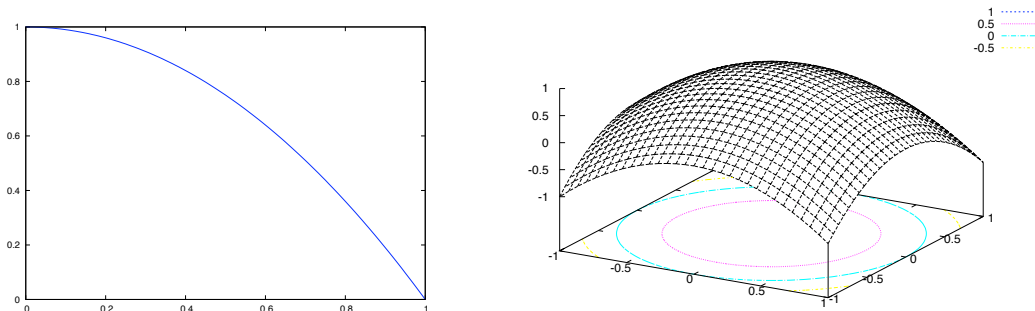


Figura 8.5 Función  $1 - x^2$  girada respecto al eje  $OY$

Pues bien, el volumen de la región así obtenida,  $\Omega$ , viene dado por

$$\text{Volumen}(\Omega) = 2\pi \int_a^b x f(x) dx$$

**Ejemplo 8.8.** Consideremos el toro  $T$  obtenido al girar el disco de centro  $(a, 0)$ ,  $a > 0$ , y radio  $R$  alrededor del eje  $OY$ . Puedes verlo para  $a = 2$  y  $R = 1$  en la Figura 8.6.

Por simetría, su volumen es el doble del volumen del sólido obtenido al girar la semicircunferencia  $y = \sqrt{R^2 - (x - a)^2}$ ,  $(a - R \leq x \leq a + R)$  alrededor del eje  $OY$ . Por tanto

$$\text{volumen}(T) = 4\pi \int_{a-R}^{a+R} x \sqrt{R^2 - (x - a)^2} dx$$

integral que calculamos con *wxMaxima*:

```
(%i54) 4*pi*integrate(x*sqrt(R^2-(x-a)^2),x,a-R,a+R);
(%o54) 2*pi^2*aR^2
```

Observa que, aunque aquí hemos escrito la salida automáticamente, sin embargo, *wxMaxima* hace varias preguntas sobre los valores de las constantes  $a$  y  $R$  para poder calcular la integral.

Gráficamente, podemos conseguir este efecto girando la circunferencia de radio 1 que podemos parametrizar de la forma  $(\cos(t), \sin(t))$ , pero que tenemos que trasladar dos unidades. Ése es el motivo de sumar 2 en la siguiente representación en coordenadas paramétricas.

```
(%i55) with_slider_draw3d(
n,0.1*pi*range(1,20),
surface_hide=true,
xrange=[-3,3],
yrange=[-3,3],
parametric_surface(cos(u)*(2+cos(v)),sin(u)*(2+cos(v)),sin(v),
u,0,n,v,0,2*pi)
)
```

En la Figura 8.7 tienes algunos de los pasos intermedios de esta animación.

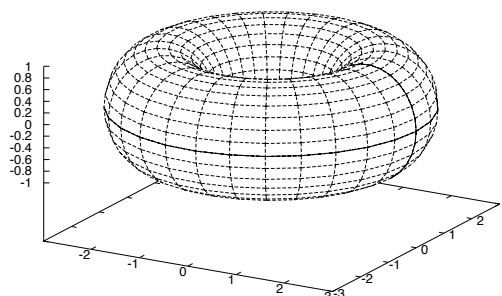


Figura 8.6 Toro

### 8.3.4 Área de superficies de revolución

Igual que hemos visto cómo podemos calcular el volumen de una figura obtenida girando una función respecto a alguno de los ejes, también podemos calcular el área de la superficie  $\Omega$  obtenida al girar respecto al eje  $OX$  una función  $f$ . El área al girar  $f$  en el intervalo  $[a, b]$  es

$$\text{área}(\Omega) = 2\pi \int_a^b f(x) \sqrt{1 + f'(x)^2} dx$$

Por ejemplo, una esfera de radio 1 la podemos obtener girando respecto del eje  $OX$  la función  $\sqrt{1 - x^2}$ .

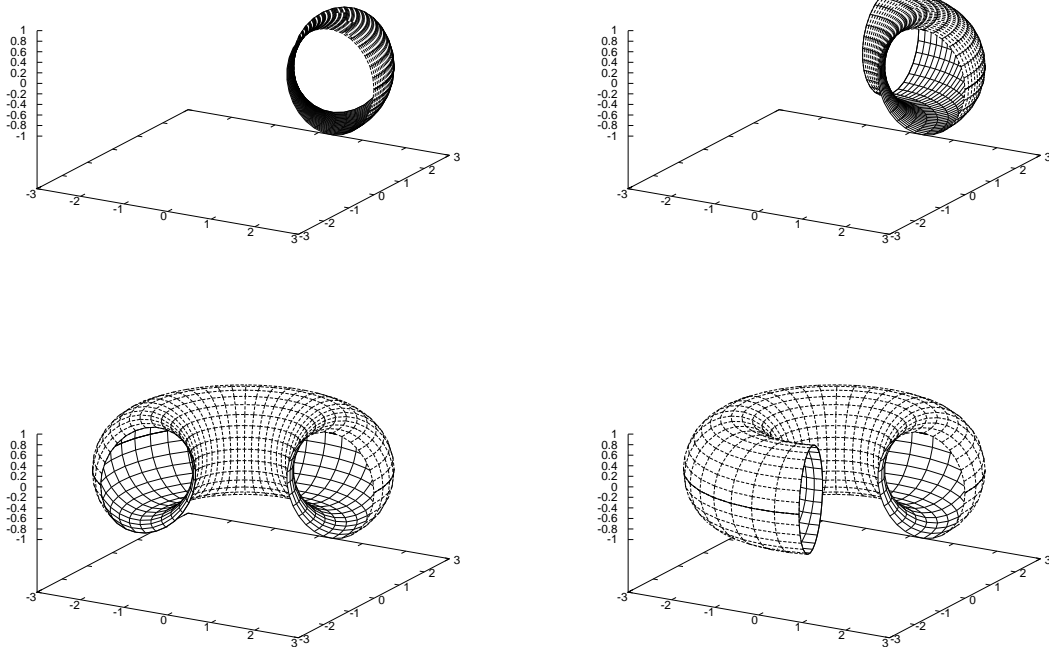


Figura 8.7 Cómo conseguir un toro girando una circunferencia

```
(%i56) integrate(f(x)*sqrt(1+diff(f(x),x)^2), x, -1, 1);
(%o56) 2 * pi^2 * a * R^2
```

## 8.4 Ejercicios

### Integración

#### Ejercicio 8.1

Calcula una primitiva de las siguientes funciones.

a)  $f(x) = \cos^5(x)$ ,

b)  $f(x) = 1/(1 + x^4)$ ,

c)  $f(x) = \sqrt{1 - x^2}$ ,

¿Sabes calcularlas sin usar *Maxima*?

#### Ejercicio 8.2

Calcula numéricamente las integrales en el intervalo  $[0, 2\pi]$  de los primeros 20 polinomios de Taylor de la función coseno.

## Teorema fundamental del Cálculo

### Ejercicio 8.3

Calcula la derivada de la función  $f(x) = \int_{\sqrt{x}}^{x^2+1} \sin(t) dt$ .

### Ejercicio 8.4

Estudia los extremos relativos de la función  $f(x) = \int_0^{(2x-7)^2} t^3 - 2t dt$

## Área entre curvas

### Ejercicio 8.5

Calcula:

- área limitada por  $y = xe^{-x^2}$ , el eje  $OX$ , la ordenada en el punto  $x = 0$  y la ordenada en el máximo.
- área de la figura limitada por la curva  $y = x^3 - x^2$  y el eje  $OX$ .
- área comprendida entre la curva  $y = \tan(x)$ , el eje  $OX$  y la recta  $x = \pi/3$ .
- área del recinto limitado por las rectas  $x = 0$ ,  $x = 1$ ,  $y = 0$  y la gráfica de la función  $f: \mathbb{R} \rightarrow \mathbb{R}$  definida por  $f(x) = \frac{1}{(1+x^2)^2}$ .
- las dos áreas en los que la función  $f(x) = |x| - x\sin(x)e^x$  divide a la bola unidad  $x^2 + y^2 = 1$ .

### Ejercicio 8.6

Calcula el área entre las curvas:

- $y = -x^2 - 2x$  e  $y = x^2 - 4$ , para  $-3 \leq x \leq 1$ .
- $y^2 = x$ ,  $x^2 + y^2 = 8$ .
- $x = 12y^2 - 12y^3$  y  $x = 2y^2 - 2y$ .
- $y = \sec^2(x)$ ,  $y = \tan^2(x)$ ,  $-\pi/4 \leq x \leq \pi/4$ .
- $(y - x)^2 = x - 3$ , y  $x = 7$ .
- $y = x^4 + x^3 + 16x - 4$  y  $y = x^4 + 6x^2 + 8x - 4$ .
- $y = xe^{-x}$ ,  $y = x^2e^{-x}$ ,  $x \geq 0$ .

## Longitud de curvas

### Ejercicio 8.7

- Calcula la longitud del arco de la cicloide  $x = t - \sin(t)$ ,  $y = 1 - \cos(t)$ ,  $(0 \leq t \leq 2\pi)$ .
- Calcula la longitud del arco de curva  $y = x^2 + 4$  entre  $x = 0$  y  $x = 3$ .

### Ejercicio 8.8

Calcula la longitud de una circunferencia de radio arbitrario  $R$ .

### Ejercicio 8.9

Sea  $f(x) = \cos(x) + e^x$  y  $P$  su polinomio de orden 5 centrado en el origen. ¿Cuál es la diferencia entre las longitudes de las gráficas de  $f$  y de  $P$  en el intervalo  $[0, 3]$ ?

## Volumen de cuerpos de revolución

### Ejercicio 8.10

Calcular el volumen del sólido generado al rotar respecto al eje  $OX$  las siguiente curvas:





# Diferenciación

## 9

En este capítulo debes aprender a calcular y evaluar derivadas parciales de cualquier orden de una función de varias variables, a aplicar el teorema de la función implícita así como a calcular extremos de funciones reales de varias variables, tanto relativos como condicionados o absolutos. Aprovecharemos la capacidad gráfica de *Maxima* para tener una idea más clara de nociones como el plano tangente.

### 9.1 Derivadas parciales de una función

El primer objetivo será aprender a calcular derivadas parciales, gradientes, hessianos... de funciones de varias variables, y aplicar todo ello al cálculo de extremos relativos y condicionados.

Recordemos que el comando `diff` nos servía para calcular derivadas de funciones de una variable. Pues bien, también nos va a servir para calcular derivadas parciales de campos escalares de varias variables. Esto es

`diff(f(x1, x2 ···), x1, n1, x2, n2, ···)` derivada parcial de la función  $f(x_1, x_2, \dots)$  respecto a  $x_1$ ,  $n_1$  veces, respecto a  $x_2$ ,  $n_2$  veces,...

Calculemos como ejemplo  $\frac{\partial^3 f}{\partial^2 x \partial y}$  y  $\frac{\partial^2 f}{\partial x \partial y}$ , donde  $f(x, y) = \sin(x) \cos(y^2)$ :

```
(%i1) f(x,y):= sin(x)*cos(y^2)$
(%i2) diff(f(x,y),x,2,y,1);
(%o2) 2 sin(x)y sin(y^2)
(%i3) diff(f(x,y),x,1,y,1);
(%o3) -2cos(x)y sin(y^2)
```

Con las derivadas parciales de primer orden construimos el vector gradiente ( $\nabla f(a)$ ) de un campo escalar en un punto, y la matriz jacobiana ( $J_f(a)$ ) de un campo vectorial en un punto. Para ambos cálculos utilizaremos el mismo comando: `jacobian(f, x)`. Por ejemplo:

**jacobian**

```
(%i4) jacobian([f(x,y)], [x,y]);
(%o4) [cos(x) cos(y^2)  -2sin(x)y sin(y^2)]
```

El resultado es una matriz fila de dos elementos, o lo que es lo mismo, un vector de dos componentes. Mientras que si ahora aplicamos el mismo comando a un campo vectorial, tenemos:

```
(%i5) jacobian([exp(x+y), x^2-y^2], [x,y]);
```

```
(%o5) [ %ey+x %ey+x
        2x      -2y ]
```

Ahora bien, si queremos calcular la matriz hessiana de un campo escalar, utilizaremos el comando `hessian` de la forma siguiente:

```
(%i6) hessian(f(x,y),[x,y]);
(%o6) [ -sin(x)cos(y2)      -2cos(x)y sin(y2)
        -2cos(x)y sin(y2)  -2sin(x)sin(y2)-4sin(x)y2cos(y2) ]
```

Recuerda que las derivadas cruzadas de segundo orden coinciden para funciones “suficientemente buenas”. Además, hay que comentar que, si estamos calculando el gradiente (o matriz jacobiana) de un campo escalar, mientras en el comando `hessian` la función se escribe  $f(x, y)$ , en el comando `jacobian` la función hay que escribirla entre corchetes. La razón es que este comando sirve tanto para campos escalares como vectoriales.

### 9.1.1 Plano tangente

Ya que sabemos cómo se calculan las derivadas parciales de una función, vamos a intentar representar gráficamente el plano tangente y cómo se obtiene a partir de las derivadas parciales.

Comencemos, por ejemplo, con la función

```
(%i7) f(x,y):=1-(x2+y2)$
```

y vamos a calcular en el punto  $(1, 1)$  sus derivadas parciales. La definición de derivada parcial respecto a la primera variable era

$$\frac{\partial f}{\partial x}(1, 1) = \lim_{h \rightarrow 0} \frac{f(1+h, 1) - f(1)}{h}$$

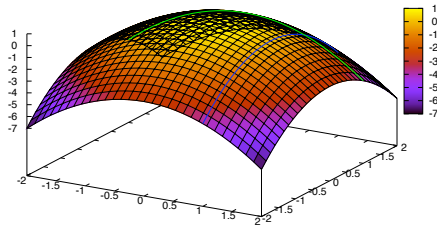
Lo que hacemos es trabajar únicamente con la función definida sobre la recta que pasa por el punto  $(1, 1)$  y es esa función (de una variable) la que derivamos. La segunda derivada parcial tiene una definición análoga. Dibujemos la gráfica de la función y la imagen de cada una de esas dos rectas. Antes de nada, cargamos el módulo `draw` y definimos las parciales de  $f$ ,

```
(%i8) load(draw)$
(%i9) define(parcialf1(x,y),diff(f(x,y),x,1))$
(%i10) define(parcialf2(x,y),diff(f(x,y),y,1))%
```

dibujamos la función y las curvas cuyas pendientes nos dan las derivadas parciales

```
(%i11) draw3d(enhanced3d=true,
  explicit(f(x,y),x,-2,2,y,-2,2),
  color=blue,
  line_width=2,
  parametric(1,t,f(1,t),t,-2,2),
  color=green,
  line_width=2,
  parametric(t,1,f(t,1),t,-2,2))$
```

```
(%o11)
```



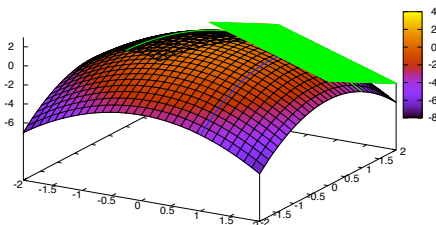
Los vectores tangentes a las dos curvas que hemos dibujado son los dos vectores que generan el plano tangente. Por tanto su ecuación será

$$\begin{aligned} z &= f(1,1) + \frac{\partial f}{\partial x}(1,1)(x-1) + \frac{\partial f}{\partial y}(1,1)(y-1) \\ &= f(1,1) + \langle \nabla f(1,1), (x-1, y-1) \rangle \end{aligned}$$

Para acabar nos falta añadir el plano tangente a la gráfica anterior:

```
(%i12) draw3d(enhanced3d=true,
  explicit(f(x,y),x,-2,2,y,-2,2),
  color=blue,
  line_width=2,
  parametric(1,t,f(1,t),t,-2,2),
  color=green,
  line_width=2,
  parametric(t,1,f(t,1),t,-2,2)),
  enhanced3d=true,
  parametric_surface(x,y,f(1,1)+parcialf1(1,1)*(x-1)
    +parcialf2(1,1)*(y-1),x,0,2,y,0,2))$
```

```
(%o12)
```



¿Cómo se podría hacer esto con derivadas direccionales en lugar de derivadas parciales? Más concretamente, dibuja la superficie y la recta que pasa por un punto  $(a,b)$  con dirección  $\vec{v}$  y su

imagen, la curva cuya pendiente nos da la derivada direccional. Para hacer todo esto, elige tú el punto y la dirección.

## 9.2 Curvas y vectores tangentes

El paso a funciones de varias variables, ya sean escalares o vectoriales, puede dar lugar a errores por intentar trasladar literalmente nuestros conocimientos de funciones de una variable.

Hasta ahora hemos dibujado el plano tangente de una función de dos variables con valores escalares. El otro ejemplo sencillo de función de varias variables es una función de una variable pero con valores vectoriales. Por ejemplo, la función  $t \mapsto (\cos(t), \sin(t))$ . Para su representación necesitaríamos 3 dimensiones, una para el dominio y dos para la imagen, pero se suele representar en dos como una curva en coordenadas paramétricas: pensamos en la variable  $t$  como tiempo y  $(\cos(t), \sin(t))$  representa la posición de una partícula en un momento  $t$ . En este caso su derivada es  $t \mapsto (-\sin(t), \cos(t))$ . Para hacernos una idea, vamos a representar en cada punto el correspondiente vector tangente.

En primer lugar definimos las funciones y el vector tangente (no te olvides de cargar el módulo draw:

```
(%i13) load(draw)$
(%i14) c(t):=[cos(t),sin(t)];
(%o14) c(t):=[cos(t),sin(t)]
(%i15) define(tangente(t),[diff(cos(t),t),diff(sin(t),t)]);
(%o15) tangente(t):=[-sin(t),cos(t)]
```

y ahora realizamos la animación.

```
(%i16) with_slider_draw(
      a,2*%pi*range(1,20)/20,
      color=blue,
      line_width=2,
      parametric(cos(t),sin(t),t,0,2*%pi),
      color=red,
      head_length=0.2,
      head_type='empty,
      vector(c(a),tangente(a)),
      xrange=[-2,2],
      yrange=[-2,2])$
```

En la Figura 9.1 puedes ver algunos pasos intermedios de la representación

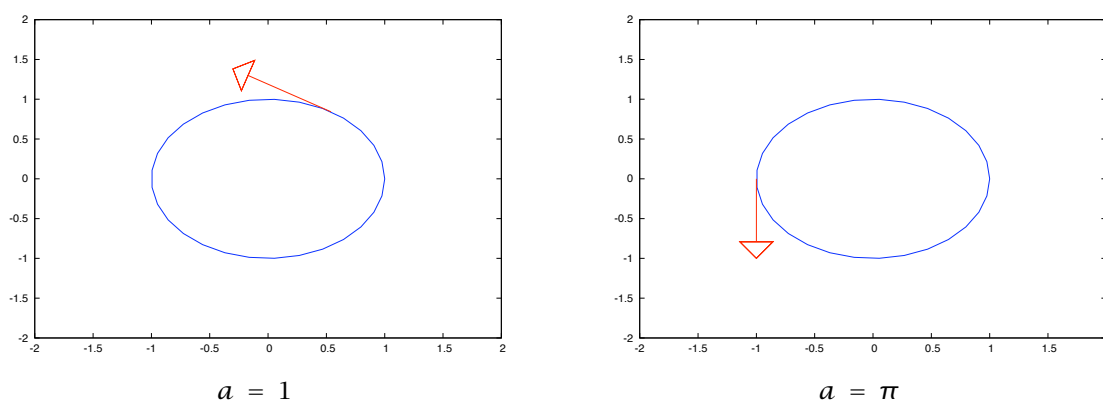


Figura 9.1 Curva y vectores tangentes

La función  $t \mapsto (\cos(t), \sin(t))$  es continua y es diferenciable y el rastro que deja cuando variamos  $t$  en cualquier intervalo de longitud mayor o igual que  $2\pi$  es una circunferencia.

Observa la gráfica de la curva que aparece en la Figura 9.2 de al lado. ¿Dirías que es “derivable”? Si has respondido que no, supongo que la justificación usual es que “tiene un pico”. Pero, ¿cómo sabes que es continua?. Tendemos a pensar en la figura como la gráfica de una función de una variable pero ¿cuál es la función de la que estamos afirmando (o no) que es continua o derivable?

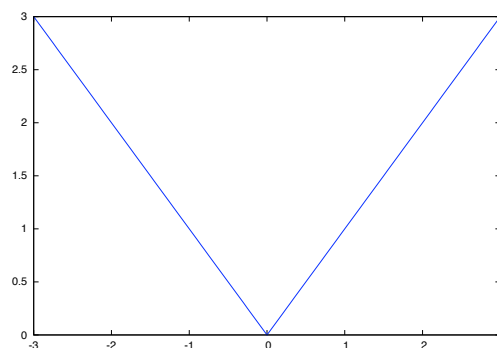


Figura 9.2 ¿Es diferenciable?

Si estás pensando que lo que hemos representado es la función valor absoluto, de acuerdo. Ahora bien, piensa por un momento que estamos ante una curva en coordenadas paramétricas. ¿Qué es ser continua en este caso? Hablando muy a la ligera (y bastante mal) que para valores cercanos sus imágenes también lo estén. Para intentar visualizarlo, vamos a dibujar la curva y un punto que nos vaya indicando en cada momento por dónde vamos.

```
(%i17) with_slider_draw(
      n, 6*range(1,20)/20-3,
      color=blue,
      nticks=250,
      parametric(t,abs(t),t,-3,3),
      color=red,
      point_type=filled_circle,
      point_size=2,
      points([[n^3,n^2*abs(n)]]),
      xrange=[-3,3],yrange=[0,3]
    )$
```

La continuidad de la función se traduce en que el punto no vaya “dando saltos” como puedes ver en la animación anterior. ¿Y que ocurre con la diferenciableidad? En este caso tenemos que fijarnos en cómo se mueve el vector tangente. Hay una gran diferencia entre recorrer la figura así:

```
(%i18) with_slider_draw(
      n,6*range(1,20)/20-3.05,
      color=blue,nticks=250,
      parametric(t,abs(t),t,-3,3),
      color=red,
      head_length=0.2,head_type='empty,
      vector([n,abs(n)], [1,signum(n)]),
      xrange=[-3,3],yrange=[-1,3]
    )$
```

que vemos cómo el vector tangente cambia bruscamente al llegar al origen y recorrer la curva de la forma  $t \mapsto (t^3, t^2 |t|)$ :

```
(%i19) with_slider_draw(
      n,6*range(1,20)/20-3,
      color=blue,nticks=250,
      parametric(t,abs(t),t,-3,3),
      color=red,
      head_length=0.2,head_type='empty,
      vector([n^3,n^2*abs(n)], [3*n^2,3*n*abs(n)]),
      xrange=[-3,3],yrange=[-1,3]
    )$
```

en la que podemos ver cómo podemos evitar ese problema frenando al llegar al origen (la derivada en ese punto vale cero) para volver acelerar al “doblar la esquina”

Si hay algo que debería quedar claro después de esto es que lo importante es cómo se recorre la curva y no la gráfica de esta.

### 9.3 Funciones definidas implícitamente

La orden `diff` deriva considerando todas las variables independientes. Por ejemplo,

```
(%i20) diff(2*x*y+cos(x*y)+y,x);
(%o20) 2y-y sin(xy)
(%i21) diff(2*x*y+cos(x*y)+y,y);
(%o21) -x sin(xy)+2x+1
```

<code>depends(var1, var2)</code>	<code>var1</code> es función de <code>var2</code>
<code>dependencies</code>	lista de variables dependientes
<code>remove(var, dependency)</code>	hacer <code>var</code> una variable independiente

pero, en ocasiones, alguna de las variables no es independiente. En *Maxima* podemos imponer este tipo de condiciones con la orden `depends`. Con ella podemos asumir que una variable o una lista

de variables depende de otra o de otras. En el caso anterior, podemos por ejemplo suponer que  $y$  es función de  $x$ :

```
(%i22) depends(y,x);
(%o22) [y(x)]
```

con lo que al derivar respecto de  $x$  nos quedaría

```
(%i23) diff(2*x*y+cos(x*y)+y,x);
(%o23) -(x(d/dx y)+y)sin(xy)+2x(d/dx y)+d/dx y+2y
```

La variable `dependencies` lleva la cuenta de todas las dependencias que hayamos definido. Hasta ahora sólo una:

```
(%i24) dependencies;
(%o24) [y(x)]
```

y, para que no haya problemas posteriormente, podemos anular esta dependencia con la orden `remove`:

```
(%i25) remove(y,dependency);
(%o25) done
```

con lo que ahora ya no tenemos ninguna dependencia definida.

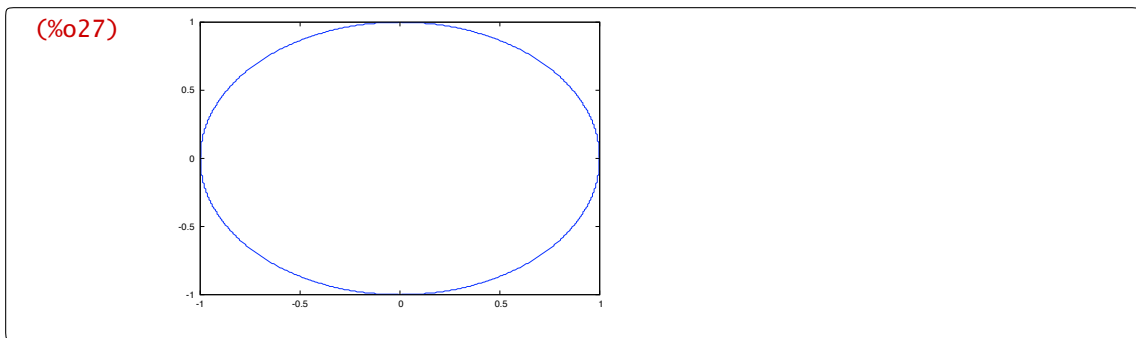
```
(%i26) dependencies;
(%o26) [ ]
```

## Teorema de la función implícita

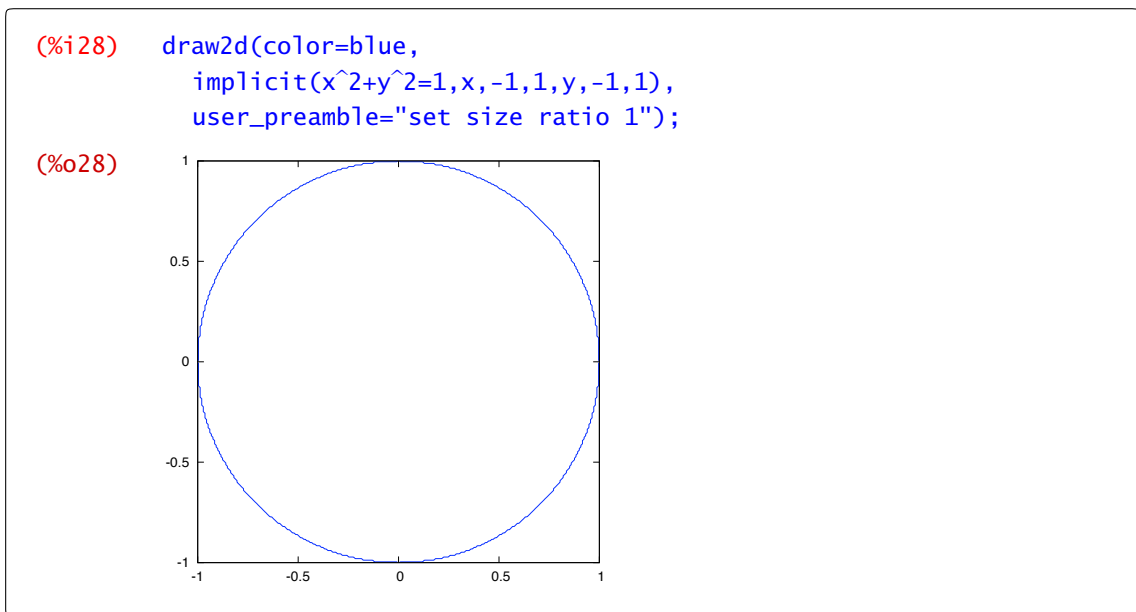
El Teorema de la función implícita nos dice cuándo podemos despejar unas variables en función de otras y qué propiedades verifica la función que nos da ese “despeje”.

Consideremos la ecuación  $x^2 + y^2 = 1$ . ¿Podemos despejar  $y$  en función de  $x$  o  $x$  en función de  $y$ ? La respuesta es que depende. Aunque estoy seguro de que sabes que los puntos que verifican  $x^2 + y^2 = 1$  representan una circunferencia de radio uno centrada en el origen, hagamos como que no sabemos nada y dibujemos dicho conjunto.

```
(%i27) draw2d(color=blue,
             implicit(x^2+y^2=1,x,-1,1,y,-1,1));
```



Mejor un poco más redondo,



Es claro que no hay ninguna función ni de  $x$  ni de  $y$  cuya gráfica puede ser la circunferencia. Ahora bien, para algunos trozos de la circunferencia sí es cierto. Por ejemplo si sólo nos interesa qué pasa en el punto  $(0, 1)$  y sus alrededores entonces seguro que sí sabes despejar  $y$  en función de  $x$ :

$$y(x) = \sqrt{1 - x^2}.$$

Esta fórmula no nos vale para el punto  $(0, -1)$ , habría que añadir un signo menos, pero también se puede despejar  $y$  en función de  $x$ . ¿Qué ocurre con el punto  $(1, 0)$ ? Pues que no se puede, cualquier trozo de circunferencia que contenga al punto  $(1, 0)$  (sin ser un extremo) no puede ser la gráfica de una función dependiente de  $x$ . Ahora bien, sí que podemos despejar  $x$  en función de  $y$ :

$$x(y) = \sqrt{1 - y^2}.$$

Consideremos la función  $f(x, y) = x^2 + y^2 - 1$ . Para poder despejar  $y$  en función de  $x$  en un entorno de un punto  $(a, b)$  hace falta que

a)  $f(a, b) = 0,$

b)  $\frac{\partial f}{\partial y}(a, b) \neq 0,$

y que  $f$  sea, al menos, de clase uno. En ese caso



$$y'(x) = -\frac{\frac{\partial f}{\partial x}(a, b)}{\frac{\partial f}{\partial y}(a, b)}$$

¿De dónde sale esta fórmula? Es sencillo, de derivar la ecuación y despejar:

$$f(x, y(x)) = 0 \implies \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial x} = 0.$$

Esto mismo lo podemos hacer con *Maxima*: definimos la ecuación,

```
(%i29) eq:x^2+y^2=1;
(%o29) x^2+y^2=1
```

decimos que  $y$  depende de  $x$ ,

```
(%i30) depends(y, x);
(%o30) [y(x)]
```

derivamos,

```
(%i31) diff(eq, x);
(%o31) 2y(d/dx y)+2x=0
```

y despejamos  $y'(x)$ ,

```
(%i32) solve(%, diff(y, x));
(%o32) [d/dx y=-x/y]
```

En el siguiente ejemplo se puede ver cómo podemos aplicar esto a una ecuación con 3 variables.

**Ejemplo 9.1.** Comprobar que la ecuación

$$z \arctan(1 - z^2) + 3x + 5z - 8y^3 = 0, (1, 1, 1)$$

verifica las condiciones del Teorema de la función implícita en el punto indicado y, por tanto, define a  $z$  como función de  $x$  e  $y$  en un entorno del punto  $(1, 1)$ . Obtener  $\frac{\partial z}{\partial x}$ ,  $\frac{\partial z}{\partial y}$ .

Antes de empezar, borremos variables y posibles dependencias que tenemos de la explicación anterior.

```
(%i33) remvalue(all);
(%o33) [eq]
(%i34) dependencies;
(%o34) [y(x)]
```

```
(%i35) remove(y,dependency);
(%o35) done
```

Definimos la ecuación,

```
(%i36) eq:z*atan(1-z^2)+3*x+5*z-8*y^3;
(%o36) -z atan(z^2-1)+5z-8y^3+3x
```

comprobamos que el punto verifica la ecuación,

```
(%i37) eq,x=1,y=1,z=1;
(%o37) 0
```

y que la derivada respecto de  $z$  no se anula,

```
(%i38) diff(eq,z);
(%o38) -atan(z^2-1) - \frac{2z^2}{(z^2-1)^2+1} +5
(%i39) %,x=1,y=1,z=1;
(%o39) 3
```

con lo que el teorema de la función implícita nos da que  $z$  depende de  $x$  e  $y$  en un entorno de  $(1, 1)$ ,

```
(%i40) depends(z,[x,y]);
(%o40) [z(x,y)]
```

derivamos,

```
(%i41) diff(eq,x);
(%o41) -\left(\frac{d}{dx}z\right)atan(z^2-1) - \frac{2z^2}{(z^2-1)^2+1}\left(\frac{d}{dx}z\right) + 5\left(\frac{d}{dx}z\right) + 3
```

y despejamos  $\frac{\partial z}{\partial x}$ ,

```
(%i42) solve(%,diff(z,x));
(%o42) [\frac{d}{dx}z = \frac{3z^4-6z^2+6}{(z^4-2z^2+2)atan(z^2-1)-5z^4+12z^2-10}]
```

El cálculo de la derivada respecto de  $y$  se hace de manera similar y evaluar en un punto tampoco debería ofrecerte ninguna dificultad.

## 9.4 Extremos relativos

El método para encontrar extremos relativos de funciones de varias variables suficientemente derivables consiste en buscar primero los puntos críticos, es decir, puntos donde se anula el gradiente, y después estudiar la matriz hessiana en esos puntos. Los resultados que conocemos nos aseguran que todos los puntos extremos de una función están entre los puntos críticos, con lo que una vez calculados éstos nos dedicaremos a estudiar la matriz hessiana en ellos, viendo si es definida, indefinida, semidefinida... Para ello podemos usar el criterio de los valores propios: si todos son del mismo signo, la matriz es definida y hay extremo; si aparecen valores propios de distinto signo es indefinida y hay punto de silla; en otro caso, la matriz es semidefinida y el criterio no decide.

**Ejemplo 9.2.** Calculemos los extremos relativos de la función  $f(x, y) = x^3 + 3xy^2 - 15x - 12y$ . Primero definimos la función, y calculamos su gradiente y su matriz hessiana.

```
(%i43) f(x,y):= x^3+3*x*y^2-15*x-12*y;
      j:jacobian([f(x,y)],[x,y]);
      define(h(x,y),hessian(f(x,y),[x,y]));

(%o43) f(x,y):=x^3+3xy^2+(-15)x+(-12)y
(%o44) [3x^2+3y^2-15  6xy-12]
(%o45) [6x  6y ]
      [6y  -6x]
```

Para calcular los puntos críticos podemos irnos a **Ecuaciones** → **Resolver sistema algebraico** e ir rellenando los datos que nos van pidiendo. De esta forma entra en acción el comando `algsys` para resolver sistemas de ecuaciones. A la hora de llamar a las ecuaciones, lo haremos de la siguiente forma:

```
(%i46) pcrit:algsys([j[1,1],j[1,2]],[x,y]);
(%o46) [[x=2,y=1],[x=1,y=2],[x=-1,y=-2],[x=-2,y=-1]]
```

donde  $j[1,1]$  es el primer elemento del gradiente, es decir  $\frac{\partial f}{\partial x}$  y  $j[1,2]$  es el segundo, es decir  $\frac{\partial f}{\partial y}$ . Observad que el resultado es una lista de listas de puntos; lista que además hemos llamado `pcrit` para luego poder acudir a ella a la hora de evaluar la matriz hessiana en los puntos críticos obtenidos. Calculamos entonces la matriz hessiana, la evaluaremos en cada uno de los puntos críticos, y la clasificaremos haciendo uso del comando `eigenvalues` que nos da la lista de valores propios, seguidos de la lista de sus multiplicidades.

```
(%i47) hessian(f(x,y),[x,y]);
(%o47) [6x  6y ]
      [6y  6x]
```

Entonces, ahora, vamos evaluando la matriz  $h(x,y)$  en cada punto crítico y calculamos los valores propios en cada una de ellas. Lo hacemos en uno de los puntos, y de forma análoga se haría en los restantes.

```
(%i48)  eigenvalues(h(2,1));
(%o48)  [[6,18],[1,1]]
```

Por tanto, en el punto  $(2, 1)$  la función  $f$  presenta un mínimo relativo al ser sus dos valores propios positivos, esto es, la forma cuadrática asociada a la matriz hessiana de  $f$  en dicho punto es definida positiva. Así mismo, en  $(-2, -1)$  hay un máximo relativo, y en los puntos  $(1, 2)$  y  $(-1, -2)$  se tienen puntos de silla.

**Ejemplo 9.3.** Calculemos los extremos relativos de  $g(x, y) = (x^2 + 3y^2)e^{1-x^2-y^2}$ . Comenzamos definiendo  $g$ :

```
(%i49)  g(x,y):=(x^2+3*y^2)*exp(1-x^2-y^2)$
```

Calculamos ahora los puntos críticos como en el ejemplo anterior y tenemos:

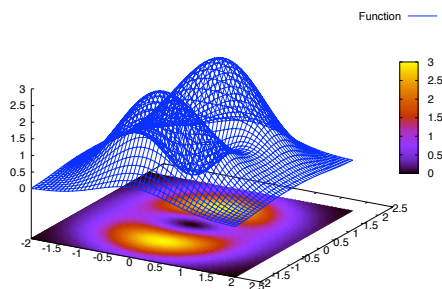
```
(%i50)  j:jacobian([g(x,y)],[x,y])$
(%i51)  pcrit:algsys([j[1,1],j[1,2]],[x,y]);
(%o51)  [[x=0,y=0],[x=-1,y=0],[x=1,y=0],[x=0,y=-1],[x=0,y=1]]
```

y vamos a calcular la matriz hessiana y a clasificarla en cada uno de los puntos críticos encontrados:

```
(%i52)  hessian(g(x,y)],[x,y])$
(%i53)  h(x,y):='('$
(%i54)  eigenvalues(h(0,0));
(%o54)  [[2%e,6%e],[1,1]]
```

Así, en el punto  $(0, 0)$  tenemos un mínimo y, razonando de la misma forma en los otros cuatro puntos, dos puntos de máximo en  $(0, -1)$  y  $(0, 1)$ , así como dos puntos de silla en  $(-1, 0)$  y  $(1, 0)$ . Podemos observar esto en la gráfica de la función, mejor aún si a la vez dibujamos el mapa de alturas:

```
(%i55)  plot3d((x^2+3*y^2)*exp(1-x^2-y^2), [x,-2,2], [y,-2,2],
[plot_format,gnuplot],[grid,50,50],
[gnuplot_preamble, "set pm3d at b"]$
(%o55)
```



## 9.5 Extremos condicionados

Usaremos el método de los multiplicadores de Lagrange para calcular extremos condicionados. Se trata de optimizar una función de varias variables en el conjunto de puntos que verifiquen una cierta ecuación o ecuaciones. Dada una función suficientemente diferenciable, y una curva o superficie en forma implícita, el método consiste en encontrar los puntos de dicha curva o superficie que verifican un sistema auxiliar, llamado “sistema de Lagrange”, que dará el equivalente a los puntos críticos; después habrá una condición sobre el hessiano equivalente a la condición para extremos relativos. Hay que notar que no todas las condiciones son válidas, sino que se ha de imponer una hipótesis técnica, pero en la práctica supondremos hecha la comprobación.

**Ejemplo 9.4.** Calculemos los extremos de la función  $f(x, y) = x^2 - y^2$  condicionados a la ecuación  $g(x, y) = x^2 + y^2 - 1 = 0$ . Comenzamos definiendo la función  $f$ , la condición, y la función auxiliar de Lagrange:

```
(%i56) f(x,y):=x^2-y^2$
      g(x,y):=x^2+y^2-1$ F(x,y,a):= f(x,y)-a*g(x,y)$
(%i57) z:F(x,y,a);
(%o57) -a(y^2+x^2-1)-y^2+x^2
```

Definimos el gradiente y planteamos el sistema de Lagrange:

```
(%i58) j:jacobian([z],[x,y])
(%i59) pcrit:algsys([j[1,1],j[1,2],g(x,y)],[x,y,a])
(%o59) [[x=1,y=0,a=1],[x=-1,y=0,a=1],[x=0,y=-1,a=-1],[x=0,y=1,a=-1]]
```

Obtenemos 4 puntos críticos, y para ver si son o no extremos, estudiamos la matriz hessiana auxiliar  $h(x, y, a)$  restringida al espacio tangente a la curva en los puntos críticos. Desafortunadamente, no podemos aplicar de una vez la asignación de valores, sino que deberemos ir punto a punto. El hessiano es

```
(%i60) define(h(x,y,a),hessian(F(x,y,a),[x,y]));
(%o60) h(x,y,a):=[ 2-2 a    0
                  0    -2 a-2 ]
```

En el primer punto, el hessiano es

```
(%i61) h(x,y,a),pcrit[1];
(%o61) [ 0  0
        0 -4 ]
```

que es semidefinido negativo. Tenemos que restringirnos al núcleo de la derivada de  $g$  en dicho punto crítico. Veamos, en primer lugar calculamos el jacobiano de  $g$ ,

```
(%i62) jacobian([g(x,y)], [x,y]);
(%o62) [ 2x  2y ]
```

evaluamos en el primer punto crítico,

```
(%i63) ev(%,pcrit[1]);
(%o63) [ 2  0 ]
```

y le calculamos el núcleo

```
(%i64) nullspace(%);
(%o64) span([ 0 ]
            [ 2 ])
```

La matriz hessiana restringida al núcleo tiene como única entrada

```
(%i65) [0,2].(ev(h(x,y,a),pcrit[1])).[0,2];
(%o65) -16
```

o sea que es definida negativa por lo que hay un máximo condicionado en el punto (1, 0). Para terminar el problema, repite el procedimiento con el resto de los puntos críticos.

**Ejemplo 9.5.** Calcular los extremos relativos de  $f(x, y, z) = x^2 + y^2 + z^2$  condicionados a la superficie  $g(x, y, z) = 2x + 2y - z + 3 = 0$ . Comenzamos con las definiciones:

```
(%i66) f(x,y,z):=x^2+y^2+z^2$ g(x,y,z):=2*x+2*y-z+3$
F(x,y,z,a):= f(x,y,z)-a*g(x,y,z)$
(%i67) t:F(x,y,z,a)$
(%o67) z^2-a(-z+2y+2x+3)+y^2+x^2
```

Definimos el gradiente de  $t$  y resolvemos el sistema de Lagrange:

```
(%i68) j:jacobian([t], [x,y,z])$
(%i69) pcrit:algsys([j[1,1],j[1,2],j[1,3],g(x,y,z)], [x,y,z,a]);
(%o69) [[x=2/3,y=-2/3,z=1/3,a=-2/3]]
```

y estudiemos el hessiano de la función auxiliar de Lagrange. En este caso

```
(%i70) hessian(t, [x,y,z]);
```

$$(\%o70) \quad \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

la matriz hessiana es definida positiva y, por tanto,  $f$  tiene un mínimo condicionado en el punto  $(\frac{2}{3}, \frac{2}{3}, \frac{1}{3})$ .

## 9.6 Extremos absolutos

En conjuntos compactos tenemos garantizada la existencia de extremos absolutos para las funciones continuas. Esto nos simplifica su cálculo: los extremos absolutos son extremos relativos, si se encuentran en el interior del dominio, o extremos condicionados, si se encuentran en la frontera. Vamos a buscar posibles candidatos a extremos relativos o a extremos condicionados y entre ellos tienen que estar los máximos y mínimos absolutos.

Dependiendo del aspecto de la frontera utilizaremos el método de los multiplicadores de Lagrange o trabajaremos directamente con la función. Comencemos con un ejemplo de este último caso.

**Ejemplo 9.6.** Vamos a calcular los extremos absolutos de la función  $f(x, y) = x^2(y - x)$  en el triángulo de vértices  $(0, 0)$ ,  $(1, 0)$  y  $(1, 1)$ .

En el interior, buscamos puntos críticos de la función:

```
(%i71) f(x,y):=x^2*(x-y)$
(%i72) j:jacobian([f(x,y),[x,y]])$
(%i73) algsys([j[1,1],j[1,2]],[x,y]);
(%o73) [[x=0,y=%r1]]
```

La interpretación que damos de las soluciones es que las únicas soluciones son los puntos de la forma  $(0, y)$  que, evidentemente, no pertenecen al interior del dominio.

La frontera del conjunto esta formada por tres segmentos. Vamos a estudiarlos uno a uno. En primer lugar el segmento que une el origen y el punto  $(1, 0)$ . En dicho segmento  $y = 0$  y  $x \in [0, 1]$ . Los posibles extremos son

```
(%i74) f1(x):=f(x,0)$
(%i75) solve(diff(f1(x),x),x);
(%o75) [[x=0]]
```

Nos sale el origen de coordenadas. En segundo lugar, el segmento que une  $(1, 0)$  y  $(1, 1)$ . En este caso  $x = 1$  e  $y$  varía entre 0 y 1:

```
(%i76) f2(y):=f(1,y)$
(%i77) solve(diff(f2(y),y),y);
(%o77) [ ]
```

No hay solución. El último segmento une el origen y el punto  $(1, 1)$ . En este caso:

```
(%i78) f3(x):=f(x,x);
(%i79) solve(diff(f3(x),x),x);
(%o79) a11
```

Todo el segmento está formado por puntos críticos. A estos puntos tenemos que añadir los extremos del intervalo (los vértices del triángulo): siempre cabe la posibilidad de que una función de una variable alcance un extremo absoluto en uno de ellos.

Para terminar sólo te queda evaluar. Calcular  $f(0,0)$ ,  $f(1,0)$ ,  $f(1,1)$  y  $f(x,x)$ . ¿Cuáles son los extremos absolutos de la función?

**Ejemplo 9.7.** Calculemos los extremos absolutos de la función  $f(x, y) = x^2 - y^2$  en el círculo de centro el origen y radio uno.

De nuevo estamos ante una función continua en un dominio cerrado y acotado por lo que tenemos garantizada la existencia de extremos absolutos. Los puntos críticos en el interior son

```
(%i80) f(x,y):=x^2-y^2$
(%i81) j:jacobian([f(x,y)],[x,y])$
(%i82) algsys([j[1,1],j[1,2]],[x,y]);
(%o82) [[x=0,y=0]]
```

Bien, ya tenemos un punto. ¿Que hacemos con la frontera? Calculamos los puntos críticos de la función de Lagrange

```
(%i83) F(x,y,a):=f(x,y)+a*(x^2+y^2-1)$
(%i84) J:jacobian([F(x,y,a)],[x,y])$
(%i85) algsys([J[1,1],J[1,2],x^2+y^2-1],[x,y,a]);
(%o85) [[x=1,y=0,a=-1],[x=-1,y=0,a=-1],[x=0,y=-1,a=1],[x=0,y=1,a=1]]
```

Evalúa la función  $f$  en estos puntos para averiguar dónde alcanza los extremos absolutos.

**Observación 9.8.** En la función auxiliar de Lagrange, unas veces hemos escrito  $f - ag$  y otras, como en el ejemplo anterior, hemos escrito  $f + ag$ . ¿Cuál es la versión correcta? ¿Hay alguna diferencia?

## 9.7 Ejercicios

Derivadas parciales. Plano tangente.

### Ejercicio 9.1

Sea  $f(x, y) = \ln(1 + x^2 + 2x + y^2)$ . Calcula el gradiente, la matriz hessiana de  $f$  y comprueba que es armónica, esto es, que



$$\frac{\partial^2 f}{\partial x^2}(x, y) + \frac{\partial^2 f}{\partial y^2}(x, y) = 0.$$

**Ejercicio 9.2**

Comprueba que las funciones

$$f(x, y) = 2 \arctan\left(\frac{y}{x + \sqrt{x^2 + y^2}}\right) \text{ y } g(x, y) = \arctan\left(\frac{y}{x}\right)$$

tienen las mismas derivadas parciales.

**Ejercicio 9.3**

Calcula el plano tangente y la recta normal a cada una de las superficies en el punto  $P$ :

- $z^2 - 2x^2 - 2y^2 - 12 = 0, P = (1, -1, 4).$
- $z = \ln(x^2 + y^2), P = (1, 0, 0).$
- $z + e^z + 2x + 2y - x^2 - y^2 = 3, P = (1, 1 + \sqrt{e}, 1).$

**Ejercicio 9.4**

Calcular las derivadas parciales de:

- $f(x, y, z) = x^{y+z}, \forall x \in \mathbb{R}^+, y, z \in \mathbb{R}$
- $f(x, y, z) = (x + y)^z, \forall x, y \in \mathbb{R}^+, z \in \mathbb{R}$
- $f(x, y) = \operatorname{sen}(x \operatorname{sen}(y)), \forall x, y \in \mathbb{R}$

**Ejercicio 9.5**

Sea  $f: \mathbb{R}^2 \setminus \{(0, 0)\} \rightarrow \mathbb{R}$  dada por  $f(x, y) = \log(x^2 + y^2)$  para todo  $(x, y) \neq (0, 0)$ . Se pide:

- Calcúlese el gradiente de  $f$  en todo punto así como la matriz hessiana.
- Compruébese que

$$\frac{\partial^2 f}{\partial x^2}(x, y) + \frac{\partial^2 f}{\partial y^2}(x, y) = 0 \quad \forall (x, y) \in \mathbb{R}^2 - \{(0, 0)\}.$$

**Extremos relativos****Ejercicio 9.6**

Calcular los extremos relativos de las siguientes funciones.

- $f(x, y) = x^2 + y^2 - 2x + 4y + 20.$
- $f(x, y) = x^3 + x^2y + y^2 + 2y + 5.$
- $f(x, y) = (x^2 + y^2)e^{x^2 - y^2}.$
- $f(x, y, z) = x^2 + y^2 + z^2 - 2x.$
- $f(x, y) = x^3y^3 - y^4 - x^4 + xy.$

**Extremos condicionados y absolutos****Ejercicio 9.7**

Calcular los extremos condicionados en los siguientes casos:

- $f(x, y) = x^2 - xy + y^2$  condicionados a  $x^2 + y^2 - 4 = 0.$
- $f(x, y) = x^3 + xy^2$  condicionados a  $xy = 1.$
- $f(x, y, z) = xyz$  condicionados a  $x^2 + y^2 + z^2 = 1.$

**Ejercicio 9.8**

- a) Hallar la mínima distancia de  $(0,0)$  a  $x^2 - y^2 = 1$ .
- b) Entre todos los ortoedros de volumen 1, determinar el que tiene superficie lateral mínima.
- c) Calcular las dimensiones de un ortoedro de superficie lateral 2, para que su volumen sea máximo
- d) Se quiere construir un canal cuya sección sea un trapecio isósceles de área dada  $S$ . Calcular la profundidad del canal y el ángulo que deben formar las paredes con la horizontal para que la superficie mojada sea mínima. (Solución: se trata de la mitad de un hexágono regular)

**Ejercicio 9.9**

Calcula los extremos absolutos de

- a)  $f(x, y) = x^2 - xy + y^2$  en el conjunto  $\{(x, y); x^2 + y^2 = 4\}$ ,
- b)  $f(x, y) = xyz$ , en el conjunto  $x^2 + y^2 + z^2 \leq 1$ .

# Integrales múltiples

## 10

En el Capítulo 8 hemos visto cómo podemos utilizar *Maxima* para resolver integrales de funciones de una variable. ¿Qué necesitamos para calcular integrales de funciones de varias variables? Bueno, tenemos el siguiente teorema:

**Teorema 10.1.** Sea  $f : [a, b] \times [c, d] \rightarrow \mathbb{R}$  integrable. Entonces

$$\int_{[a,b] \times [c,d]} f(x, y) d(x, y) = \int_a^b \left( \int_c^d f(x, y) dy \right) dx = \int_c^d \left( \int_a^b f(x, y) dx \right) dy.$$

<code>integrate(f(x), x)</code>	primitiva de la función $f(x)$
<code>integrate(f(x), x, a, b)</code>	$\int_a^b f(x) dx$
<code>quad_quags(f(x), x, a, b)</code>	aproximación numérica de $\int_a^b f(x) dx$
<code>romberg(f(x), x, a, b)</code>	aproximación numérica de $\int_a^b f(x) dx$

Esto quiere decir que una integral doble se calcula mediante dos integrales de una variable. Como ya sabemos hacer integrales de una variable, podemos comprobarlo en un ejemplo. La integral de la función  $f(x, y) = 3xy$  en  $[0, 1] \times [2, 6]$  vale

```
(%i1) integrate(integrate(3*x*y, x, 0, 1), y, 2, 6);
(%o1) 24
```

o, cambiando el orden de integración

```
(%i2) integrate(integrate(3*x*y, y, 2, 6), x, 0, 1);
(%o2) 24
```

### Recintos de tipo 1 y 2

Como recordarás de clase, sabemos calcular la integral de una función  $h(x, y)$  en recintos de alguno de los siguientes tipos:

$$A = \{(x, y) \in \mathbb{R}^2 : a \leq x \leq b, f(x) \leq y \leq g(x)\} \text{ o}$$

$$B = \{(x, y) \in \mathbb{R}^2 : a \leq y \leq b, f(y) \leq x \leq g(y)\}$$

Las correspondientes integrales serían, vía el teorema de Fubini,

$$\int_A h(x, y) d(x, y) = \int_a^b \left( \int_{f(x)}^{g(x)} h(x, y) dy \right) dx, \text{ y}$$

$$\int_B h(x, y) d(x, y) = \int_a^b \left( \int_{f(y)}^{g(y)} h(x, y) dx \right) dy.$$

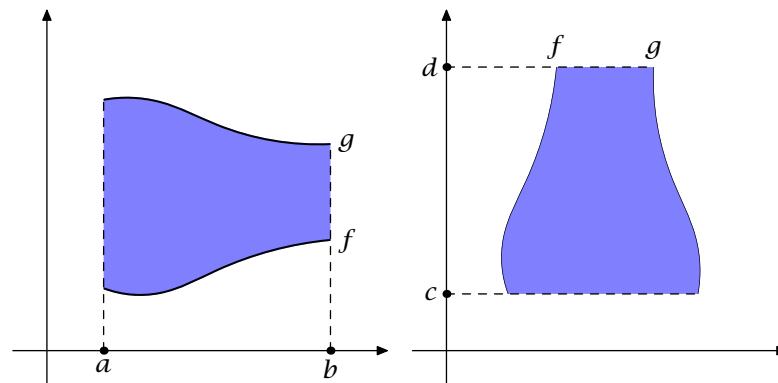


Figura 10.1 Recintos de tipo 1 y 2

El caso más fácil, ya lo hemos visto, se presenta cuando integramos en rectángulos. En este caso las funciones  $f$  o  $g$  son constantes. Por ejemplo, si queremos calcular la integral de  $h(x, y) = x^2 + y^2$  en  $[0, 2] \times [1, 5]$  tendríamos que hacer lo siguiente:

```
(%i3) integrate(integrate(x^2 + y^2, x, 0, 2), y, 1, 5);
(%o3) 280
      3
```

Pasemos a integrales un poco más complicadas. Supongamos que queremos integrar la función  $f(x, y) = x$  en el conjunto

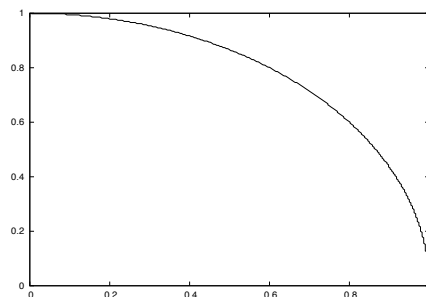
$$A = \{(x, y) \in \mathbb{R}^2 : x^2 + y^2 \leq 1, x \geq 0, y \geq 0\}.$$

El primer paso es intentar averiguar cómo es el conjunto, para lo que nos puede ser útil dibujar  $x^2 + y^2 = 1$ . Para ello nos hace falta cargar el paquete draw.

```
(%i4) load(draw)$
```

El rango de variación de las variables  $x$  e  $y$  lo tenemos que decidir nosotros. En nuestro caso,  $x$  e  $y$  tienen que ser mayores o iguales que cero y menores que 1.

```
(%i5) draw2d(implicit(x^2+y^2=1, x, 0, 1, y, 0, 1));
(%o5)
```



Después de ver el conjunto  $A$  donde estamos integrando, es fácil ver cómo podemos poner una variable en función de la otra. Por ejemplo, si ponemos  $y$  en función de  $x$ , para un valor concreto de  $x$  entre 0 y 1,  $y$  varía entre 0 y  $\sqrt{1 - x^2}$ . Podemos calcular ahora el valor de la integral:

```
(%i6) integrate(integrate(x,y,0,sqrt(1-x^2),x,0,1);
Is (x-1)*(x+1) positive, negative, or zero? positive;
Defint: Upper limit of integration must be real.
-- an error. To debug this try debugmode(true);
```

Después de decirle a *Maxima* que  $(x-1)*(x+1)$  es positivo, *Maxima* da un error y no sabe resolver la integral. Bueno, parece que no nos va a ser de gran ayuda para calcular una integral que parecía fácil. Hagamos un último intento. La primera integral no es difícil, de hecho es inmediato calcular una primitiva (incluso sin usar *Maxima*)

```
(%i7) integrate(x,y);
(%o7) xy
```

Vale, apliquemos nosotros mismos la regla de Barrow y tenemos que

$$\int_0^{\sqrt{1-x^2}} x \, dy = xy \Big|_0^{\sqrt{1-x^2}} = x\sqrt{1-x^2}.$$

Por último, calculemos

```
(%i8) integrate(x*sqrt(1-x^2),x,0,1);
(%o8) 1/3
```

¡Conseguido! Nos ha costado un poco pero tenemos resuelta la integral doble.

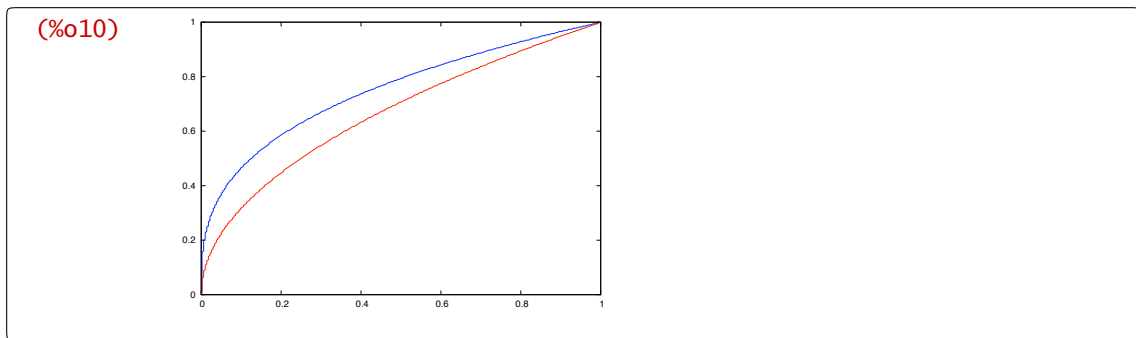
Observarás que no hemos cambiado a coordenadas polares. Para integrales en varias variables, interesa cambiar de variable cuando el dominio o la función son complicados. En el primer caso, si no podemos escribir el dominio de una de las formas (1) o (2) tendremos que cambiar de variable nosotros (*Maxima* no lo hace automáticamente). El caso de funciones complicadas no nos debe preocupar, pues *Maxima* se encarga de resolver la integral en la inmensa mayoría de los casos, y si no se puede resolver directamente, siempre queda la posibilidad de aproximar el valor de integral.

Veamos un par de ejemplos.

**Ejemplo 10.2.** Calcular la integral de la función  $f(x, y) = e^{x/y}$ , en el conjunto  $A = \{(x, y) \in \mathbb{R}^2 : 0 \leq y^3 \leq x \leq y^2\}$ .

Estamos ante un conjunto del tipo 2. Tenemos la variable  $x$  en función de  $y$ . Vamos primero a ver los puntos de corte de las ecuaciones  $y^3 = x$ ,  $y^2 = x$  y luego las dibujaremos para hacernos una idea de cuál es el dominio de integración:

```
(%i9) solve(y^3=y^2,y)
(%o9) [y=0,y=1]
(%i10) draw2d(color=blue,
implicit(y^3=x,x,0,1,y,0,1),
color=red,
implicit(y^2=x,x,0,1,y,0,1));
```



De lo hecho se deduce que la integral que queremos calcular es

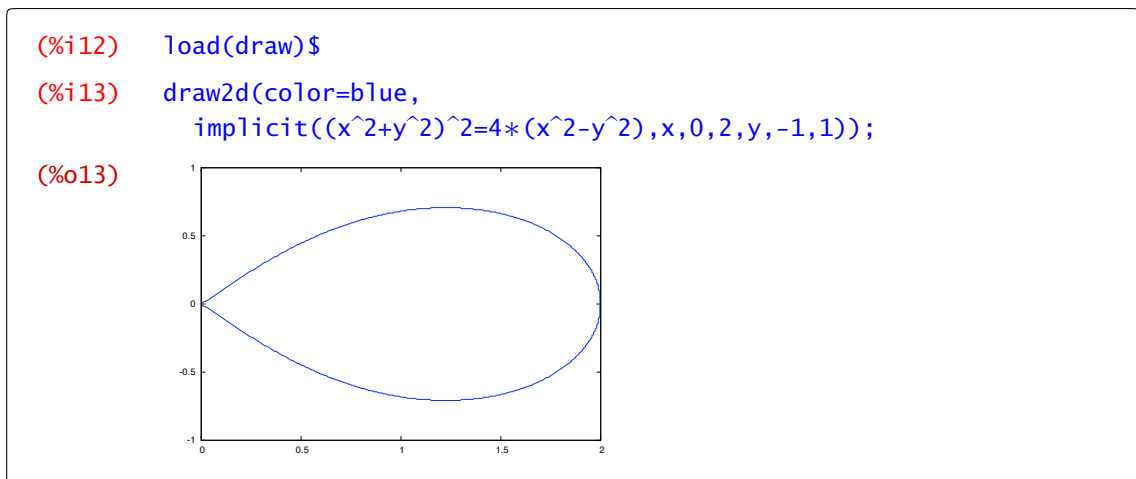
(%i11) `integrate(integrate(exp(x/y), x, y^3, y^2), y, 0, 1);`

(%o11)  $-\frac{\%e-3}{2}$

**Ejemplo 10.3.** Calcular la integral de la función  $f(x, y) = x^2 + y^2$  en el conjunto

$$A = \{(x, y) \in \mathbb{R}^2 : (x^2 + y^2)^2 \leq 4(x^2 - y^2), x \geq 0\}.$$

En este caso la función es relativamente sencilla pero el dominio no lo es tanto. No se ve una manera fácil de escribir la integral en términos de  $x$  e  $y$ . Parece mejor cambiar a coordenadas polares. Antes de hacerlo podemos echar un vistazo al dominio. Recuerda que el rango de valores de las variables  $x$  e  $y$  es algo que debes decidir por ti mismo.



Las condiciones del dominio escritas en coordenadas polares son:

$$\rho^4 \leq 4\rho^2(\cos^2(\theta) - \sin^2(\theta)), \rho \cos \theta \geq 0.$$

Después de operar, nos quedaría

$$\rho \leq 2\sqrt{\cos^2(\theta) - \sin^2(\theta)} \text{ y } \theta \in \left[-\frac{\pi}{4}, \frac{\pi}{4}\right],$$

y teniendo en cuenta que  $\cos^2(\theta) - \sin^2(\theta) = \cos(2\theta)$ , tenemos que resolver la integral

$$\int_{-\pi/4}^{\pi/4} \left( \int_0^{2\sqrt{\cos(2\theta)}} \rho^3 d\rho \right) d\theta$$

```
(%i14) integrate(integrate(r^3,r,0,2*sqrt(cos(2*t))),t,-%pi/4,%pi/4);
Is cos(2*t) positive, negative, or zero? positive;
```

```
(%o14)  $\pi$ 
```

## 10.1 Ejercicios

### Ejercicio 10.1

Calcular la integral de las siguientes funciones en los recintos determinados por las siguientes ecuaciones:

- $f(x, y) = \sqrt{4x^2 - y^2}$ ,  $x = 1$ ,  $y = 0$ ,  $y = x$
- $f(x, y) = xe^{-x^2/y}$ ,  $x = 0$ ,  $y = 1$ ,  $y = x^2$
- $f(x, y) = y$ ,  $y \geq 0$ ,  $x^2 + y^2 = a^2$ ,  $y^2 = 2ax$ ,  $x = 0$

### Ejercicio 10.2

Calcular el volumen del conjunto  $A$  en cada uno de los siguientes casos:

- $A = \{(x, y, z) \in \mathbb{R}^3 : x^2 + y^2 \leq z \leq \sqrt{x^2 + y^2}\}$
- $A = \{(x, y, z) \in \mathbb{R}^3 : \frac{x^2}{a^2} + \frac{y^2}{b^2} \leq 1, 0 \leq z \leq \sqrt{\frac{x^2}{a^2} + \frac{y^2}{b^2}}\}$
- $A = \{(x, y, z) \in \mathbb{R}^3 : 0 \leq z \leq x^2 + y^2, x + y \leq 1, x, y \geq 0\}$
- $A = \{(x, y, z) \in \mathbb{R}^3 : 0 \leq z \leq \sqrt{x^2 + y^2}, x^2 + y^2 \leq 2y\}$
- $A = \{(x, y, z) \in \mathbb{R}^3 : 0 \leq z \leq 4 - y^2, 0 \leq x \leq 6\}$
- $A = \{(x, y, z) \in \mathbb{R}^3 : \sqrt{x} \leq y \leq 2\sqrt{x}, 0 \leq z \leq 9 - x\}$
- $A = \{(x, y, z) \in \mathbb{R}^3 : x^2 + y^2 \leq z^2, x^2 + y^2 + z^2 \leq 2z\}$

### Ejercicio 10.3

Calcular el volumen limitado superiormente por el cono  $4x^2 + 4y^2 - z^2 = 0$ , inferiormente por el plano  $z = 0$  y lateralmente por el cilindro  $x^2 + (y - 2)^2 = 4$ .

### Ejercicio 10.4

Calcular el volumen de los sólidos siguientes:

- Intersección de los cilindros  $x^2 + y^2 = r^2$ ,  $y^2 + z^2 = r^2$ .
- El limitado por el plano  $z = 0$ , el cilindro  $x^2 + y^2 = 2ax$  ( $a > 0$ ) y el cono  $x^2 + y^2 = z^2$ .





# Ecuaciones diferenciales ordinarias

## 11

### 11.1 Resolución de edos de primer y segundo orden

La posibilidad de resolver una ecuación diferencial que has visto en clase pasa por ser capaz de encuadrarla dentro de un tipo que sepamos resolver: decidir si es lineal, de variables separadas, o cualquier otro. Al igual que `integrate` calcula integrales sin decirnos de qué tipo es la función que estamos integrando, aquí va a pasar lo mismo. Vamos a tener una orden que resuelve ecuaciones diferenciales y punto.

```
ode2(ecuación diferencial, y, x)  resuelve la ecuación diferencial
```

La orden `ode2` resuelve algunas ecuaciones diferenciales ordinarias de primer y segundo orden. Para escribir una de estas ecuaciones utilizamos el comando `diff` teniendo la precaución de utilizar la comilla, `'`, para evitar que *Maxima* calcule la derivada. Comencemos por una ecuación de variables separadas: la definimos

```
(%i1)  edo:(2*x+3)*'diff(y,x)+y*(x-1)=0;
(%o1)  (2x+3)  $\frac{d}{dx}$  y+(x-1)y=0
```

y la resolvemos

```
(%i2)  ode2(edo, y, x);
(%o2)  y=%c %e  $\frac{5 \log(2x+3)}{4} - \frac{x}{2}$ 
```

La solución, en este caso dada en forma explícita, depende de un parámetro `%c`. Hay veces en que *Maxima* da la solución en forma implícita. Por ejemplo, la ecuación de variables separadas

```
(%i3)  edo1:x*'diff(y,x)+cos(y)*(x-1)=0;
(%o3)  x  $\frac{d}{dy}$  y+cos(y)(x-1)=0
(%i4)  ode2(edo1, y, x);
(%o4)   $-\frac{\log(\sin(y)+1)-\log(\sin(y)-1)}{2} = -\log(x)+x+%c$ 
```

En este caso, *Maxima* no es capaz de despejar `y` en función de `x`, pero esto no siempre es así:

```
(%i5)  ode2(x*'diff(y,x)+y^2*(x-1)=0, y, x);
```

$$(\%o5) \quad \frac{1}{y} = -\log(x) + x + \%c$$

$$(\%i6) \quad \text{solve}(\%, y)$$

$$(\%o6) \quad [y = -\frac{1}{\log(x) - x - \%c}]$$

También podemos resolver ecuaciones exactas,

$$(\%i7) \quad \text{ode2}(x^2*(3*y^2+1)+y*(2*x^3+y)*'diff(y,x), y, x);$$

$$(\%o7) \quad \frac{y^3+3x^3y^2+x^3}{3} = \%c$$

ecuaciones lineales,

$$(\%i8) \quad \text{ode2}('diff(y,x)=x+y, y, x);$$

$$(\%o8) \quad y = (-x-1)*e^{-x} + \%c * e^x$$

ecuaciones homogéneas, etc.

$$(\%i9) \quad \text{ode2}((x^2+y^2)+x*y*'diff(y,x), y, x);$$

$$(\%o9) \quad \frac{2*x^2*y^2+x^4}{4} = \%c$$

Incluso podemos resolver ecuaciones de segundo orden, ya sean homogéneas,

$$(\%i10) \quad \text{ode2}('diff(y,x,2)-3*'diff(y,x)+2*y, y, x);$$

$$(\%o10) \quad y = \%k1 * e^{2*x} + \%k2 * e^x$$

o no.

$$(\%i11) \quad \text{ode2}('diff(y,x,2)-3*'diff(y,x)+2*y=2*x, y, x);$$

$$(\%o11) \quad y = \%k1 * e^{2*x} + \%k2 * e^x + \frac{2x+3}{2}$$

### 11.1.1 Condiciones iniciales o de contorno

En todas las soluciones de las ecuaciones diferenciales anteriores aparecen una o dos constantes dependiendo de si se trata de ecuaciones de primer o segundo orden. Volvamos, por ejemplo a la primera ecuación de este tema y consideremos el problema de valores iniciales

$$(2x + 3)y' + y(x - 1) = 0$$

$$y'(0) = 2$$

La solución era

```
(%i12) ode2(edo, y, x);
(%o12) y=%c %e5log(2x+3)/4 - x/2
```

<code>ic1(ecuación, x=a, y=b)</code>	resuelve problema de valores iniciales de primer orden
<code>ic2(ecuación, x=a, y=b, diff(y, x)=c)</code>	resuelve problema de valores iniciales de segundo orden
<code>bc2(ecuación, x=a, y=b, x=c, y=d)</code>	resuelve problema de contorno

El comando `ic1` permite ajustar el valor de la constante `%c`:

ic1

```
(%i13) ic1(% , x=0, y=2);
(%o13) y=2 * %e5log(2x+3)/4 - x/2 - 5log(3)/4
```

y el comando `ic2` juega el mismo papel para ecuaciones de segundo orden.

ic2

**Ejemplo 11.1.** Vamos a resolver el problema de valores iniciales

$$\begin{aligned}
 y'' - y' &= e^x, \\
 y(0) &= 0, \\
 y'(0) &= 1.
 \end{aligned}$$

```
(%i14) ode2('diff(y, x, 2) - 'diff(y, x)=%e^x, y, x);
(%o14) y=(x-1)%ex+%k1%ex+%k2
(%i15) ic2(% , x=0, y=0, diff(y, x)=1);
(%o15) y=(x-1)%ex+%ex
```

Los problemas de valores de contorno se resuelven de manera similar pero con `bc2`. Por ejemplo la solución del problema

bc2

$$\begin{aligned}
 y'' - y' &= e^x, \\
 y(0) &= 0, \\
 y(1) &= 0.
 \end{aligned}$$

la calculamos rápidamente aprovechando el ejemplo anterior:

```
(%i16) bc2(%o14, x=0, y=0, x=1, y=0);
(%o16) y=(x-1)%ex - %ex/%e-1 + %e/%e-1
```

## 11.2 Resolución de sistemas de ecuaciones lineales

**desolve** La orden **desolve** resuelve ecuaciones o sistemas de ecuaciones diferenciales ordinarias lineales mediante la transformada de Laplace.

<code>desolve(edos, vars)</code>	resuelve la ecuación diferencial lineal
<code>atvalue(f(x), x=a, b)</code>	$f(a) = b$

La forma de escribir las ecuaciones o el sistema de ecuaciones es muy parecida a los comandos que hemos visto, pero hay que indicar explícitamente todas las dependencias entre variables. Por ejemplo, en lugar de escribir `'diff(f, x)` tendremos que escribir `'diff(f(x), x)` en el caso de que  $f$  sea función de  $x$ . Veamos un ejemplo.

Para resolver la ecuación diferencial  $y'' - 2y = 10x$  escribimos

```
(%i17) desolve('diff(y(x), x, 2)-2*y(x)=10*x, [y(x)]);
```

```
(%o17) y(x) =  $\frac{\sinh(2^{\frac{1}{2}}x) \left( \frac{d}{dx} y(x) \Big|_{x=0} + 5 \right)}{\sqrt{2}} + y(0) \cosh(2^{\frac{1}{2}}x) - 5x$ 
```

y la solución del sistema de ecuaciones

$$x'(t) = 3x - 4y$$

$$y'(t) = 4x - y$$

se obtiene mediante

```
(%i18) desolve(['diff(x(t), t)=3*x(t)+4*y(t), 'diff(y(t), t)=4*x(t)-y(t)], [x(t), y(t)]);
```

```
(%o18) [x(t)=e^t \left( \frac{(2(4y(0)+x(0))+2x(0))\sinh(2\sqrt{5}t)}{4\sqrt{5}} + x(0)\cosh(2\sqrt{5}t) \right),  
y(t)=e^t \left( \frac{(2y(0)+2(4x(0)-3y(0)))\sinh(2\sqrt{5}t)}{4\sqrt{5}} + y(0)\cosh(2\sqrt{5}t) \right)]
```

Una cuestión importante cuando queramos añadir condiciones iniciales *en el origen* es que éstas deben de escribirse antes de resolver el sistema. Por ejemplo, si queremos resolver

$$x'(t) = 3x - 4y$$

$$y'(t) = 4x - y$$

$$x(0) = 1$$

$$y(0) = 0$$

**atvalue** primero indicamos el valor de  $x$  e  $y$  con el comando **atvalue**

```
(%i19) atvalue(x(t), t=0, 1);
```

```
(%o19) 1
```

```
(%i20) atvalue(y(t), t=0, 0);
```

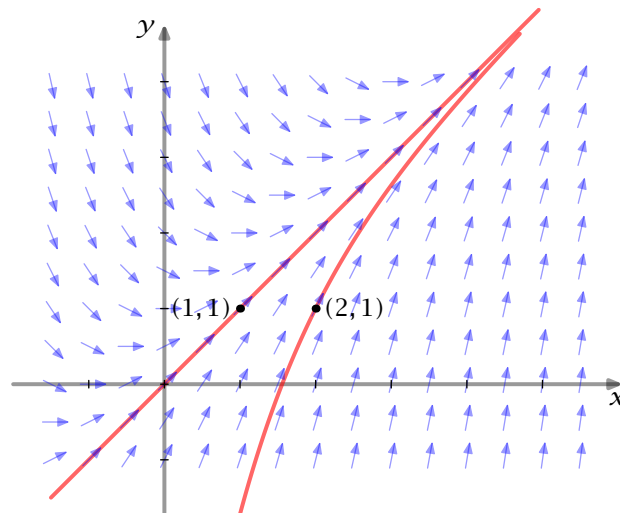
```
(%o20) 0
```

y, después de esto, resolvemos el sistema

```
(%i21) solve(['diff(x(t),t)=3*x(t)+4*y(t)', 'diff(y(t),t)=4*x(t)-y(t)'],
             [x(t),y(t)]);
(%o21) [x(t)=%e^t * (sinh(2*sqrt(5)*t)/sqrt(5) + cosh(2*sqrt(5)*t)), y(t)=2*%e^t * sinh(2*sqrt(5)*t)/sqrt(5)]
```

### 11.3 Campos de direcciones y curvas integrales

Hemos visto en clase que podemos hacernos una idea de cuál es la solución de una ecuación diferencial de primer orden  $y' = f(x, y)$ . Lo que hacemos es dibujar en cada punto del plano  $(x, y)$  un segmento que nos indique la pendiente en ese punto, dada por  $f(x, y)$ .



**Figura 11.1** Campo de pendientes y curvas integrales

Una vez que tenemos las pendientes dibujadas, una curva es solución de la ecuación diferencial si en cada punto sigue las direcciones marcadas. Este dibujo puede, por tanto, darnos información sobre las soluciones de una ecuación diferencial.

```
plotdf(func, opciones) dibuja el campo de direcciones dado por func
```

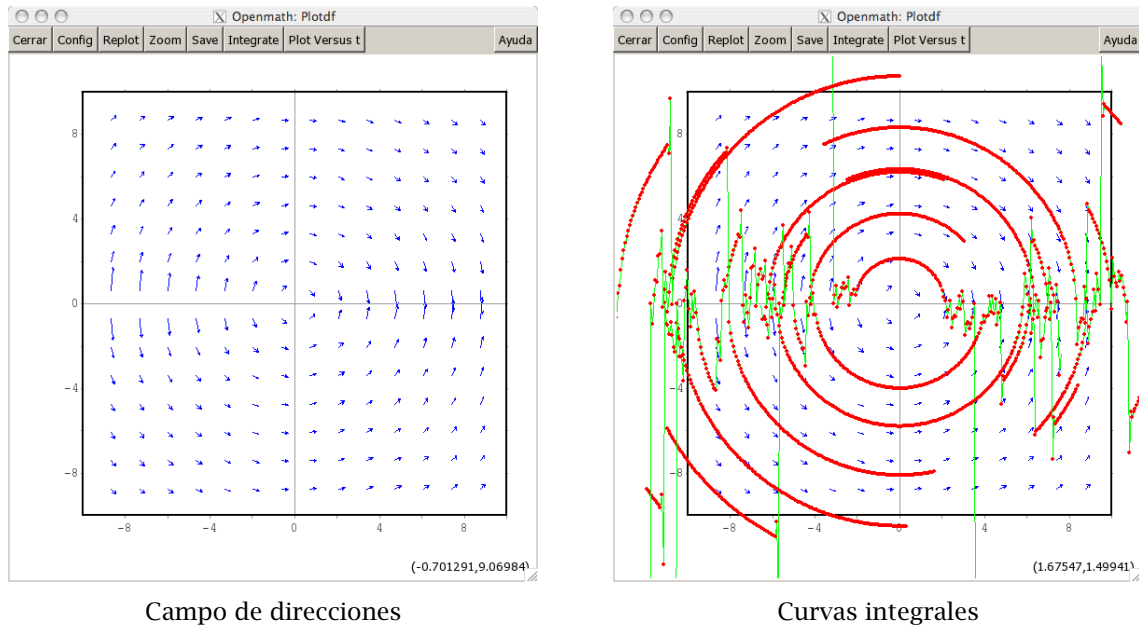
El módulo `plotdf` nos permite realizar este tipo de gráficos. Como todos los módulos adicionales, comenzamos cargándolo.

```
(%i22) load(plotdf)$
```

En su versión más fácil, podemos dibujar el campo de direcciones de la ecuación  $y' = -x/y$  de la siguiente forma:

```
(%i23) plotdf(-x/y);
```

y obtenemos la ventana de la Figura 11.2.



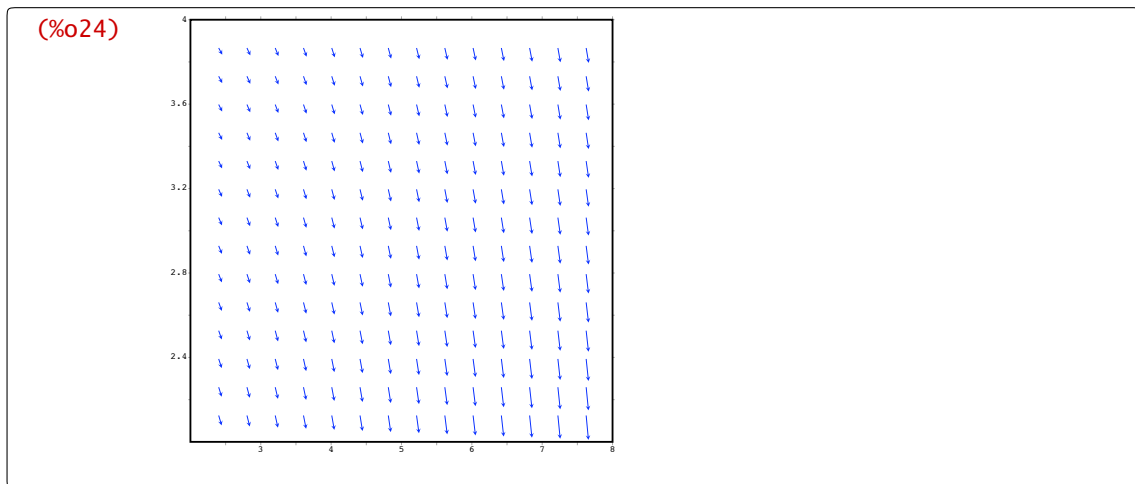
**Figura 11.2** Plotdf y Openmath

En primer lugar, observa que estamos utilizando *Openmath* para representar los gráficos y no *Gnuplot* como venía siendo habitual. En segundo lugar, ¿qué pasa cuándo pinchas con el ratón en un punto cualquiera de la gráfica? Lo que ocurre es que se representa la correspondiente curva integral que pasa por dicho punto. Puedes repetir la operación las veces que quieras y vas obteniendo una imagen parecida a la segunda de la Figura 11.2.

## Opciones

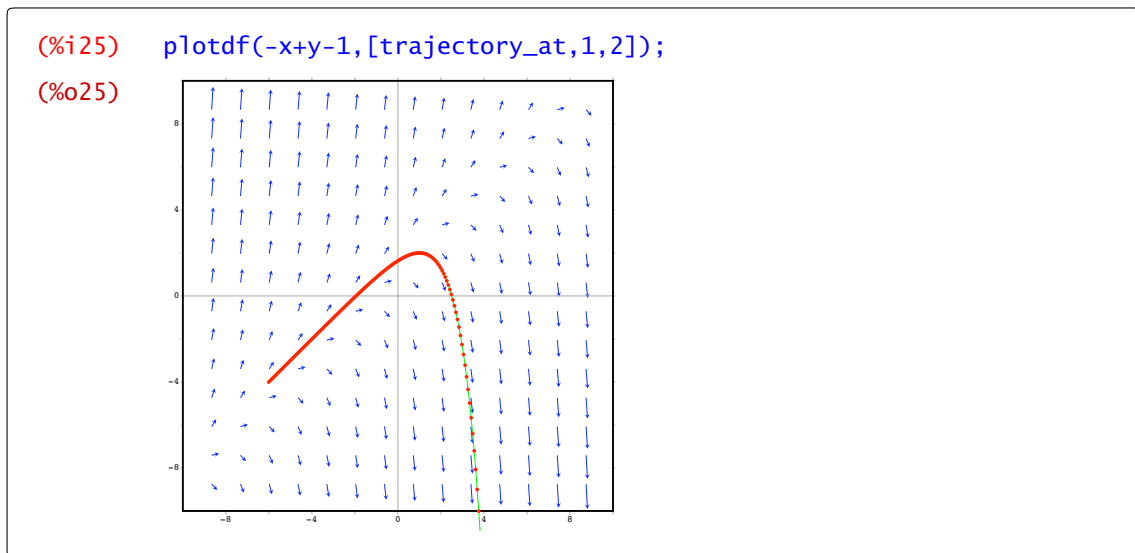
Hay muchas opciones con las que modificar el comportamiento de la orden `plotdf`. Quizá la primera que alguno se pregunte es cómo variar el rango que se representa de  $x$  e  $y$ . A diferencia de otros gráficos, aquí se da el centro y el “radio” del gráfico. Por defecto el centro es el origen y el radio en ambos ejes es 10 con lo que las representaciones se hacen en  $[-10, 10] \times [-10, 10]$ . Las opciones `xcenter`, `xradius`, `ycenter` e `yradius` nos permiten variar esto.

```
(%i24) plotdf(-x/y, [xradius, 3], [xcenter, 5], [yradius, 1], [ycenter, 3]);
```

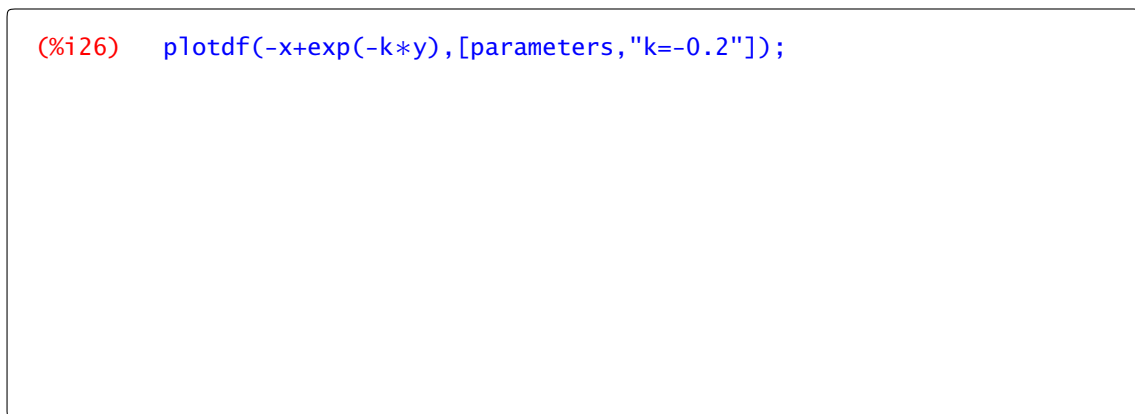


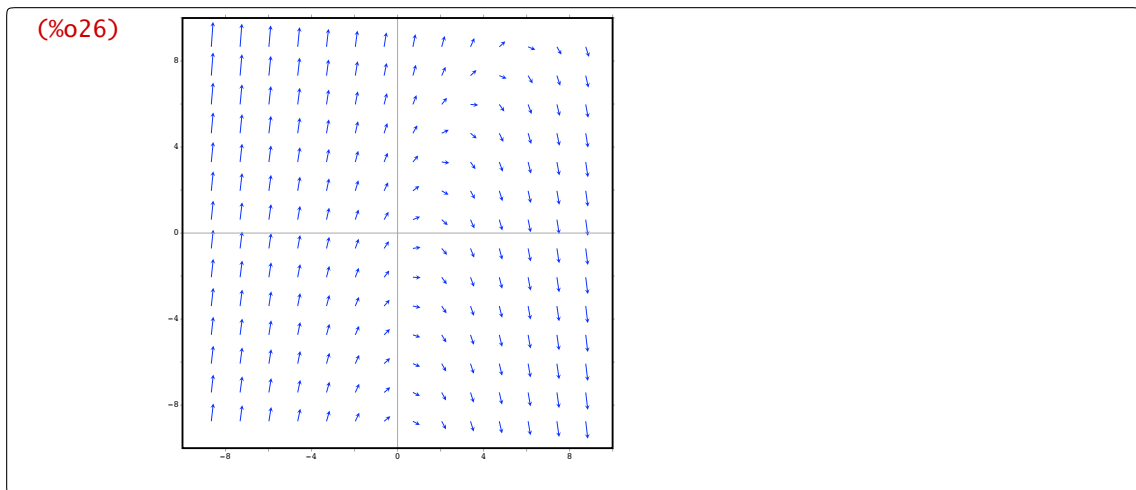
Podemos fijar de partida la curva integral que deseamos dibujar eligiendo el punto por el que pasa con la opción `trajectory_at`. Por ejemplo la curva integral de la ecuación  $y' = -x + y - 1$  que pasa por el punto (1, 2) tiene el siguiente aspecto.

`trajectory_at`



Es relativamente común encontrar ecuaciones diferenciales que dependen de algún parámetro. La opción `parameters` permite pasar valores de los parámetros a `plotdf`





aunque claro, lo interesante sería ver qué ocurre cuando damos distintos valores al parámetro. Eso es justamente lo que permite la opción `sliders`: dar varios valores al parámetro. Por ejemplo

```
(%i27) plotdf(-x+exp(-k*y), [parameters, "k=-0.2"], [sliders, "k=-3:3"]);
```

nos da como resultado la ventana de la Figura 11.3

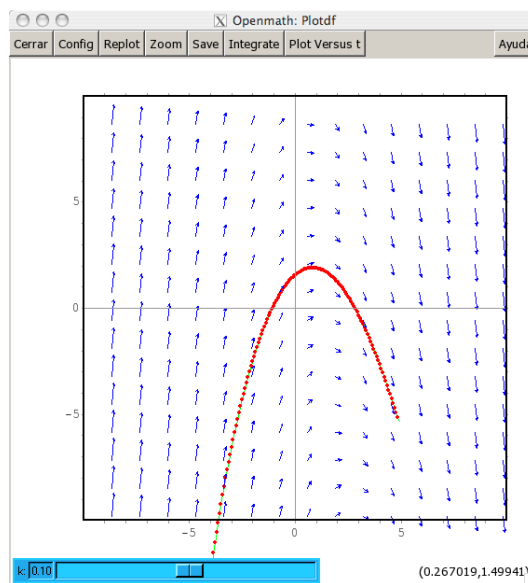


Figura 11.3 Parámetros en plotdf

Si te fijas, en la parte inferior tienes un botón que te permite cambiar los valores del parámetro  $k$ . Como hemos escrito la opción `[parameters, "k=-0.2"]`, la gráfica que presenta *Maxima* es justamente esa y, por supuesto, podemos seguir dibujando curvas integrales que siguen cambiando automáticamente cuando variamos el valor de parámetro.



## 11.4 Ejercicios

### Ejercicio 11.1

Resuelve las ecuaciones diferenciales:

- |   |  |
|---|--|
| a) $(2x + 3)y' + xy = 0.$                                       | h) $2x^2 \frac{dy}{dx} = 3xy + y^2; \quad y(1) = -2$       |
| b) $y' = x + y$   | i) $y' + 2y = e^{-x}$                                      |
| c) $y' - \frac{\cos(x)}{\sin(x)} y = 4 \operatorname{sen}^2(x)$ | j) $x dy + y dx = \operatorname{sen}(x)$                   |
| d) $y \cos(x) dx + \operatorname{sen}(x) dy = 0,$               | k) $y'' + 3y' - 4y = 0, \quad y(0) = 2, \quad y'(0) = -3$  |
| e) $(x^2 + y^2) dx + xy dy = 0.$                                | l) $y'' + 4y = 0, \quad y(\pi/6) = 1, \quad y'(\pi/6) = 0$ |
| f) $(x^2 - y^2) dx + x^2 dy = 0.$                               | m) $y'' - y = xe^{3x}, \quad y(0) = 0, \quad y'(0) = 1$    |
| g) $(x + y + 1) dx - (x - y + 1) dy = 0.$                       |  |

### Ejercicio 11.2

Resolver los siguientes sistemas de ecuaciones diferenciales:

a)

$$\begin{aligned} x' - 4y &= 1 \\ x + y' &= 2 \end{aligned}$$

b)

$$\begin{aligned} 2x' - 5x + y' &= e^t \\ x' - x + y' &= 5e^t \end{aligned}$$

### Ejercicio 11.3

Hallar las curvas de  $\mathbb{R}^2$  que verifican la siguiente propiedad: en cada punto el vector de posición y el vector tangente son ortogonales.

### Ejercicio 11.4

Sabiendo que la semivida del Carbono 14 radioactivo (C-14) es aprox. 5600 años, determinar la edad de un fósil que contiene 1/1000 de la cantidad original de C-14.

### Ejercicio 11.5

Utiliza el método de las poligonales de Euler para aproximar la solución del problema de valores iniciales

$$y' = x + y, \quad y(0) = 1.$$



## Avisos y mensajes de error

### A

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  $\sin(x) = 0$  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.



---

## Bibliografía

### B

---

- 1) La primera fuente de documentación sobre *Maxima* es el propio programa. La ayuda es muy completa y detallada.
- 2) 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.
- 3) “*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/>
- 4) “*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/>
- 5) 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/>
- 6) 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.





---

## Índice alfabético

---

- ' 18
- " 112
- . 69
- ?? 26
- % 9
  
- a**
- abs 84
- acos 14
- algsys 91, 161
- allroots 92
- apply 71
- asin 14
- atan 14
- atvalue 178
  
- b**
- base 53
- bc2 177
- bfloat 10
- block 125
- both 54
  
- c**
- carg 84
- charpoly 79
- color 52
- contour 53
- cos 13, 85
- cosh 85
- cot 14
- csc 14
- cylindrical 56
  
- d**
- define 29, 112
- demo 28
- demoivre 85
- denom 21
- dependencies 157
- depends 156
- describe 26
- desolve 178
- determinant 75
- diagmatrix 77
- diff 99, 111-112, 114, 118, 175
- discrete 40
- do 95
- draw 47
- draw2d 47
- draw3d 47
  
- e**
- eigenvalues 79, 161
- eigenvectors 79
- ellipse 59
- enhanced3d 50
- entermatrix 77
- entier 97
- erf 134
- ev 23
- example 27
- exp 12, 84
- expand 20
- explicit 47, 54
- exponentialize 85
  
- f**
- factor 22
- fill\_color 50-51, 144
- filled\_func 50, 144
- find\_root 93, 186
- first 68
- flatten 69
- float 8, 10
- for 94
- fpprec 11
- fullratsimp 24
- functions 30
- fundef 31
  
- g**
- genmatrix 78
- grid 49
  
- h**
- hessian 152

**i**

ic1 177  
 ic2 177  
 if 96  
 implicit 57  
 ind 104  
 integrate 133  
 inverse 75

**j**

jacobian 151

**k**

key 53  
 kill 18

**l**

last 68  
 length 69  
 lhs 86  
 limit 103  
 lines 42  
 line\_width 52  
 linsolve 90  
 lista 67  
 log 12, 84  
 logexpand 21

**m**

makelist 59, 70, 140  
 map 54, 71, 89  
 matrix 72  
 matrixp 73  
 matriz\_size 72  
 minor 76  
 mnewton 100  
 multiplicities 87

**n**

niceindices 129  
 nticks 37, 52, 59  
 nullspace 76  
 num 21  
 numer 10  
 nusum 127

**o**

ode2 175

**p**

parameters 181  
 parametric 47, 55  
 parametric\_surface 56  
 part 68, 88  
 partfrac 20  
 plot2d 33, 38  
 plot3d 43  
 plotdf 179  
 points 41, 59  
 point\_size 50  
 point\_type 50  
 polar 56  
 polarform 84  
 powerseries 129  
 print 95

**q**

quadpack 138  
 quad\_qagi 138  
 quad\_quags 138

**r**

radcan 24  
 radexpand 21  
 random 15, 59, 140-141  
 range 61, 70  
 rank 76  
 ratsimp 24  
 realonly 91  
 realpart 84  
 realroots 92  
 recta  
   normal 117  
   tangente 115  
 rectangle 58  
 rectform 83  
 remfunction 30  
 remove 157  
 remvalue 18  
 rhs 86, 88  
 romberg 138

**s**

sec 14  
 second 68  
 simpsum 127  
 sin 13, 85  
 sinh 85

sliders 182  
 solve 86, 89  
 solve\_rec 106  
 sort 69  
 spherical 57  
 sqrt 8  
 style 41  
 submatrix 76  
 sum 127, 140  
 surface 53  
 surface\_hide 53

**t**

tan 13  
 taylor 123, 129  
 tenth 68  
 teorema  
     de Newton-Raphson 99  
 title 49  
 trajectory\_at 181  
 transpose 75  
 triangularize 76  
 trigexpand 24-25  
 trigexpandplus 25  
 trigexpandtimes 25  
 trigreduce 24  
 trigsimp 24

**u**

und 104  
 unique 70

unless 95

**v**

values 18  
 vector 60

**w**

while 95  
 with\_slider 61, 117, 125  
 with\_slider\_draw 62, 154-156  
 with\_slider\_draw3d 62, 147

**x**

xaxis 50  
 xcenter 180  
 xlabel 50  
 xradius 180  
 xrange 48

**y**

yaxis 50  
 ycenter 180  
 ylabel 50  
 yradius 180  
 yrange 48

**z**

zaxis 50  
 zlabel 50  
 zrange 48



